Penetration Testing

AnimeBlast Attack

10203360

Created by: Hamza AL-Risheq

Submitted to: Eng. Sami AlMashaqbeh

2023-2024 Spring

Table of Contents

[Part 2: Penetration Testing foundations 4](#_Toc168688587)

[Part 2.1: Strategic Jumping: Crafting Successful Memory Exploits 4](#_Toc168688588)

[Part 2.2: Memory Corruption 6](#_Toc168688589)

[Part 2.2.1: Memory Corruption Techniques 6](#_Toc168688590)

[Part 2.2.2: Memory Corruption Mitigation 7](#_Toc168688591)

[Part 2.3: Privilege Escalation 8](#_Toc168688592)

[Part 2.3.1 Privilege Escalation Definition 8](#_Toc168688593)

[Part 2.3.2 Privilege Escalation Types 9](#_Toc168688594)

[Part 2.3.3 Common Types of Privilege Escalation Techniques 9](#_Toc168688595)

[Part 2.3.4 How to Prevent Privilege Escalation Attacks 10](#_Toc168688596)

[Part 3: Target #1 Exploit-Me server 10](#_Toc168688597)

[Part 3.a. Automated Port Scanning Using Bash Script 10](#_Toc168688598)

[Part 3.a.a. Screenshot of the code 10](#_Toc168688599)

[Part 3.a.b. Description of the bash script: 10](#_Toc168688600)

[Part 3.a.c. Results of the scanning: 11](#_Toc168688601)

[Part 3.b. Efficient Network Scanning: Custom Python Scripts for Identifying Vulnerable Services 12](#_Toc168688602)

[Part 3.b.a. Screenshot of the code 12](#_Toc168688603)

[Part 3.b.b. Description of the code 13](#_Toc168688604)

[Part 3.a.c. Results of the scanning: 13](#_Toc168688605)

[Part 3.c. Utilizing Fuzzers and Debuggers for Memory Analysis and Software Crash Detection 14](#_Toc168688606)

[Part 3.d. Creating Short and Back Long Jumps with NASM Tool 17](#_Toc168688607)

[Part 3.e. Python Script to Do Buffer Overflow Attack Using JMP ESP Technique 18](#_Toc168688608)

[Part 3.f. Exploiting Target Systems with Message Box and Shellcode Payloads 20](#_Toc168688609)

[Part 3.f.a. Message Box 20](#_Toc168688610)

[Part 3.f.b. Shellcode 21](#_Toc168688611)

[Part 3.g. Setting Up and Exploiting with Metasploit Multi/Handler 23](#_Toc168688612)

[Part 4: Target #2 AnimeBlast server 23](#_Toc168688613)

[Part 4.a: Scanning the Target for Open Ports Using Gateway Session 23](#_Toc168688614)

[Part 4.b: Creating Tunnels and Pivoting to Attack the AnimeBlast Server 24](#_Toc168688615)

[Part 4.c: Identify the vulnerable services using Metasploit auxiliary 25](#_Toc168688616)

[Part 4.d: Identifying and Exploiting a SQL Injection Vulnerability 25](#_Toc168688617)

[Part 4.e: Demonstrating a Cross-Site Scripting (XSS) Vulnerability and Its Potential Exploits 27](#_Toc168688618)

[Part 4.f: Finding and Exploiting a Remote Code Execution (RCE) Vulnerability 28](#_Toc168688619)

[Part 4.g: Brute Forcing FTP Credentials: Discovering Usernames and Passwords 29](#_Toc168688620)

[Part 4.h: Exploiting a File Upload Vulnerability to Deploy a PHP Shell 30](#_Toc168688621)

[Part 4.i: CTF Challenge: Discover and Decode the Hidden Flags 31](#_Toc168688622)

[References 32](#_Toc168688623)

# Part 2: Penetration Testing foundations

## Part 2.1: Strategic Jumping: Crafting Successful Memory Exploits

Strategic jumping in the context of successful memory exploits involves intentionally changing control flow instructions within the targeted program's memory space. This manipulation includes techniques such as back jumps, forward leaps, long jumps, short jumps, conditional jumps, and indirect jumps to change the program's execution flow. Attackers use these strategic jumps to exploit software vulnerabilities, enabling them to run arbitrary code and gain unauthorized access or control of the system. Gaining control over the program's execution flow allows attackers to bypass security measures, avoid detection, and execute malicious payloads in the target process's memory.

Jump (or call) a register referring to the shell code and it’s one of the forcing techniques used to execute shellcode. Using this technique, you just use a register containing the address where shellcode lives and store it in “EIP”. You try to locate the opcode for a jump to that register in one of the DLL’s (Dynamic Link Libraries) loaded when the application runs. When creating your payload, instead of overwriting EIP with a memory address, you should use the address of the jump to the register. ( Sedory, 2013)

Every strategy serves a specific function and has a major effect on the exploit's success. Common Jumping Strategies and Their Significance:

|  |  |  |
| --- | --- | --- |
| *Jumping Strategy* | *Description* | *Significance* |
| Backward Jumps | Redirect program's flow to previous instructions or memory locations.  After moving into the memory addresses, you did not find enough space to add your payload. However, in the previous addresses in the memory (have already visited them), there is valid space. Therefore, we will perform a backward jump to these memory addresses. So, we will jump to negative offset. | When there isn't enough space for a payload to be added, this strategy comes in helpful. - Allows returning previously visited memory addresses to locate suitable space for payload placement. - Allows for leaps to negative offsets, making it easier to attack. |
| Forward Jumps | Redirect program's flow to memory locations ahead. Forward jumps are like flags pointing forward, directing the program's path. They help to avoid challenges and move the program along easily, making the exploitation process faster and more difficult to detect. | Directing the programs to the next offset. Assists in overcoming barriers and moving the program forward smoothly. Speed up the exploitation process. Makes the exploitation more difficult to detect by avoiding hurdles and security checks. |
| Long Jumps | Redirect execution to distant memory addresses. You find yourself in an area of memory, but you need to access another location far away to upload your shell code. Since physically moving the shell code isn't an option, you employ a long-distance jump to reach that distant offset. This strategic jump enables you to fill the gap between your current location and the target offset, facilitating the execution of your payload. | Allows access to external memory locations, which is essential for uploading shell code. Covers the gap between the current location and the destination offset, allowing for efficient payload execution. Improves exploit efficacy by effectively accessing distant memory addresses. And will help to find more contiguous available addresses for out shell code. |
| Short Jumps | Short jumps are like small steps in memory exploitation, allowing programs to quickly jump to nearby instructions over a few bytes. Its opcode is 0xEB (EB refers to unconditional short jump). They are useful for quickly jumping to adjacent instruction addresses. They provide exact control over program flow in confined locations, which makes them essential for exploit developers despite their limited range. | Provides accurate control over program flow in restricted places, which is critical for exploit development. Allows rapid access to nearby instructions in a few bytes. Valuable for executing targeted actions within exploit payloads. |
| Conditional Jumps | The "jump if condition is met" strategy depends on analyzing specific conditions flags—like CF, OF, PF, SF, and ZF—that are kept in the EFLAGS register. If the flags satisfy the required condition, this method allows a jump to a specific instruction. Based on the current value of the EIP register, a relative offset is computed to determine the location of the target instruction. Basically, this method enables the program to modify its execution flow conditionally based on the state of the designated flags. | Allows conditional modification of the execution flow based on specific conditions flags. Increases flexibility and complexity during exploit execution. It dynamically identifies the target instruction location, boosting the effectiveness in exploiting weaknesses. |
| Indirect Jumps | Redirect execution to addresses stored in registers or memory. In simpler terms, when exploiting a payload, you can store the address of your payload in a different memory location. Then, when you're ready to execute the payload, you jump to the memory location that holds the address of your payload. | Allows for the dynamic identification of target addresses by redirecting execution to addresses stored in registers or memory. Required for creating complicated exploit chains and circumventing security measures. Allows for the execution of payloads stored in different memory locations, increasing exploit execution flexibility. |

## Part 2.2: Memory Corruption

### Part 2.2.1: Memory Corruption Techniques

|  |  |
| --- | --- |
| *Memory Corruption Technique* | *Description* |
| Buffer Overflow | This occurs when the amount of data in the buffer exceeds its storage capacity. That extra data overflows into nearby memory locations, corrupting or overwriting the data in those areas. (Azeria Labs, 2022) (Fortinet, 2024) |
| Use-After-Free (UAF) | Refers to a memory corruption bug that happens when an application attempts to use memory that is no longer assigned to it (or freed) after it has been assigned to another application. This can result in crashes and unintended data overwrites, or in cyberattack situations, it can lead to arbitrary code execution or allow an attacker to achieve remote code execution capabilities. (Anon., n.d.) |
| Integer Overflow/Underflow | Integer overflow occurs when the result of an arithmetic operation exceeds the maximum value represented by the integer's bit structure.  Integer underflow occurs when the result of an arithmetic operation is less than the minimal value represented by the integer's bit structure. This can lead to unexpected behavior, including memory corruption or execution of unintended code. (Olympix, 2023) |
| Format String Vulnerabilities | A format string vulnerability is a type of software vulnerability that can affect applications that use format strings to process user input. Format string vulnerabilities can be used by attackers to read or write arbitrary memory locations, execute arbitrary code, or cause the program to crash. The vulnerability occurs when an attacker-controlled format string is provided to a function that does not validate or sanitize it. This allows the attacker to manipulate the format string in order to gain access to or modify memory regions that were not meant to be accessed or changed. (Streichsbier, 2023) |
| Heap Overflow | A heap overflow is a memory corruption vulnerability that happens when a program writes data that exceeds the limit of a dynamically generated buffer in heap memory. This causes corruption of near heap data structures, which may allow an attacker to execute arbitrary code. Heap overflow vulnerabilities are a major security concern because they can be used to gain unauthorized access to a system, escalate privileges, or create a denial of service by crashing the application. To avoid data corruption and unauthorized code execution, heap overflow vulnerabilities must be mitigated using safe memory allocation and deallocation methods such as heap integrity checks and heap segmentation. (geeksforgeeks, 2023) |

