**Buffer Overflow and Mitigation Measures Report**

**Introduction to Buffer Overflow**

During my experiment on a Kali Linux system, I analyzed a common vulnerability in C programs known as "Buffer Overflow" (BOF). This occurs when data written to a buffer exceeds its allocated memory capacity. The overflow can overwrite critical memory values, leading to unexpected program behavior, crashes, and potential security vulnerabilities.

**Procedure**

To demonstrate a Buffer Overflow, I created a simple C program that asked users to enter a name. The program used a 10-character buffer. When I input a name exceeding this limit (specifically 30 characters), the program crashed with a "segmentation fault," indicating unauthorized memory access.Immagine che contiene testo, schermata, Carattere, Software multimediale

Descrizione generata automaticamente

**Steps Taken:**

1. Opened terminal in Kali Linux
2. Navigated to desktop directory:

cd /home/Kali/Desktop

1. Created source file using nano:

nano BOF.c

Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente

1. Inserted the vulnerable code
2. Saved and exited nano using:
   1. Ctrl+X to exit
   2. Y + Enter to save
3. Compiled the program:

gcc -g BOF.c -o BOF

1. Executed the program:

./BOF

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**Results**

* **5-character input:** Program worked normally
* **30-character input:** Program crashed with segmentation fault

This demonstrated the Buffer Overflow vulnerability, which could potentially allow arbitrary code execution - a serious security threat.

**Modification and Mitigation**

To fix this, I:

1. Increased buffer size from 10 to 30 characters
2. Recompiled the program

After modification, the program handled 30-character inputs without crashing.

Immagine che contiene testo, schermata, Software multimediale, Carattere

Descrizione generata automaticamente

**Discussion and Conclusion**

Buffer Overflow remains a dangerous programming flaw, especially in memory-unsafe languages like C. Key prevention measures include:

* Using safe string functions that check buffer sizes
* Avoiding unsafe functions like gets() or unlimited scanf()
* Preferring high-level languages with automatic memory management

This exercise highlighted the importance of secure coding practices to prevent memory-related vulnerabilities.