Pratica S7L4

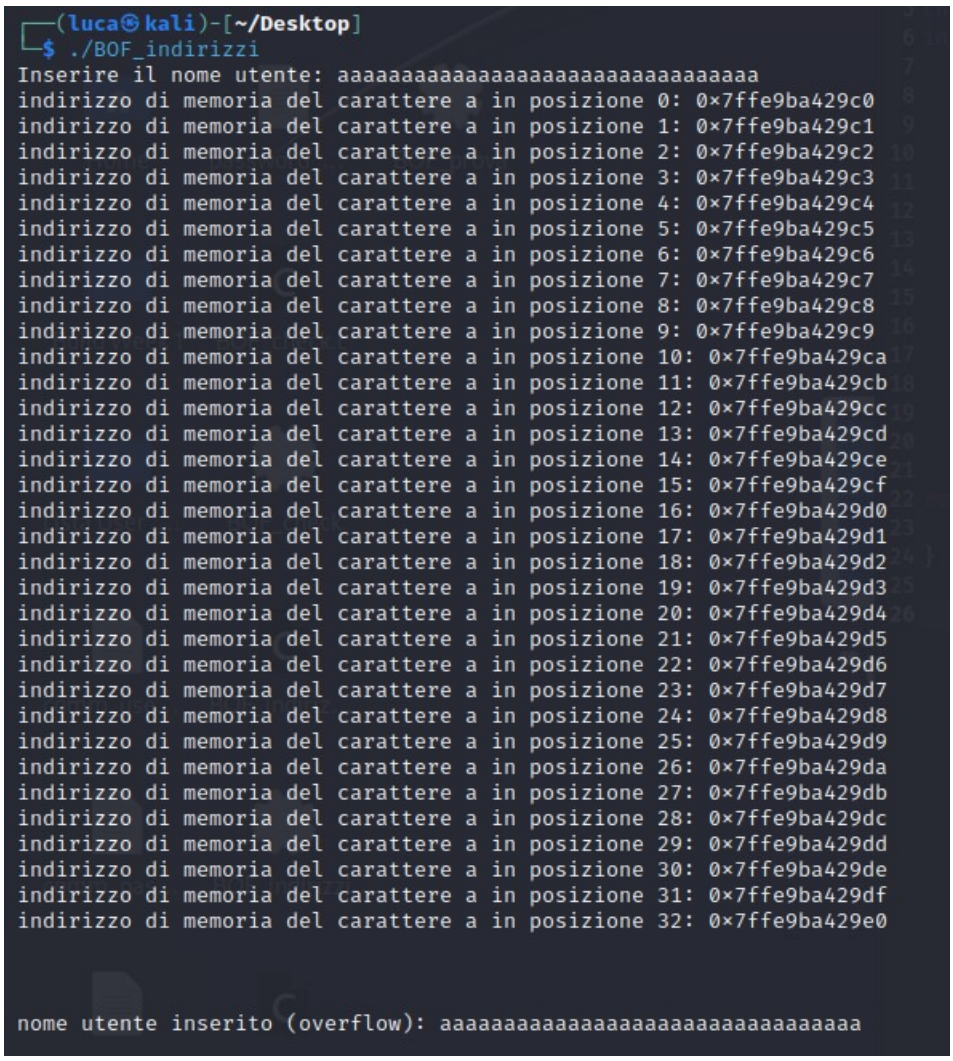
Consideriamo il seguente programma:

```c
#include <stdio.h>
#include <string.h>

int main () {
char buffer [30];
int lunghezza_nome = 0;

        printf("Inserire il nome utente: ");
        scanf("%s", buffer);
        lunghezza_nome = strlen(buffer); //assegno a lunghezza_nome la lunghezza del nome inserito dall'utente
        if (lunghezza_nome > 30) {
                for (int i=0; i < lunghezza_nome; i++) { //incremento i finché non ho raggiunto la lunghezza del nome
                        printf("indirizzo di memoria del carattere %c in posizione %d: %p\n", buffer[i], i,  &buffer[i]);
                }
                printf("\n\n\nnome utente inserito (overflow): %s\n", buffer);
        }
        else {
                printf("nome utente inserito: %s\n", buffer);
        }


return 0;

}