### Part 2.2.2: Memory Corruption Mitigation

|  |  |  |
| --- | --- | --- |
| *Mitigation Strategy* | *Description* | *Bypassing Techniques* |
| Data Execution Prevention (DEP) | DEP is a security measure that stops programs from being run in memory offsets labelled as non-executable. It helps to avoid buffer overflows and other memory corruption attacks by preventing attempts to execute injected malicious code. (Roldan, 2020) | Return-Oriented Programming (ROP): Instead of executing code directly from the stack, attackers use a series of small instruction sequences already present in executable memory, ending with a RETN instruction, to perform malicious actions. These sequences, known as "gadgets," are chained together to create a payload that bypasses DEP by reusing existing code.  When exploiting a buffer overflow vulnerability in an application that is without DEP (Data Execution Prevention), we often replace the stored instruction pointer on the stack frame of the affected function with a pointer to a specific memory location containing our shellcode. This is frequently accomplished by redirecting the program's execution to a "JMP ESP" instruction, which jumps to the memory location specified by the ESP register and executes our shellcode.  However, with DEP enabled, we cannot execute instructions directly from the stack, but we can still influence the program's execution flow. Instead of returning to a "JMP ESP" instruction, we can direct the execution flow to a "RETN" instruction. The "RETN" instruction is frequently found in function epilogues, and it looks for a memory address on the stack to use as the next instruction pointer, also known as an EIP. In other words, "RETN" expects to find pointers to instructions rather than execute them directly from the stack, allowing us to manage the program's flow carefully. |
| Address Space Layout Randomization (ASLR) | Address space Layout randomization (ASLR) is a memory corruption mitigation strategy for operating systems (OSes) that gatekeepers use to protect against buffer overflow attacks by randomizing the area where framework executables are stacked into memory. (Beschokov, 2024) | The concept behind bypassing ASLR that in each time a process runs, ASLR gives it a new address. However, between executions, the offsets between a single function and the base address, as well as those between functions themselves, stay constant. This can be exploited to determine the address of a specific function during runtime.  Information Leakage: Attackers may exploit vulnerabilities that leak memory addresses, such as format string vulnerabilities or improper handling of pointers, to discover the base addresses of libraries or modules. Once they have this information, they can calculate the addresses of other functions or gadgets relative to the leaked base address, effectively bypassing ASLR. Brute Force: In some cases, attackers may repeatedly attempt to exploit the vulnerability, each time trying different addresses until they succeed.  Also, can be bypassed using Egg Hunter. |
| Visual Studio /GS Flag | The /GS (Buffer Security Check) flag improves program security by including runtime checks that detect buffer overflows. It includes a security cookie with stack variables to ensure its validity before the function returns. This compiler feature is critical for detecting and preventing buffer overflows, especially ones that affect the return address. Furthermore, /GS protects sensitive parameters provided into functions, such as pointers or references, by moving them to a secure location before local variables are assigned. (Anon., 2021) | Stack Pivoting: Attackers can sometimes bypass /GS by redirecting execution flow to a part of memory where the security cookie is not present or not checked, allowing them to overwrite critical parts of the stack. Heap Spraying: By filling the heap with known values and predictable structures, attackers can create a situation where they can control the execution flow without triggering the stack-based security checks. |
| Visual Studio Safe SEH | Structured Exception Handling (SEH) is an essential function in Windows programming that helps manage exceptions during program execution. Visual Studio SafeSEH, a security feature built into Visual Studio, improves program security by creating a table of allowed exception handlers within the program image. This table ensures that only verified handlers run during exception handling processes, preventing control-flow hijacking threats. Visual Studio SafeSEH improves the overall security posture of Visual Studio-developed Windows applications by reducing vulnerabilities such as buffer overflows. (Anon., 2022) | Use of Modules without SafeSEH: If an application loads a module (DLL) that is not compiled with SafeSEH, attackers can exploit this module to inject their malicious SEH handler.  Partial Overwrite: By partially overwriting the SEH record or targeting specific parts of the SEH chain, attackers can bypass SafeSEH checks and execute their payload. Finding Gadgets in Unsafe Modules: Attackers can use return-oriented programming (ROP) techniques to find useful instruction sequences in modules that are not protected by SafeSEH and chain these to gain control. |

## Part 2.3: Privilege Escalation

### Part 2.3.1 Privilege Escalation Definition

Privilege escalation is a cyberattack technique in which an attacker gains unauthorized access to higher rights, permissions, entitlements, or privileges beyond those assigned to an identity, account, user, or machine by exploiting security flaws, weaknesses, and vulnerabilities on the targeted system. Privilege escalation is a critical stage of the cyberattack chain that usually requires exploiting a privilege escalation vulnerability, such as a system defect, misconfiguration, or insufficient access controls. In most cases, the first successful attack effort is insufficient to achieve the necessary level of access to data. Which will help attackers then use privilege escalation to acquire greater access to networks, assets, and sensitive data. (Anon., 2023)

### Part 2.3.2 Privilege Escalation Types

* Horizontal privilege escalation:

This involves obtaining access to the permissions of another account, whether human or machine, that has identical privileges to the one am on it. This operation is known as "account takeover." Typically, this would apply to lower-level accounts, who may lack enough security. With each additional horizontal account hacked, an attacker expands their radius of access by gaining identical rights. This is a basic lateral movement. (Haber, 2024)

* Vertical privilege escalation:

This involves obtaining additional permissions or privileged access to a user, program, or other asset. This entails transitioning from a low to a high level of privileged access. To achieve vertical privilege escalation, the attacker may need to perform a number of intermediary steps to bypass or override privilege controls, exploit flaws in software, firmware, or the kernel, or obtain privileged credentials for other applications or the operating system itself. (Haber, 2024)

### Part 2.3.3 Common Types of Privilege Escalation Techniques

|  |  |
| --- | --- |
| *Technique* | *Description* |
| Social engineering | An attacker fooled a user into giving away their credentials or performing actions that grant the attacker elevated privileges. One of the most common social engineering techniques is phishing attacks. |
| Vulnerabilities and exploits | Attackers use unpatched software vulnerabilities, vulnerabilities in the system, buffer overflows, or other backdoors to acquire privilege escalation by using payloads or scripts. |
| Misconfigurations | The attacker uses misconfigured systems to escalate their privileges. Weak passwords, insecure network services, open ports, authentic failures, and other misconfigured systems to gain access or to upgrade their privileges. |
| DLL Hijacking | Attackers run malicious DLL (dynamic link library) files with elevated privileges by taking advantage of applications' unstable loading systems. A malicious DLL file can be used by an attacker to take control of an application's execution flow and increase their level of system privileges by putting it in a directory that the application's loading process searches. |
| Weak or Default Credentials | Attackers use weak or default credentials to access privileged accounts. They could use methods like password spraying, credential stuffing, or brute-force attacks to guess or get legitimate credentials and increase their level of privilege on the system. |
| Exploiting Shared Resources | Attackers use shared resources, including network shares, folders, or system files to run program or get access to private data which will help them to gain or find higher credentials. (Anon., 2023) |

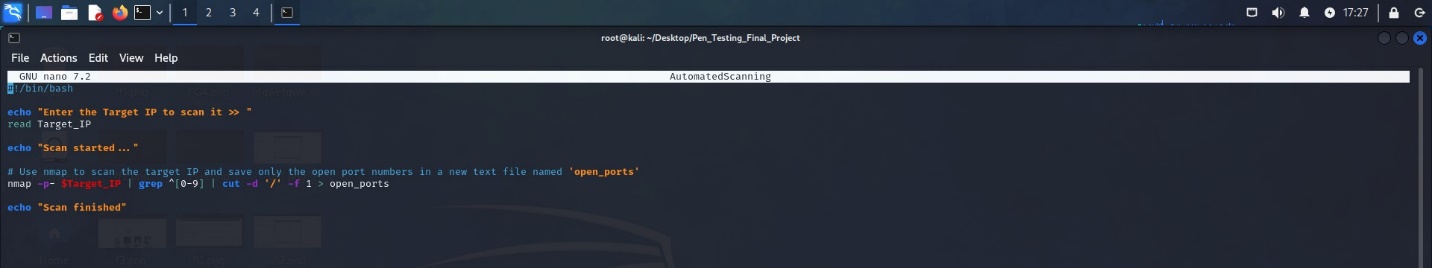
### Part 2.3.4 How to Prevent Privilege Escalation Attacks

* Use Strong password and username policies.
* Monitor and manage all privileged sessions.
* Update and patch your systems frequently.
* Enforce least privilege policy.
* Vulnerability scanners.
* Closed all the unused ports.
* UEBA (User and Entity Behavior Analytics): UEBA is a security technology that analyses user behavior and finds unusual activities using machine learning. This program may identify attempts to obtain sensitive data, alter access patterns, or increase rights. (Anon., 2023)

# Part 3: Target #1 Exploit-Me server

## Part 3.a. Automated Port Scanning Using Bash Script

### Part 3.a.a. Screenshot of the code



*Hamza AL-Risheq*

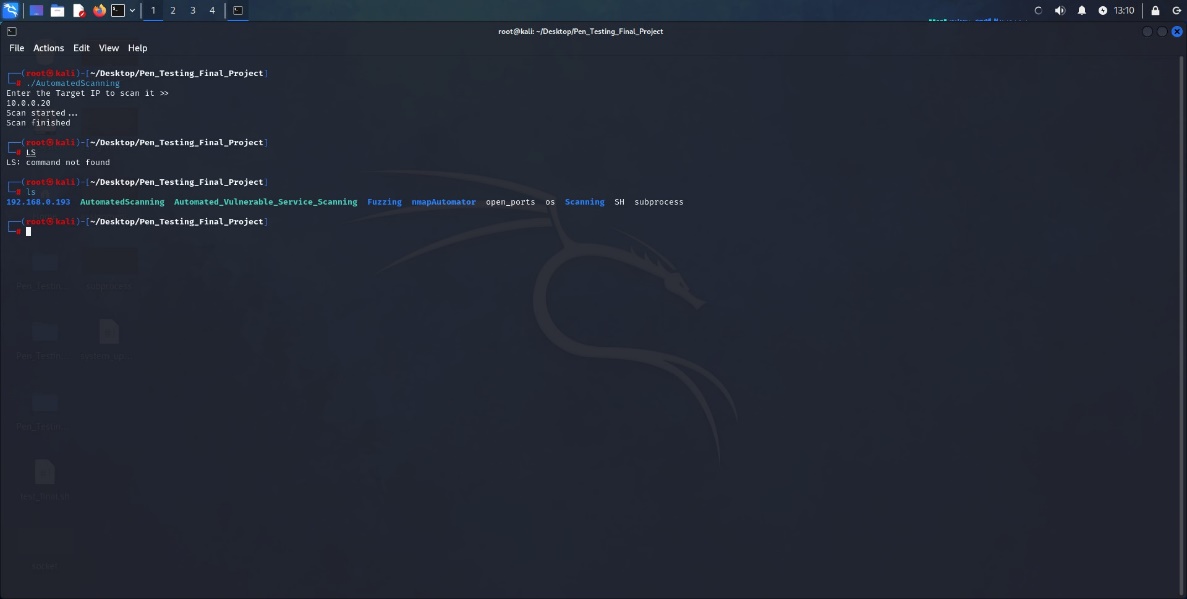
### Part 3.a.b. Description of the bash script:

This script enables the user to enter a target IP address, scans all ports on that IP address using nmap command (nmap -p- $Target\_IP | grep ^[0-9] | cut -d '/' -f 1 > open\_ports), extracts the open port numbers from the scan results, and saves them to a file called open\_ports. Finally, it alerts the user that the scan is complete.

