

ESERCIZIO W17D4

L'esercizio ci porta a comprendere meglio come funziona una vulnerabilità del tipo buffer overflow. Procediamo col creare un file di testo per il nostro programma in C. Scriviamo e lanciamo `sudo nano BOF.c`.

Dichiariamo un array che andremo a riempire con degli input di tipo stringa.

```
#!/usr/bin/perl
#include <stdio.h>

int main(){
    char buffer[10];

    printf ("Si prega di inserire il nome utente:");
    scanf ("%s", buffer);

    printf ("nome utente inserito: %s\n", buffer);

    return 0;
}
```

Buffer code

Una volta salvato il file, usciamo da quest'ultimo e passiamo alla fase di compilazione. Creiamo un file compilato chiamato BOF, da avviare per mezzo del comando `./BOF`.

```
(kali@kali)-[~/C]
$ sudo nano BOF.c
[sudo] password for kali:
(kali@kali)-[~/C]
$ gcc -g BOF.c -o BOF
```

Nano e Compilatore

Il programma sembra rispondere bene. Abbiamo provato con un numero adeguato di input. Adesso, vediamo cosa succede se esageriamo.

```
(kali@kali)-[~/C]
$ ./BOF
Si prega di inserire il nome utente:qwertyuiopè
nome utente inserito: qwertyuiopè
```

correct input

```
(kali㉿kali)-[~/C]  
$ ./BOF  
Si prega di inserire il nome utente:eyywefbwefwyfwfywnfwefuwfnwief  
nome utente inserito: eyywefbwefwyfwfywnfwefuwfnwief  
zsh: segmentation fault ./BOF
```

Segmentation
fault

Capiamo di essere andati troppo oltre quando sul terminale compare “segmentation fault”. Concludiamo l’esercitazione con l’ultima task. Cambiamo la grandezza del buffer dal programma, ricompiliamo, e diamoci alla pazza gioia con la tastiera, di nuovo.

```
$ ./BOF  
Si prega di inserire il nome utente:djkfndjnfdsfdsfndsjfnkjfdnkjsndnfjsdfdsj  
fndsfkjdnfjnsdnkfknsdfjkjsdfjkdsnsdjknj  
nome utente inserito: djkfndjnfdsfdsfndsjfnkjfdnkjsndnfjsdfdsjfndsfkjdnfjns  
dnkfknsdfjkjsdfjkdsnsdjknj  
zsh: segmentation fault ./BOF
```

30 characters buffer
broken