```

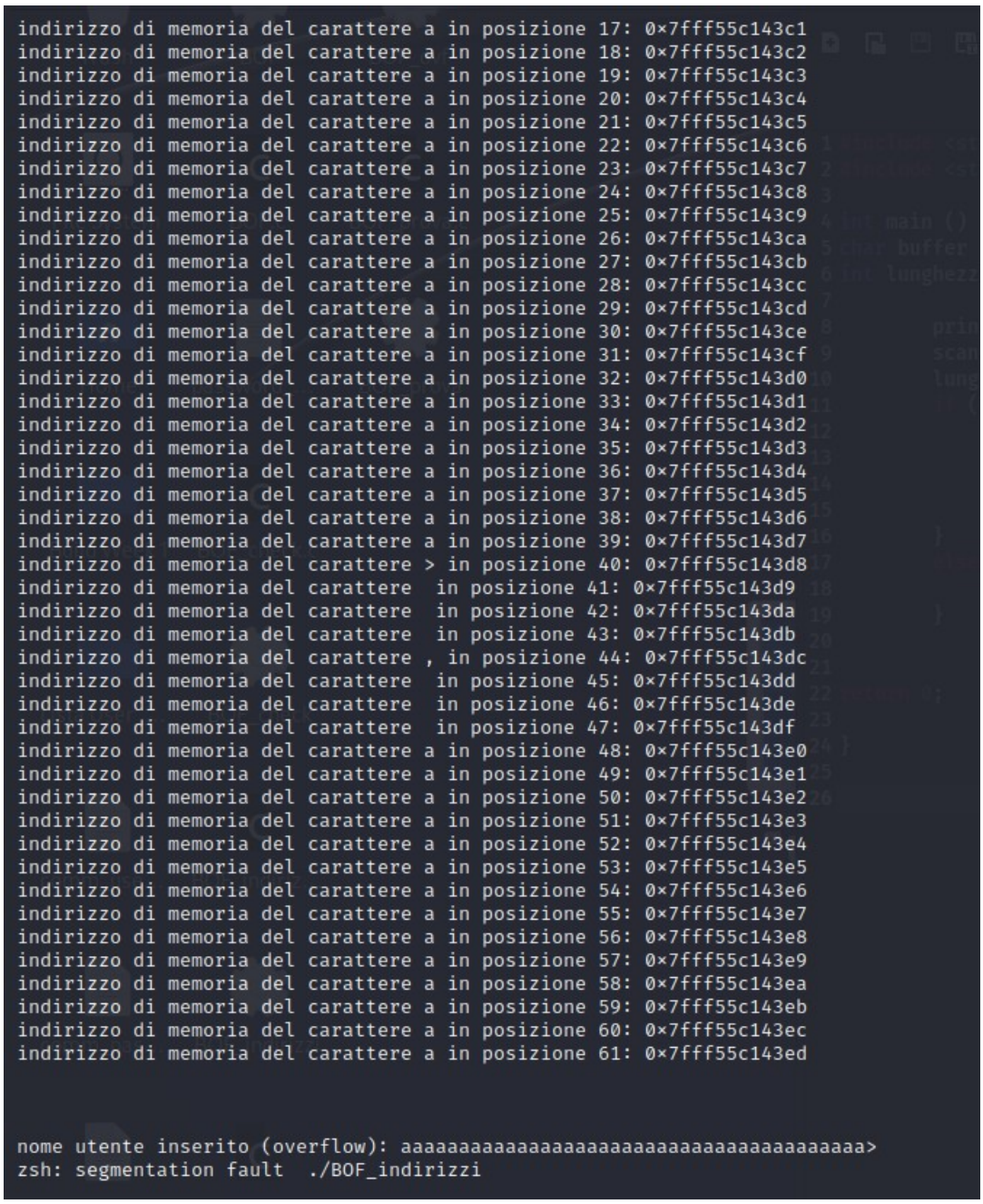
Il programma accetta in input una stringa dall'utente di dimensione predefinita (30 caratteri). Se il nome inserito è minore di 30 caratteri restituisce a schermo il nome inserito. Vediamo un esempio:

```
┌──(luca㉿kali)-[~/Desktop]
└─$ ./BOF_indirizzi
Inserire il nome utente: LucaDanelli
nome utente inserito: LucaDanelli
```

La situazione cambia se l'utente sceglie di inserire un input molto più lungo, che vada ad eccedere il buffer preallocato per la variabile buffer. Una volta dichiarata una variabile, C assegna uno spazio in memoria di dimensione predefinita, in questo caso andrà a riservare spazio per una stringa da 30 caratteri come da dichiarazione alla riga 5. Per chiarire meglio la situazione, il programma stampa gli indirizzi di memoria corrispondenti a ciascun carattere inserito dall'utente con un ciclo for (riga 11) servendosi della funzione **strlen()** in modo che esca dal ciclo una volta raggiunta la lunghezza della stringa inserita dall'utente. Vediamo i risultati andando a stampare una stringa di 33 caratteri:



Come vediamo gli indirizzi in memoria di ogni carattere sono incrementali, ed espressi in esadecimale. Ci siamo serviti del puntatore **&buffer[i]** per accedere all'indirizzo di memoria all'indice i ad ogni iterazione del ciclo for. Per ora però il programma funziona ancora bene, nel senso che la stringa riportata in output è esattamente quella in input. Proviamo quindi ad aumentare ancora la lunghezza della stringa inserita dall'utente, portandola ad esempio a 62 caratteri:



In questo caso possiamo notare due cose. Il programma smette di leggere correttamente la stringa dalla posizione 40 in poi, dove legge un carattere **“>”** nonostante il nostro input da tastiera fosse “a”. Inoltre termina restituendo l'errore **segmentation fault.** Questa situazione prende il nome di **buffer overflow.** In caso il codice sia vulnerabile come in questo caso, un attaccante potrebbe sfruttare questa situazione per mandare in crash il programma o potenzialmente anche accedere ad aree di memoria attigue a quelle previste per gli input, dato che come si vede dall'output sopra al momento della lettura della stringa la memoria è stata richiamata fino alla cella con indirizzo **0x7fff55c143ed** nonostante il buffer compreso di tolleranza arrivasse sino all'indirizzo **0x7fff55c143d7**