Explain of the nmap command:

This command uses Nmap to perform a complete scan of all TCP ports on a specified target IP. It then uses the 'grep' command to sort through the Nmap output, looking for lines that begin with a digit (0-9) representing a port number. The 'cut' command then splits each line using the '/' delimiter and extracts the first field, which represents the port number. Finally, the output is saved to a file called 'open\_ports'.

### Part 3.a.c. Results of the scanning:



*Hamza AL-Risheq*

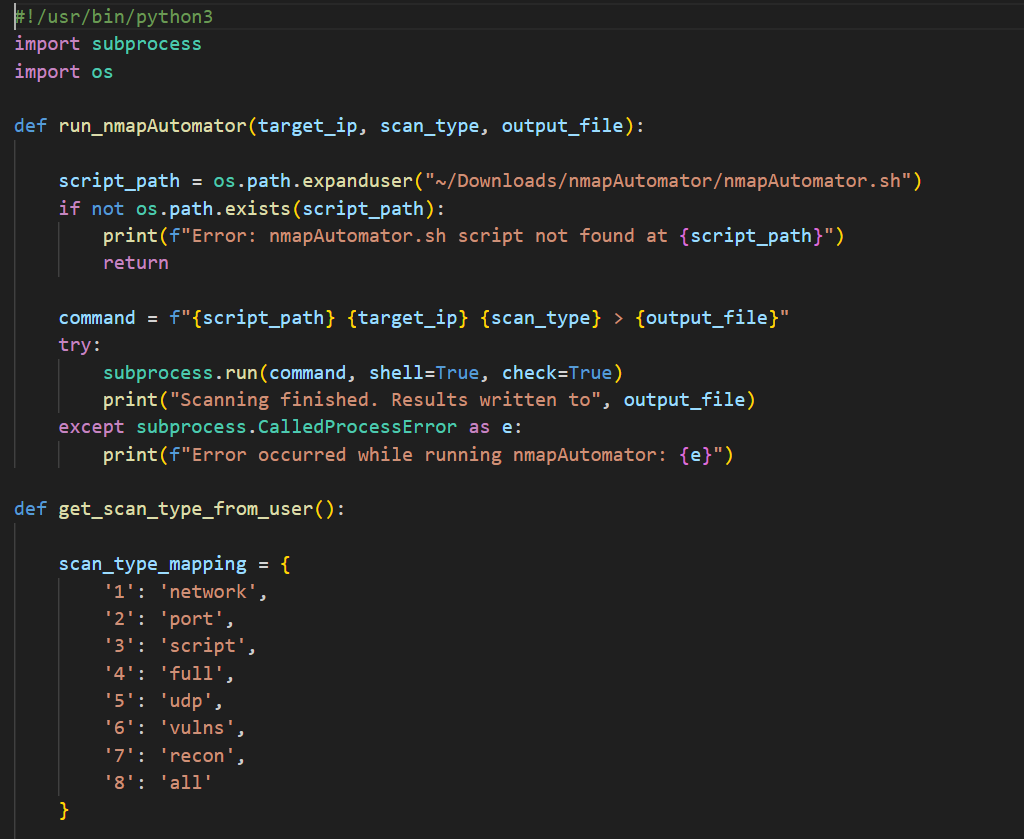
A computer screen with a dark background

Description automatically generated

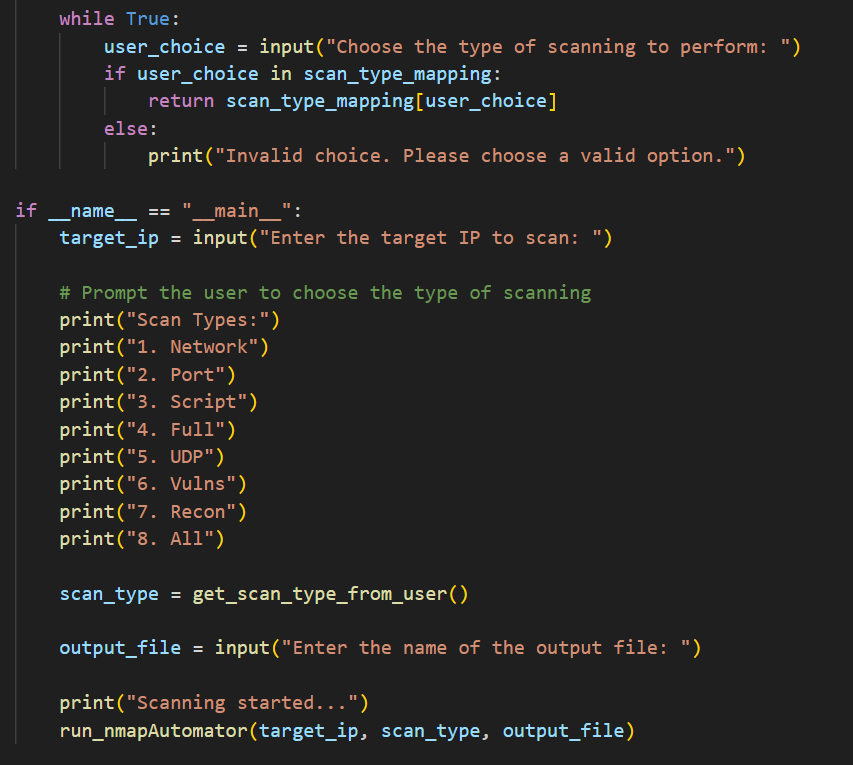
*Hamza AL-Risheq*

## Part 3.b. Efficient Network Scanning: Custom Python Scripts for Identifying Vulnerable Services

### Part 3.b.a. Screenshot of the code



*Hamza AL-Risheq*



*Hamza AL-Risheq*

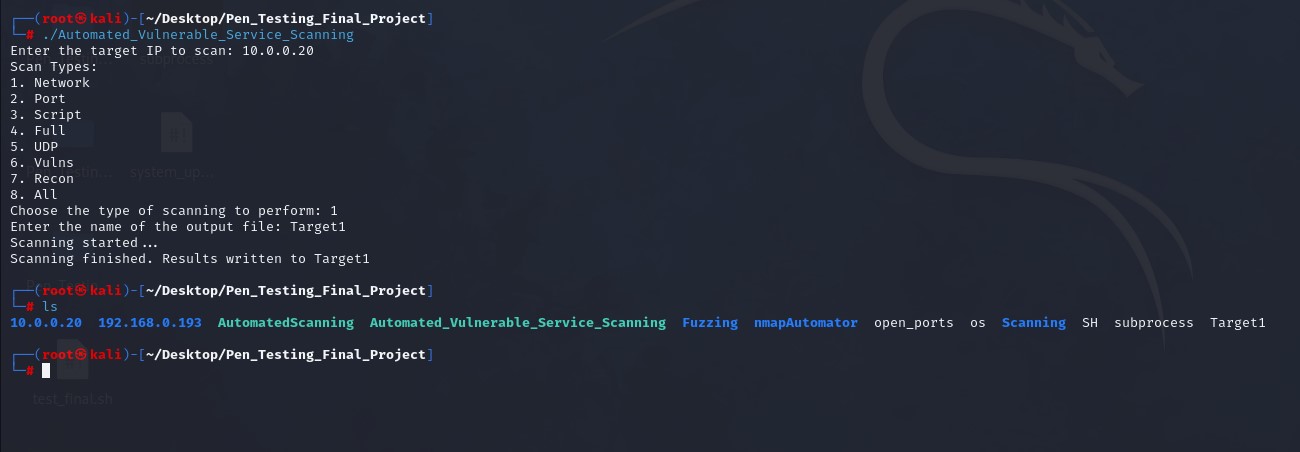
### Part 3.b.b. Description of the code

This Python script provides an interface for automating Nmap scans with the nmapAutomator.sh script. It begins by asking the user to enter a target IP address for scanning. Then it displays a choice of scan kinds, such as network, port, and script, allowing the user to select the required scan type. After selecting the scan type, the user enters the name of the output file in which the scan results will be saved.

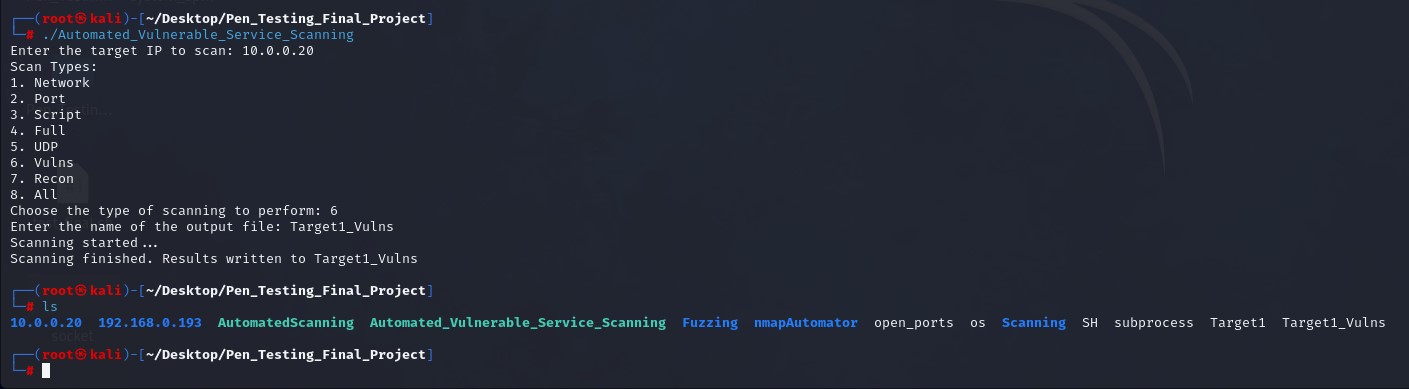
After receiving the relevant inputs, the script calls the run\_nmapAutomator function. This function generates a command string using the target IP, scan type, and output file name. It looks for the nmapAutomator.sh script and, if it finds it, uses the subprocess.run method to execute the command.

