Creazione file su Kali

```
┌──(kali㉿kali)-[~]
└─$ cd /home/kali/Desktop

┌──(kali㉿kali)-[~/Desktop]
└─$ sudo nano BOF.c
[sudo] password for kali:

┌──(kali㉿kali)-[~/Desktop]
└─$ gcc -g BOF.c -o BOF

┌──(kali㉿kali)-[~/Desktop]
└─$ ./BOF
Inserisci nome utente: ileana
Il nome utente inserito è: ileana

┌──(kali㉿kali)-[~/Desktop]
└─$ ./BOF
Inserisci nome utente: ahahhahahahahahahahahahahahaha
Il nome utente inserito è: ahahhahahahahahahahahahahahaha
zsh: segmentation fault  ./BOF

┌──(kali㉿kali)-[~/Desktop]
└─$
```

```
  GNU nano 5.4
#include <stdio.h>

int main () {

char buffer [10];

printf("Inserisci nome utente: ");
scanf("%s", buffer);

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

return 0;



}
```

Modifica dimensione vettore a 30

```
  GNU nano 5.4
#include <stdio.h>

int main () {

char buffer [30];

printf("Inserisci nome utente: ");
scanf("%s", buffer);

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

return 0;



}
```

Come evidenzia il segmentation fault in basso, questa misura non è sufficiente ad eliminare la possibilità di generare un BOF

```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./BOF
Inserisci nome utente: ahahahahahahahahahahahahahahahahahahahahahahahaha
Il nome utente inserito è: ahahahahahahahahahahahahahahahahahahahahahahahaha
zsh: segmentation fault  ./BOF

┌──(kali㉿kali)-[~/Desktop]
└─$
```

Anche in questo caso infatti si verifica un crash del programma, perché il buffer può contenere solo fino a 30 caratteri (29 caratteri + il terminatore di memoria nullo /0) ma non c'è sanificazione dell'input, dunque un attaccante potrebbe inserire un argomento più lungo dei 30 bytes allocati nello stack per questa funzione, sovrascrivendo il return address dello stack, e causare un Denial of Service.
In questo caso quindi i caratteri che eccedono la lunghezza massima del vettore *buffer* andranno a riempire il return address dello stack (nella loro forma esadecimale), che diventerà il EIP della funzione. Quando la funzione riprenderà la sua esecuzione, l'EIP troverà a quell'indirizzo di memoria solo codice random (o uno spazio vuoto) e il programma andrà in crash.

Nel caso di un buffer overflow inoltre un attaccante potrebbe anche sovrascrivere il EIP sostituendolo con un indirizzo di memoria cui ha accesso, per inserirvi codice malevolo e ottenere successivamente accesso alla macchina vittima.
Il buffer overflow di questo codice potrebbe essere evitato implementando del codice per la validazione dell'input e assicurarsi del fatto che solo il numero massimo consentito di caratteri, nonché solo il tipo di caratteri consentiti, vengano inseriti nello stack di memoria.

Alternative al codice proposto:

1. Invece di *scanf* si potrebbe usare *fgets*, che è generalmente più sicura perché è possibile specificare una dimensione massima del buffer. Fgets legge fino al newline o finché il numero dei caratteri non è uguale a n-1 e consente spazi all'interno della stringa.
2. Invece di *scanf (%s)* si potrebbe utilizzare *scanf("%29s")*, che specifica il massimo numero di caratteri accettati nel vettore. Scanf non legge il newline e non consente spazi all'interno della stringa, a meno che non si inserisca lo scanset: *scanf("%[^\n]", buffer)*;

3. Validazione dell'input del nome utente per assicurarsi che solo la lunghezza massima e il formato corretto siano consentiti.

In tutti i casi precedenti però è necessario svuotare il buffer perché i caratteri extra resteranno nel buffer. A questo fine, si potrebbe utilizzare *strcspn(buffer, "\n")*; che trova la posizione del newline nel nome e, se presente, assegna a quella posizione il marker di endpoint della stringa '\0'

oppure *char buffer[30] = {'\0'}*;.

In alternativa, si può aggiungere uno spazio prima dello specificatore di formato: *scanf(" %29s")*.

Alternativa 1

```c
#include <stdio.h>

int main () {

char buffer [30];

printf("Inserisci nome utente: ");
fgets(buffer, sizeof(buffer), stdin);

buffer[strcspn(buffer, "\n")] = '\0';

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

return 0;



}
```

Alternativa 2

```c
#include <stdio.h>

int main () {

char buffer [30] = {'\0'};

printf("Inserisci nome utente: ");
scanf(" %29s", buffer);

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

return 0;



}
```

Alternativa 3

```c
#include <stdio.h>


int nomeUtenteValido(char *buffer, int lunghezza)

{
    int nomeValido = '\0';

    while (!nomeValido)
    {
        printf("\nInserisci il tuo nome (massimo %d caratteri): \n", lunghezza - 1);
        fgets(buffer, lunghezza, stdin);
        buffer[strcspn(buffer, "\n")] = '\0';

        if (strlen(buffer) >= lunghezza)
        {
            printf("Il tuo nome è troppo lungo! Per favore, inserisci un nome di al massimo 29 caratteri (spazi inclusi).\n");
            getchar() != '\n'; // Pulisce il buffer
        }
        else
        {
            buffer[strcspn(buffer, "\n")] = '\0'; // Trova la posizione del newline nel nome e, se presente, assegna a quella posizione il marker di endpoint della stringa '\0'
            nomeValido = 1;                  // Se il nome è valido, esce dal loop
        }
    }
}


int main() {

    char buffer[30];

    nomeUtenteValido(buffer, sizeof(buffer));

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

    return 0;
}
```

Altre tecniche di mitigazione per questo attacco sono l'Address Space Layout Randomization (ASLR) che previene attacchi con shellcode, randomizzando gli indirizzi di memoria dello stack oltre che di altre sezioni critiche di un programma e il Data Execution Prevention (DEP), che impedisce l'esecuzione di codice in regioni di memoria che non dovrebbero contenere istruzioni eseguibili marcando determinate aree come non eseguibili. Spesso ASLR e DEP vengono utilizzati insieme per rendere più difficile agli attaccanti sfruttare vulnerabilità nelle applicazioni utilizzando shellcode oppure Return-Oriented Programming (ROP).