IDATT2503 - Exercise 03 - CTF "challenge"

I chose to create a simple buffer-overflow challenge. My main motivation is to improve my understanding of how buffer overflows work. While this challenge isn't particularly advanced or innovative, it's an honest reflection of where I am right now. The simplicity is intentional and aimed more at my own learning process rather than being useful to anyone else.

Challenge Description:

This challenge focuses on exploiting a buffer overflow vulnerability in a C program. The challenge involves triggering the overflow and redirecting the flow of execution to a hidden function that reveals the flag.

Vulnerable Function: The C program uses the unsafe `gets()` function, which does not perform bounds checking on the input buffer.

Hidden Function: A function called `platypus()` contains the flag. This function is never called in the normal execution flow of the program.

Objective: Perform a buffer overflow to overwrite the return address of the stack and redirect execution to the `platypus()` function.

Architecture: The program is compiled for a 64-bit system. Addresses and buffers in 64-bit need to be checked.

The program is compiled without stack protection, Position-Independent Executable (PIE) and address randomization (ASLR) , and with executable stack:

```
gcc -o main-ctf main.c -fno-stack-protector -z execstack -no-pie
```

How To:

1. **Identify the Vulnerability**:
   - The program uses `gets()` to read user input, which makes it vulnerable to buffer overflows because `gets()` doesn't limit the size of the input.
   - The goal is to exploit this by overflowing the buffer to control the return address.
2. **Determine Buffer Size**:
   - By using a debugger like GNU Debugger (GDB) or similar to find addresses, and crafting payloads to target the inconsicious function, a player can determine the number of bytes needed to overflow the buffer and reach the saved return address on the stack.
   - When the program crashes, they can check the `rip` register in GDB to see which part of the payload that overwrote the return address. This gives the exact number of bytes required to overflow the buffer.

3. **Find the Address of the Hidden Function**:
    - The `platypus()` function's address can be found by decompiling the binary or using GDB’s `info functions` command. Let’s assume it’s `0x0000000000401176`.
    - The address needs to be encoded in little-endian format for the 64-bit system.
4. **Craft the Payload**:
    - Once the offset to the return address is known, players can craft the payload:Buffer:
        - Fill with `A` characters (e.g., 56 `A`s).
        - Overwrite return address: Use the little-endian encoded address of `platypus()`
    - Example payload:

      ```
      python3 -c "from pwn import *; sys.stdout.buffer.write(b'A' * 56 +
      p64(0x0000000000401176))" | ./main-ctf
      ```

Hints for the Players:

- These ol' C-programs gets out of control if you let it.

Other Details:

- **Difficulty**: Easy (basic understanding of stack-based buffer overflows, familiarity with GDB and `pwntools`).
- **Tools**: GDB for debugging and `pwntools` for crafting the exploit.
- **Flag**: The flag is printed when the `platypus()` function is called.
- **Protections**: None.

Setup:

- Either run the the compiled binary, or compile from source file

Compile:

```
gcc -o main-ctf main.c -fno-stack-protector -z execstack -no-pie
```