I used modules such as os and subprocess to automate Nmap scans in Python. The os library allows me to interface with the operating system by creating file paths and checking for the presence of files or folders. whereas the subprocess module allows external commands to be executed from within a Python script, allowing for smooth interaction with tools such as nmapAutomator.

### Part 3.a.c. Results of the scanning:



*Hamza AL-Risheq*



*Hamza AL-Risheq*

A screenshot of a computer

Description automatically generated

*Hamza AL-Risheq*

## Part 3.c. Utilizing Fuzzers and Debuggers for Memory Analysis and Software Crash Detection

In this part, I used fuzzers as Spike Fuzzer and debuggers as Immunity Debugger to analyze software in memory. By fuzzing the software with unexpected inputs, to find flaws and probable crash locations.

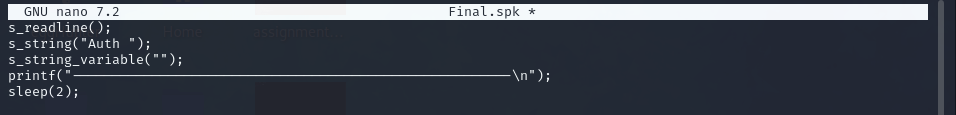
I will use fuzzers such as Spike Fuzzer, which is well-known for its ability to create diverse and faulty inputs, to repeatedly assault the target software with unexpected data, and unexpected data size. This approach seeks to cause unexpected behavior, potentially uncovering hidden weaknesses that attackers can use.

At the same time, I will use strong debugging tools such as Immunity Debugger to closely watch the execution of the software being tested. By establishing breakpoints, inspecting memory states, and tracing code execution pathways, will get me crucial insights into the software's internal operations and identify possible vulnerabilities.

The steps of this process:

* Set Up Fuzzing Environment:

Select a suitable fuzzer tool, such as Spike Fuzzer or AFL (American Fuzzy Lop), that is compatible with the nature of the software and the inputs that accepts. And configure the fuzzing environment by creating input seed files.

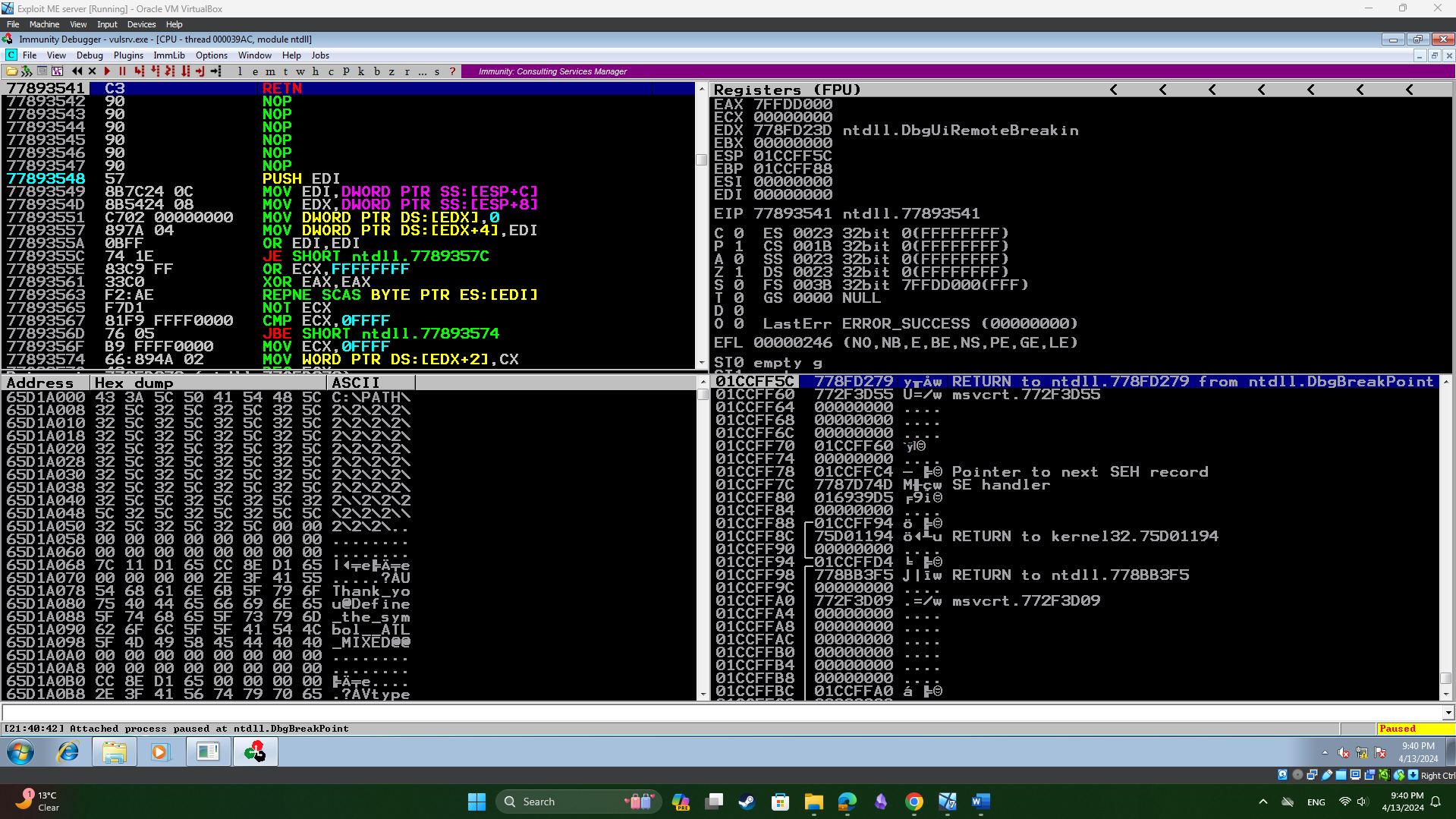


*Hamza AL-Risheq*

* Monitor the target software:

To effectively conduct fuzzing, it's crucial to first download and configure tools like Immunity Debugger. These tools enable us to monitor and analyze the behavior of the target software.

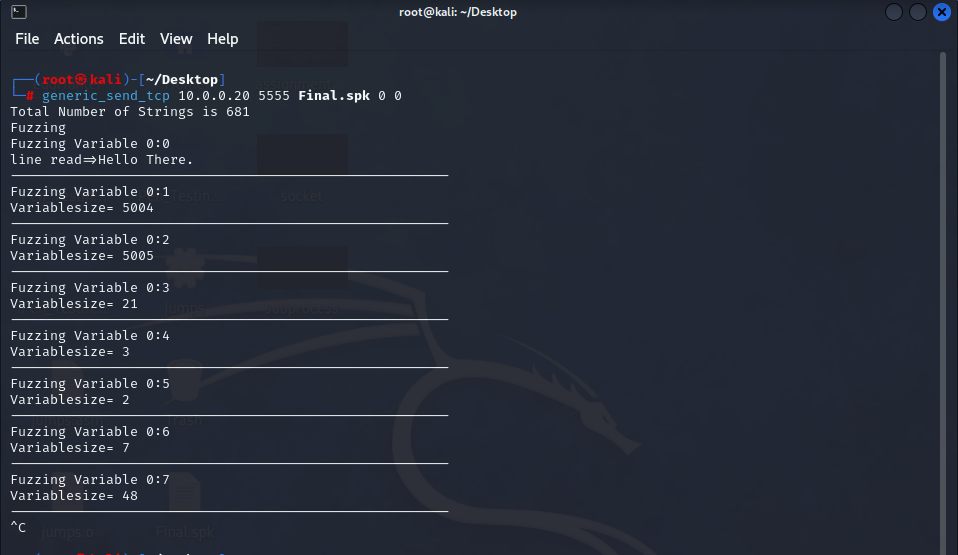
I will open the software and attach the software in the immunity debugger.



*Hamza AL-Risheq*

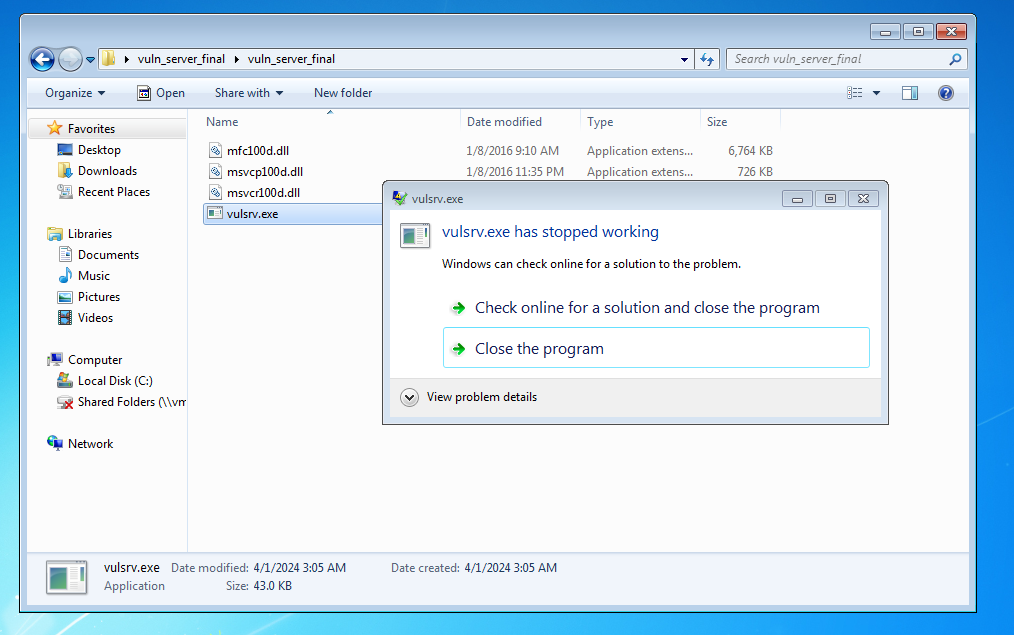
* Perform Fuzzing to the target:

I'll use my Kali machine to conduct fuzzing on the software. First, I'll establish a connection to the target machine, followed by connecting to the software port. Once connected, I'll initiate the fuzzing process on the program.



*Hamza AL-Risheq*

The result of the fuzzing process in this step:

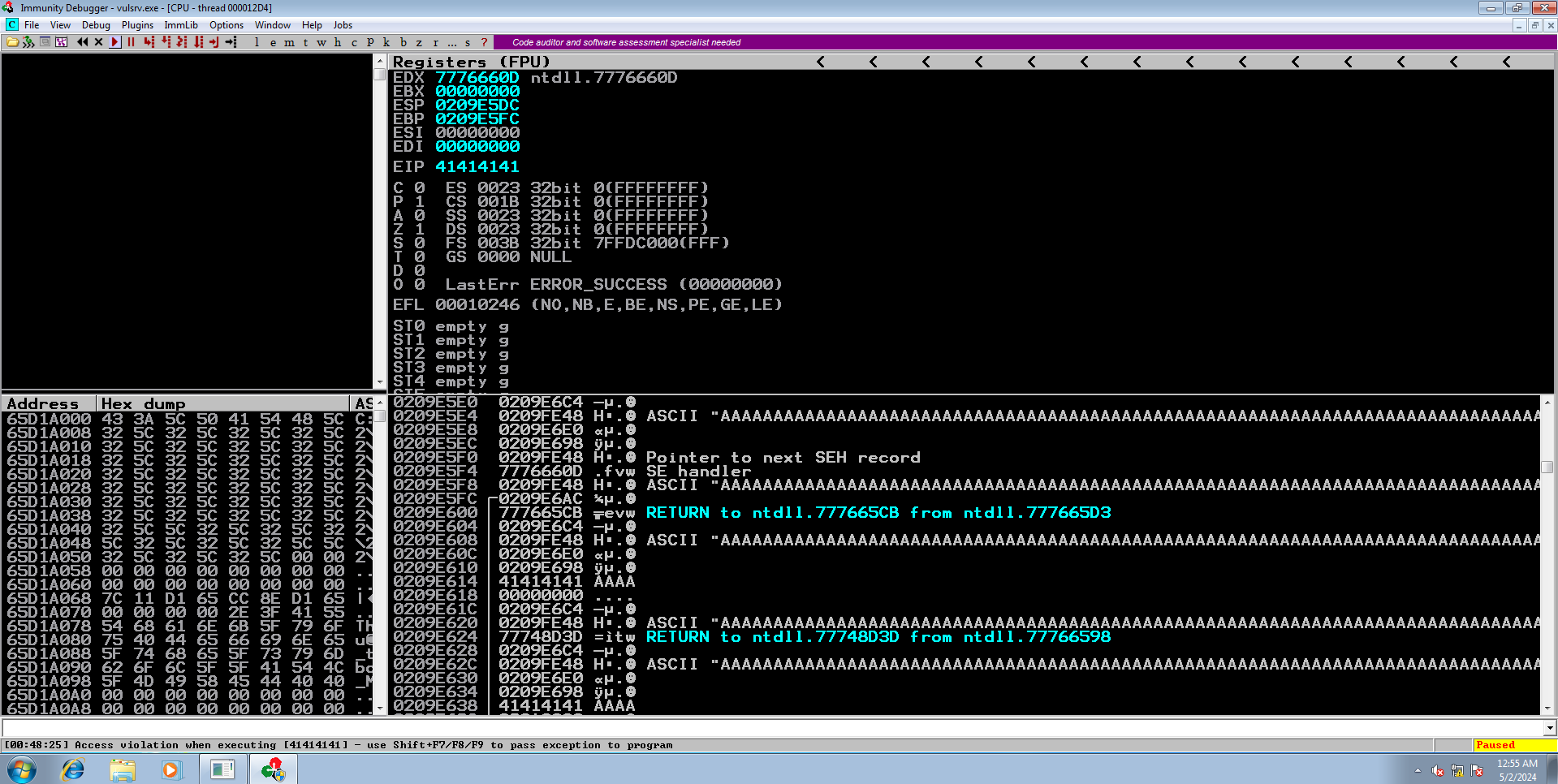


*Hamza AL-Risheq*

Based on the image, it appears that our program "vulsrv" has crashed, prompting the operating system to request its closure due to a crash.

This crash indicates that during one of the fuzzing attempts, the program failed to handle a particular payload, leading to its crash. Now, our task is to identify which fuzz payload caused the program to crash.

* Screenshots that the program crashed using immunity debugger:



*Hamza AL-Risheq*

From the images, it's evident that the entire memory stack is flooded with 'A's, causing a crash. This scenario is critical since it indicates a possible vulnerability. Furthermore, one of our system's most important registers, EIP (Instruction Pointer that holds the memory address of the next instruction to be executed by the processor), is overwritten with 'A's. This suggests that I may take control of vital registers such as EIP and change memory, posing a significant security risk. Such an event shows the possibility of exploiting buffer overflow vulnerabilities, emphasizing the significance of strong input validation and memory management in program design.

## Part 3.d. Creating Short and Back Long Jumps with NASM Tool

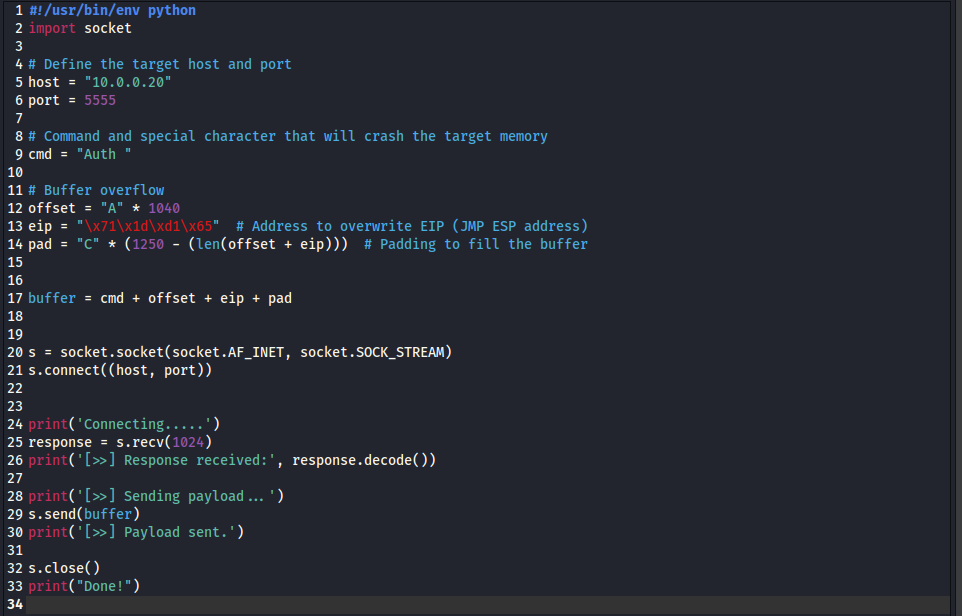
A black surface with white spots

Description automatically generated with medium confidence

*Hamza AL-Risheq*

* jmp long -4000: This command leads to perform a backward long jump to a location in memory that is 4000 bytes before the current instruction pointer. It's encoded as E9 followed by the relative address in little-endian format.
* jmp short 10: This command leads to perform a short jump, which means it can only jump within a limited range, typically -128 to +127 bytes from the current instruction pointer. It jumps forward 10 bytes from the current instruction. It's encoded as EB followed by the relative address in little-endian format.

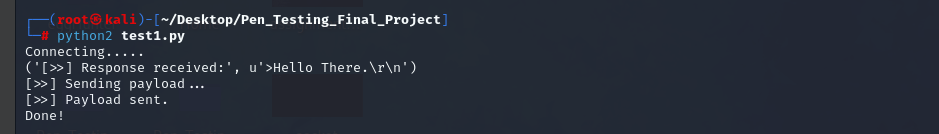
## Part 3.e. Python Script to Do Buffer Overflow Attack Using JMP ESP Technique



*Hamza AL-Risheq*

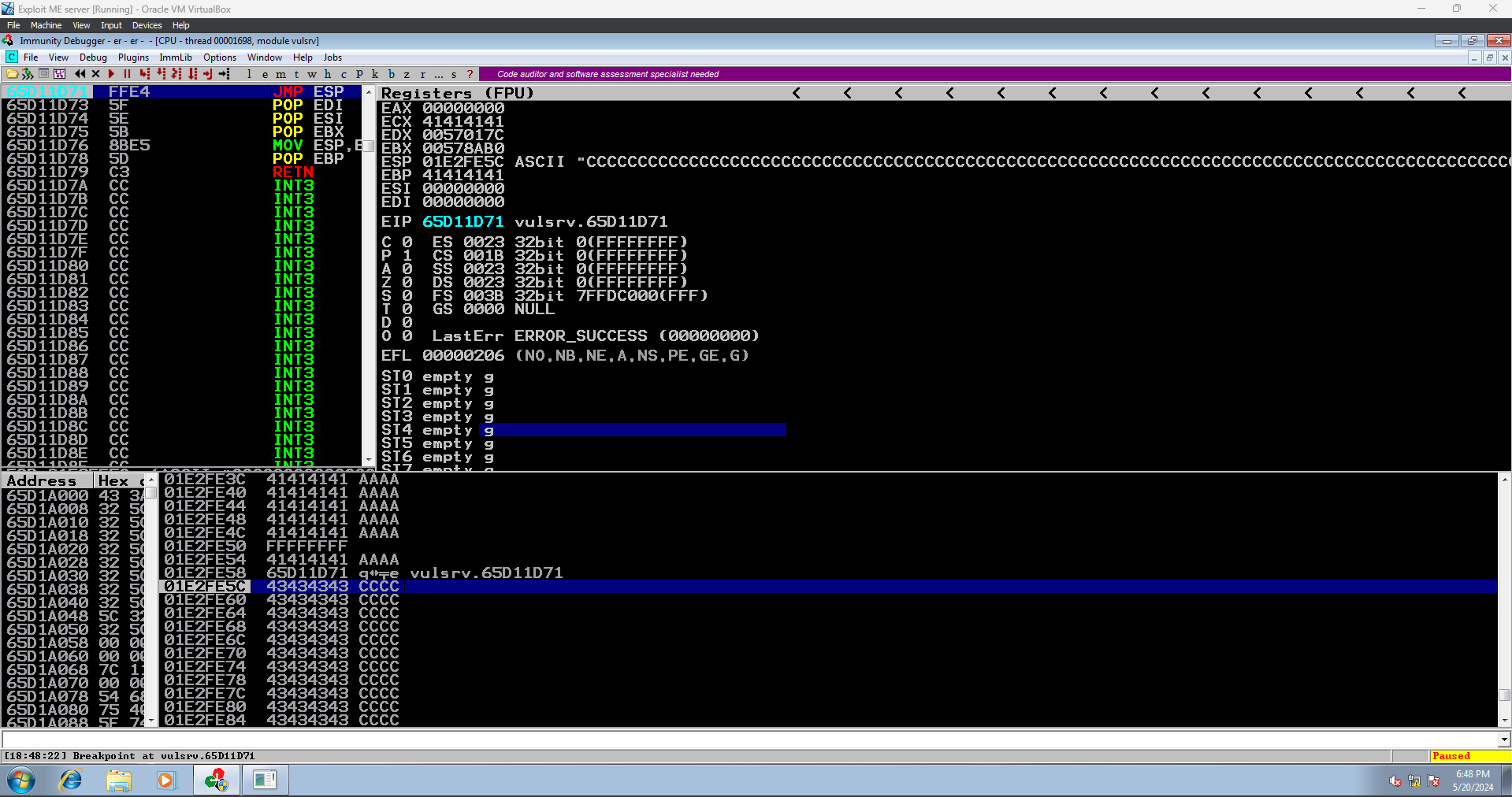
This Python script is designed to exploit a vulnerable service running on a specified host and port using a buffer overflow attack with the JMP ESP technique. The script defines the target IP address and port, which are crucial for connecting to the service. It then constructs a payload starting with a command string ("Auth ") that specifies special functions and the space after the special character that will let us expand the buffer for the function (the character that will give us the ability to overwrite the memory), followed by a series of 'A' characters (offset) to overflow the buffer and reach the Extended Instruction Pointer (EIP). The script overwrites the EIP with an address pointing to a JMP ESP instruction, which redirects execution to the subsequent padding. This padding ensures the buffer reaches the necessary length to exploit the vulnerability effectively.

After connecting to the server, it receives a first answer from the server, which it prints for verification. The script then sends the created payload to the server, to cause a buffer overflow and take control of the program's execution flow. This organized approach demonstrates how buffer overflow vulnerabilities can be used to control program execution, emphasizing the necessity of secure coding practices in preventing such attacks.



*Hamza AL-Risheq*

The script will be sent to server using python2.

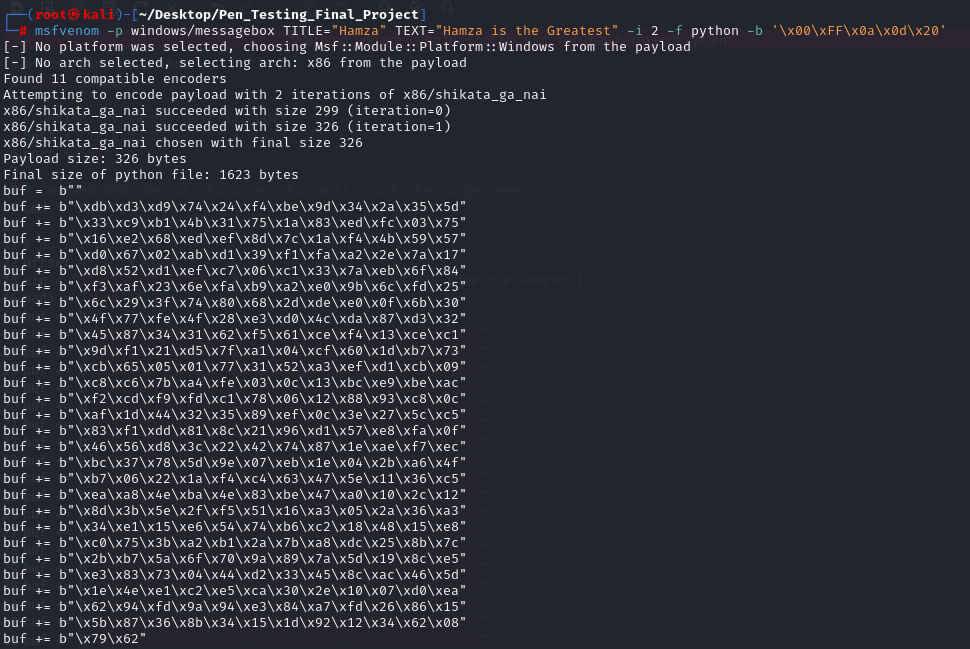


*Hamza AL-Risheq*

This is the result of running the code; we can see that we gained control of the EIP and reached the JMP ESP instruction.

## Part 3.f. Exploiting Target Systems with Message Box and Shellcode Payloads

### Part 3.f.a. Message Box

Creating message box using msfvenom.

*Hamza AL-Risheq*

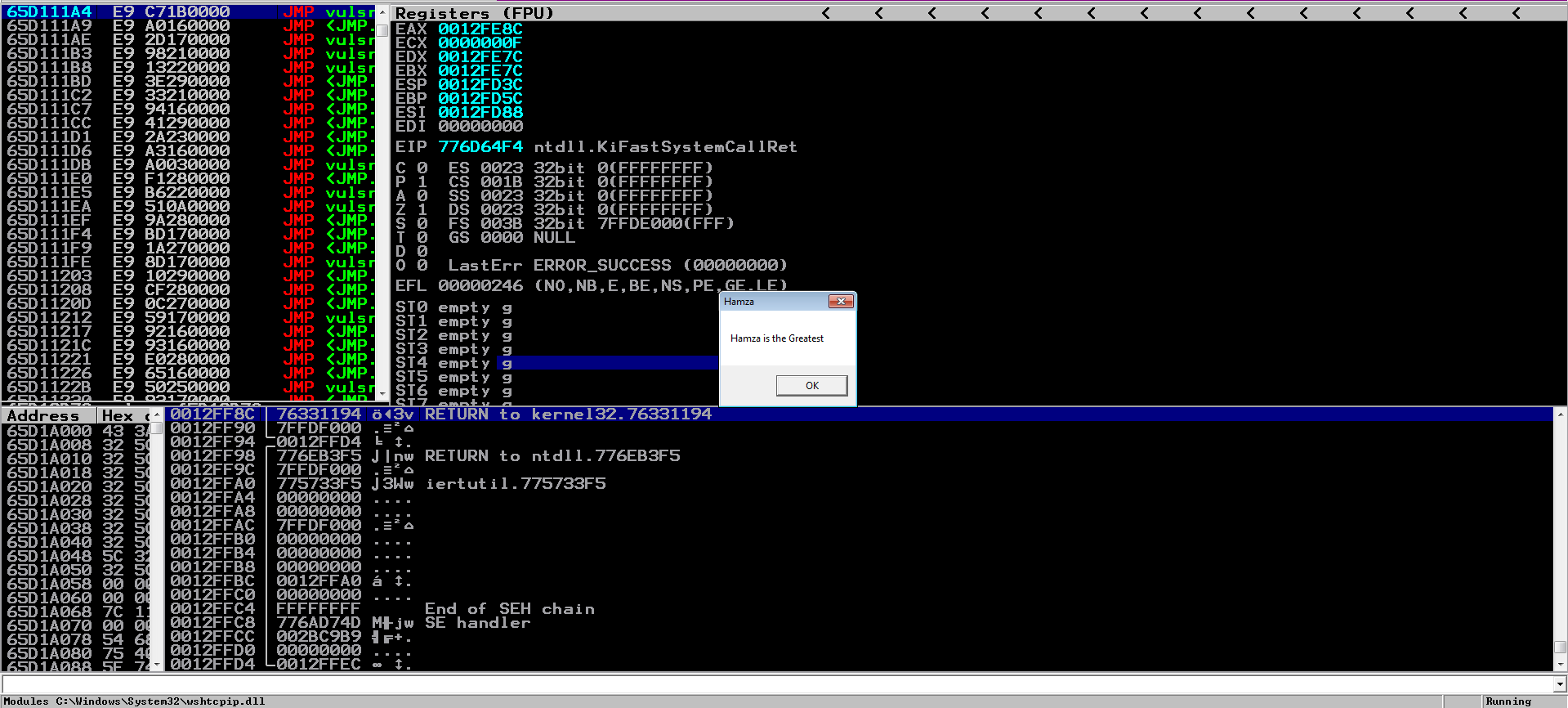
Adding the message box code to the python script to send it with the buffer.

A screenshot of a computer

Description automatically generated

*Hamza AL-Risheq*

The result of sending the message box code.



*Hamza AL-Risheq*

### Part 3.f.b. Shellcode

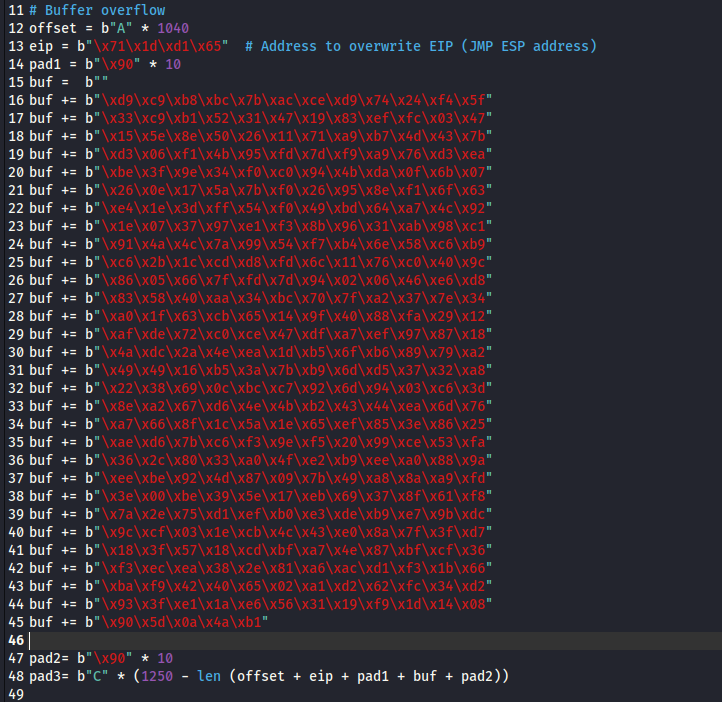
Creating shellcode using msfvenom. This shellcode must open the port number 8888.

A screen shot of a computer

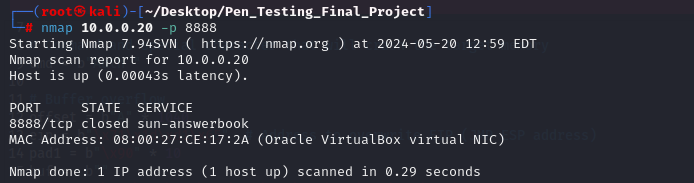
Description automatically generated

*Hamza AL-Risheq*

Adding the shellcode to the script.

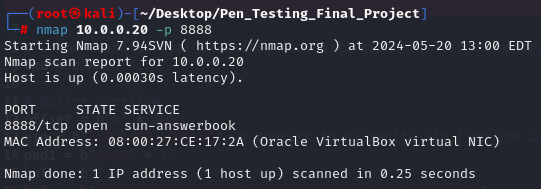


*Hamza AL-Risheq*

Checking port 8888 before sending the buffer using namp.

*Hamza AL-Risheq*

Checking port 8888 after sending the buffer using namp.



*Hamza AL-Risheq*

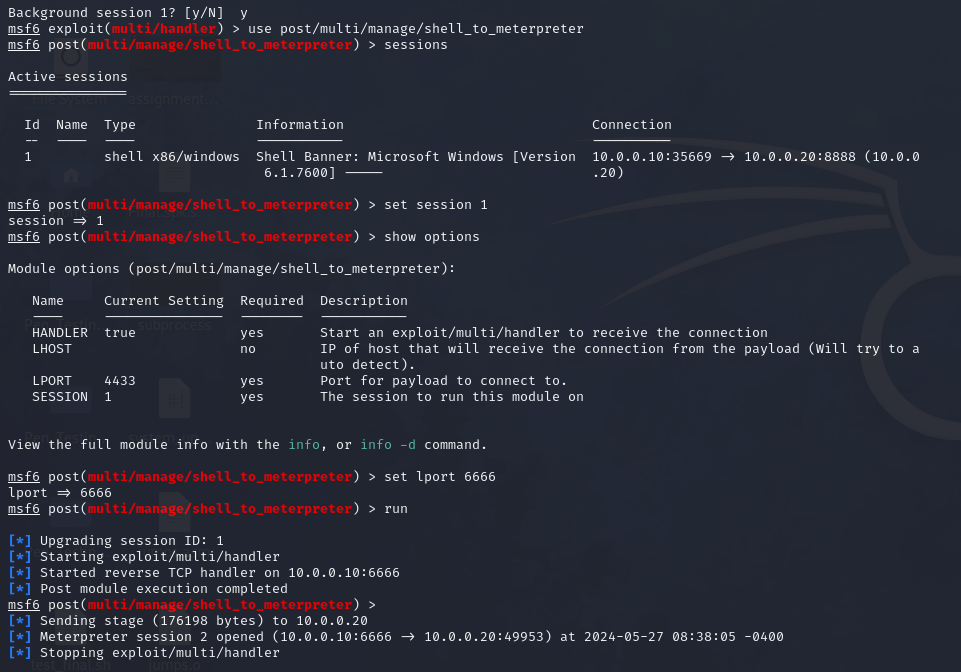
### Part 3.g. Setting Up and Exploiting with Metasploit Multi/Handler



*Hamza AL-Risheq*

# Part 4: Target #2 AnimeBlast server

## Part 4.a: Scanning the Target for Open Ports Using Gateway Session



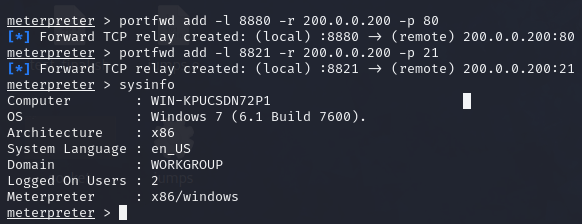
*Hamza AL-Risheq*

A screenshot of a computer program

Description automatically generated

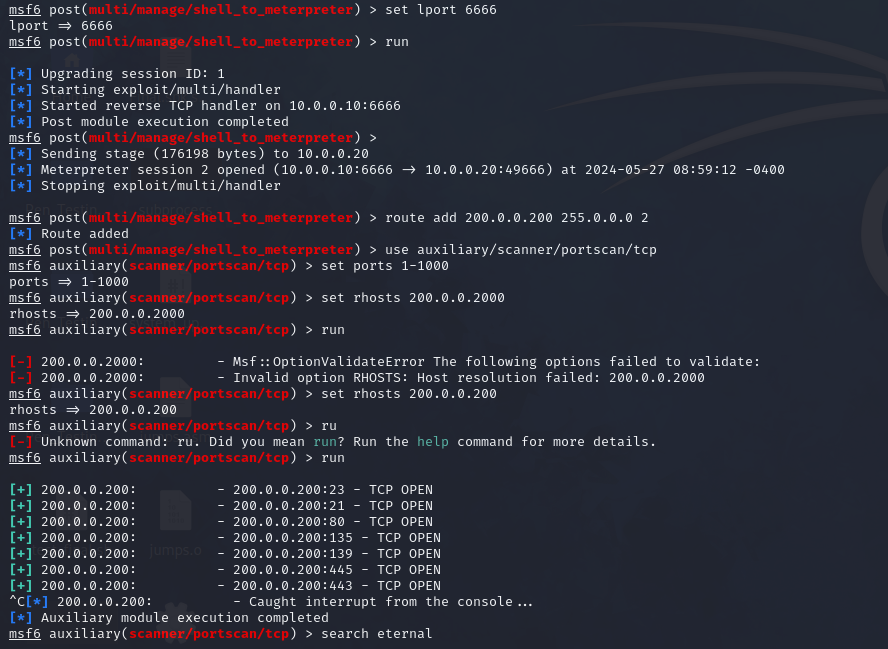
*Hamza AL-Risheq*

## Part 4.b: Creating Tunnels and Pivoting to Attack the AnimeBlast Server



*Hamza AL-Risheq*

## Part 4.c: Identify the vulnerable services using Metasploit auxiliary



*Hamza AL-Risheq*

## Part 4.d: Identifying and Exploiting a SQL Injection Vulnerability

A screenshot of a cartoon

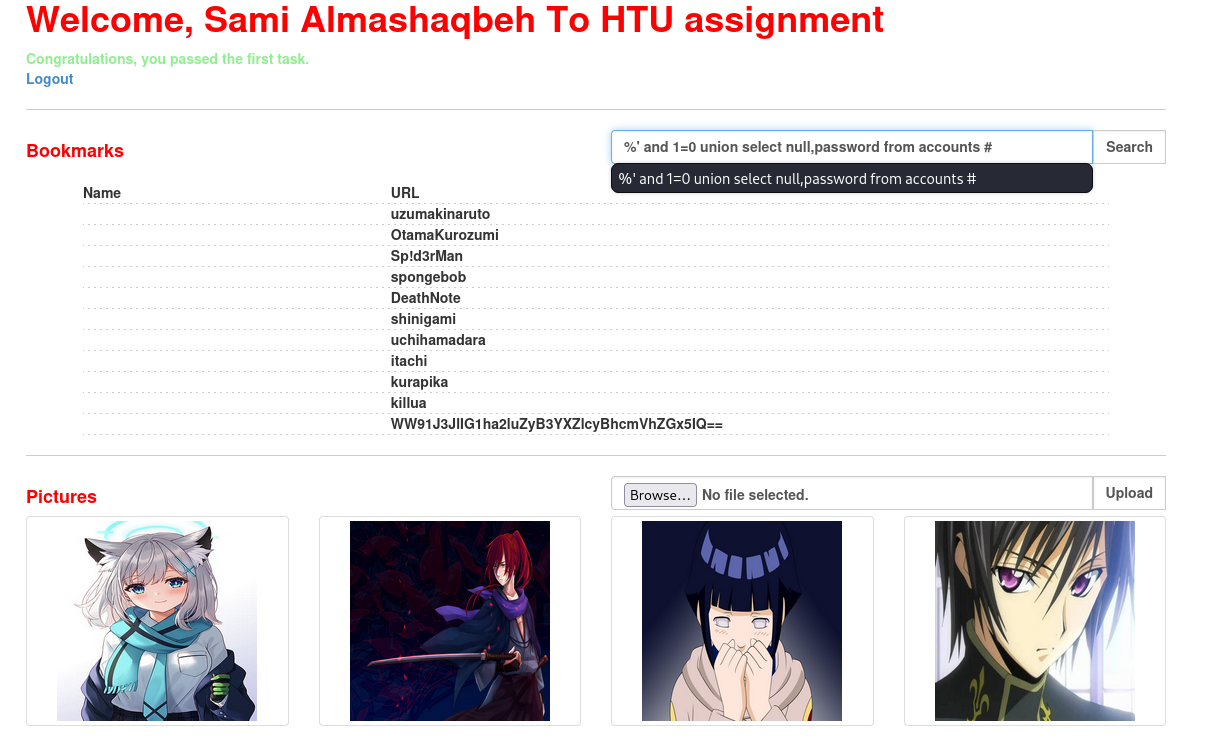
Description automatically generated

*Hamza AL-Risheq*

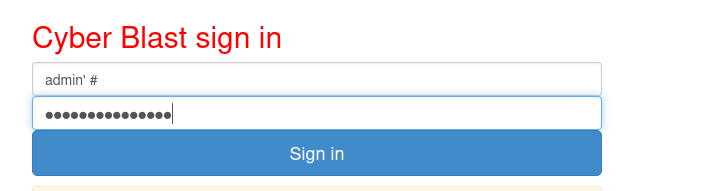
A screenshot of a computer

Description automatically generated

*Hamza AL-Risheq*



*Hamza AL-Risheq*



*Hamza AL-Risheq*

## Part 4.e: Demonstrating a Cross-Site Scripting (XSS) Vulnerability and Its Potential Exploits



A screenshot of a computer

Description automatically generated

*Hamza AL-Risheq*

An attacker (or penetration tester) can leverage an XSS vulnerability in various ways:

By injecting a script that sends the user's cookies to the attacker's server, an attacker can hijack the user's session, gaining unauthorized access to their account.

Why Cookies Are Valuable:

* Session Hijacking: Cookies often contain session tokens that keep a user logged into a website. By stealing these cookies, the attacker can impersonate the user, gaining full access to their account without needing to know their password.
* Personal Information: Cookies may store personal information such as usernames, email addresses, or other sensitive data, which the attacker can use for further exploitation.

Scenario of attacker (or penetration tester) can leverage an XSS:

* An attacker posts a comment on a vulnerable forum with the malicious script.
* Another user views the comment.
* The script runs in the background and sends the user’s cookies to the attacker.
* The attacker uses the stolen cookies to log in as the user, accessing their account and potentially performing malicious actions like changing settings, making purchases, or stealing more personal information.

## Part 4.f: Finding and Exploiting a Remote Code Execution (RCE) Vulnerability



*Hamza AL-Risheq*

A screenshot of a computer

Description automatically generated



A close-up of a website

Description automatically generated

*Hamza AL-Risheq*

## Part 4.g: Brute Forcing FTP Credentials: Discovering Usernames and Passwords



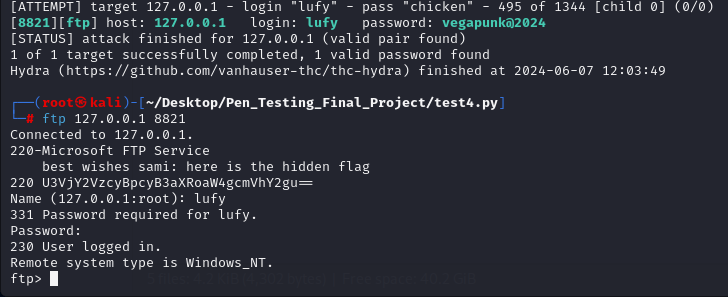
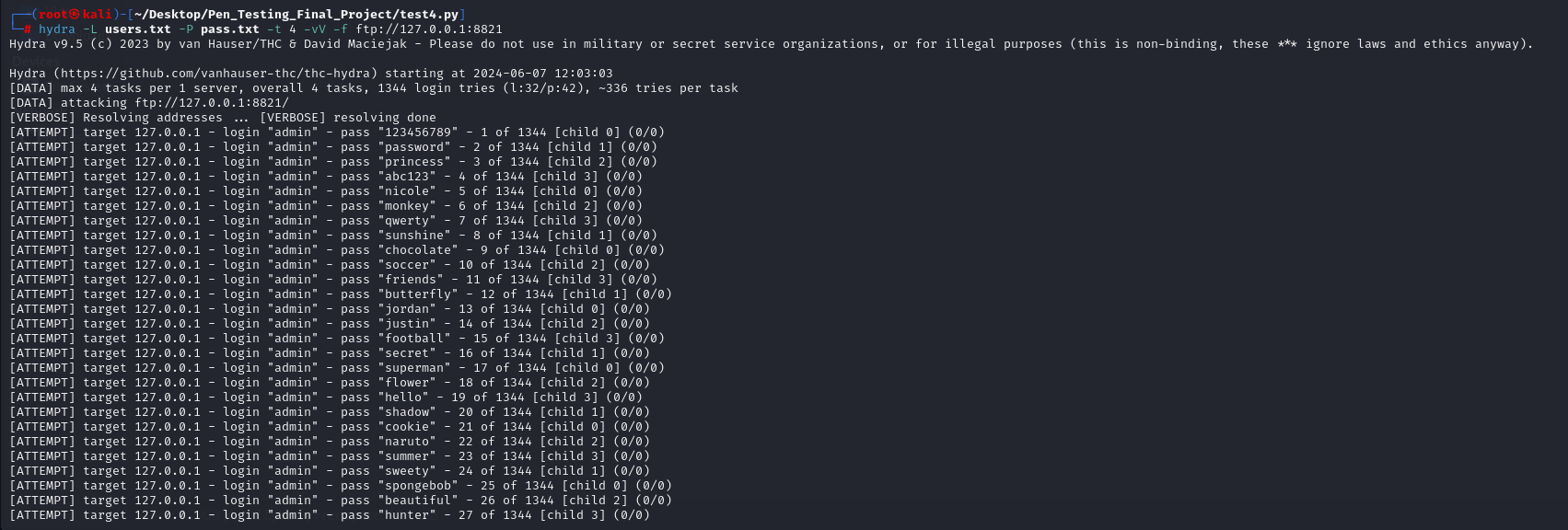
*Hamza AL-Risheq*

A screen shot of a computer program

Description automatically generated

*Hamza AL-Risheq*

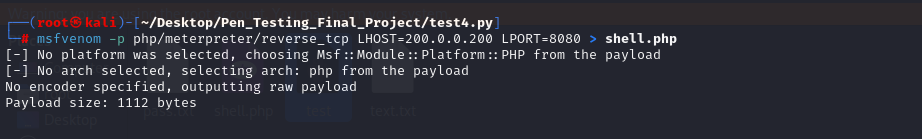
Using the scanner auxiliary from Metasploit did not work normally due to problems in the connection, so after doing the port forwarding, I used the Hydra tool to brute force the username and password, as we can see in the following pictures.



*Hamza AL-Risheq*

*Hamza AL-Risheq*

## Part 4.h: Exploiting a File Upload Vulnerability to Deploy a PHP Shell

A screenshot of a computer

Description automatically generated

*Hamza AL-Risheq*

*Hamza AL-Risheq*

## Part 4.i: CTF Challenge: Discover and Decode the Hidden Flags

|  |  |
| --- | --- |
| ***Flag*** | ***Flag After Decoded*** |
| U3VjY2VzcyBpcyB3aXRoaW4gcmVhY2gu== | Success is within reach. |
| Q29uZmlkZW5jZSBzb2FyaW5nIHNreS1oaWdolQ== | Confidence soaring sky-high |
| SW5maW5pdGUgcG90ZW50aWFsIGF3YXl0cyB5b3Uu== | Infinite potential awayts you. |
| WW91J3JlIG1ha2luZyB3YXZlcyBhcmVhZGx5IQ== | You're making waves areadly! |
| WW91J3Z1IGV4Y2VsZWQgYmV5b25kIGV4cGVjdGF0aW9ucyE= | You'vu exceled beyond expectations! |
| WW91ciB1ZmZvcnQgdHJ1bHkgc2hpbmVzIQ== | Your uffort truly shines! |
| QnJpZ2h0IGZ1dHVyZSBsaWVZIGFkaGVhZC4== | Bright future lieY adhead. |
| NC4pIFRoZSBwb29sIG9uIHRoZSByb29mIG11c3QgaGF2ZSBhIGX1YWsu # | 4.) The pool on the roof must have a eak. |
| SW5zcGlyaW5nIHByb2dyZXNZIGV2ZXJ5IGRheS4= | Inspiring progresY every day. |
| 54fe0353d03a0b0c5780c70446fd4027f27a27eb7438c60350bd1df77878f17078cbd23132b7b b40 | Endless possibilities awayt exploration. |

**For the last flag the password is (naruto).**

# References

Sedory, D., 2013. *Using SHORT (Two-byte) Relative Jump Instructions.* [Online]   
Available at: https://thestarman.narod.ru/asm/2bytejumps.htm  
[Accessed 6 apirl 2024].

Anon., 2021. */GS (Buffer Security Check).* [Online]   
Available at: https://learn.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check?view=msvc-170  
[Accessed 6 april 2024].

Anon., 2022. */SAFESEH (Image has Safe Exception Handlers).* [Online]   
Available at: https://learn.microsoft.com/en-us/cpp/build/reference/safeseh-image-has-safe-exception-handlers?view=msvc-170  
[Accessed 6 april 2024].

Anon., 2023. *What are Privilege Escalations? Attacks, Understanding its Types & Mitigating Them.* [Online]   
Available at: https://www.eccouncil.org/cybersecurity-exchange/penetration-testing/privilege-escalations-attacks/  
[Accessed 6 april 2024].

Anon., n.d. *Use After Free (UAF).* [Online]   
Available at: https://www.firewalls.com/blog/security-terms/use-free-uaf/  
[Accessed 6 april 2024].

Azeria Labs, 2022. *PROCESS MEMORY AND MEMORY CORRUPTIONS.* [Online]   
Available at: https://azeria-labs.com/process-memory-and-memory-corruption/  
[Accessed 6 april 2024].

Beschokov, M., 2024. *What Is ASLR (Address Space Layout Randomization)?.* [Online]   
Available at: https://www.wallarm.com/what/what-is-aslr-address-space-layout-randomization  
[Accessed 6 april 2024].

Fortinet, 2024. *Buffer Overflow.* [Online]   
Available at: https://www.fortinet.com/resources/cyberglossary/buffer-overflow#:~:text=Also%20known%20as%20a%20buffer,the%20data%20in%20those%20locations.  
[Accessed 6 april 2024].

geeksforgeeks, 2023. *Heap overflow and Stack overflow.* [Online]   
Available at: https://www.geeksforgeeks.org/heap-overflow-stack-overflow/  
[Accessed 6 april 2024].

Haber, M., 2024. *Privilege Escalation Attack & Defense Explained.* [Online]   
Available at: https://www.beyondtrust.com/blog/entry/privilege-escalation-attack-defense-explained#:~:text=A%20privilege%20escalation%20attack%20is,actor%20or%20an%20insider%20threat.  
[Accessed 19 June 2023].

Olympix, 2023. *Integer Overflows and Underflows in <0.8 Solidity Smart Contracts.* [Online]   
Available at: https://www.linkedin.com/pulse/integer-overflows-underflows-08-solidity-smart-contracts-olympix/  
[Accessed 6 april 2024].

Roldan, A., 2020. *Bypassing DEP with ROP.* [Online]   
Available at: https://fluidattacks.com/blog/bypassing-dep/  
[Accessed 6 april 2024].

Streichsbier, S., 2023. *Format String.* [Online]   
Available at: https://docs.guardrails.io/docs/vulnerability-classes/insecure-processing-of-data/format-string#:~:text=A%20format%20string%20vulnerability%20is,cause%20the%20program%20to%20crash.  
[Accessed 6 april 2024].

### 