Report Buffer Overflow - Traccia 3

> A cura di Pierantonio Miglietta, Iris Canole, Rebecca Talone, Francesco Miolli, Tiziano Bramonti, Andrea Sottile, Alessandro Ricci

Obiettivi

Dato il seguente programma in C:

```c
#include <stdio.h>

int main () {

int vector [10], i, j, k;
int swap_var;


printf ("Inserire 10 interi:\n");

for ( i = 0 ; i < 10 ; i++)
  {
  int c= i+1;
  printf("[%d]:", c);
  scanf ("%d", &vector[i]);
  }


printf ("Il vettore inserito e':\n");
for ( i = 0 ; i < 10 ; i++)
        {
        int t= i+1;
        printf("[%d]: %d", t, vector[i]);
  printf("\n");
  }


for (j = 0 ; j < 10 - 1; j++)
  {
  for (k = 0 ; k < 10 - j - 1; k++)
    {
      if (vector[k] > vector[k+1])
```

```
      {
      swap_var=vector[k];
      vector[k]=vector[k+1];
      vector[k+1]=swap_var;
      }
    }
  }
printf("Il vettore ordinato e':\n");
for (j = 0; j < 10; j++)
  {
  int g = j+1;
  printf("[%d]:", g);
  printf("%d\n", vector[j]);
  }

return 0;


}
```

- descrivere il funizonamento del programma prima dell'esecuzione
- Riprodurre ed eseguire il programma ne laboratorio e valutare le ipotesi fatte precedentemente sul funzionamento del programma
- Modificare il programma affinché si verifichi un errore di segmentazione

> **Tip**
>
> Ricordare che un BOF sfrutta una vulnerabilità nel codice relativo alla mancanza di controllo dell'input utente rispetto alla capienza del vettore di destinazione. Concentrarsi quindi per trovare la soluzione nel punto dove l'utente può inserire valori in input, e modificare il programma in modo tale che l'utente riesca ad inserire più valori di quelli previsti.

Interpretazione del programma iniziale

- `vector [10]`: Un **array** che può contenere 10 numeri interi
- `i`, `j`, `k`: Variabili usate come **contatori** per i cicli
- `swap_var`: Variabile temporanea usata per lo **scambio** di valori durante l'ordinamento
- Il programma chiede all'utente di inserire 10 numeri interi usando la funzione `printf ("Inserire 10 interi:\n");`
- Il primo ciclo `for ( i = 0 ; i < 10 ; i++)` legge i 10 valori e li memorizza nell'array `vector`
- Il secondo ciclo `for ( i = 0 ; i < 10 ; i++)` stampa i 10 numeri e li stampa nell'ordine esatto in cui vengono inseriti dall'utente in console
- `if (vector[k] > vector[k+1])`: qui avviene una sorta di ordinamento, nel quale se il numero corrente è maggiore del successivo, allora l'elemento più grande viene spostato alla fine dell'array ad

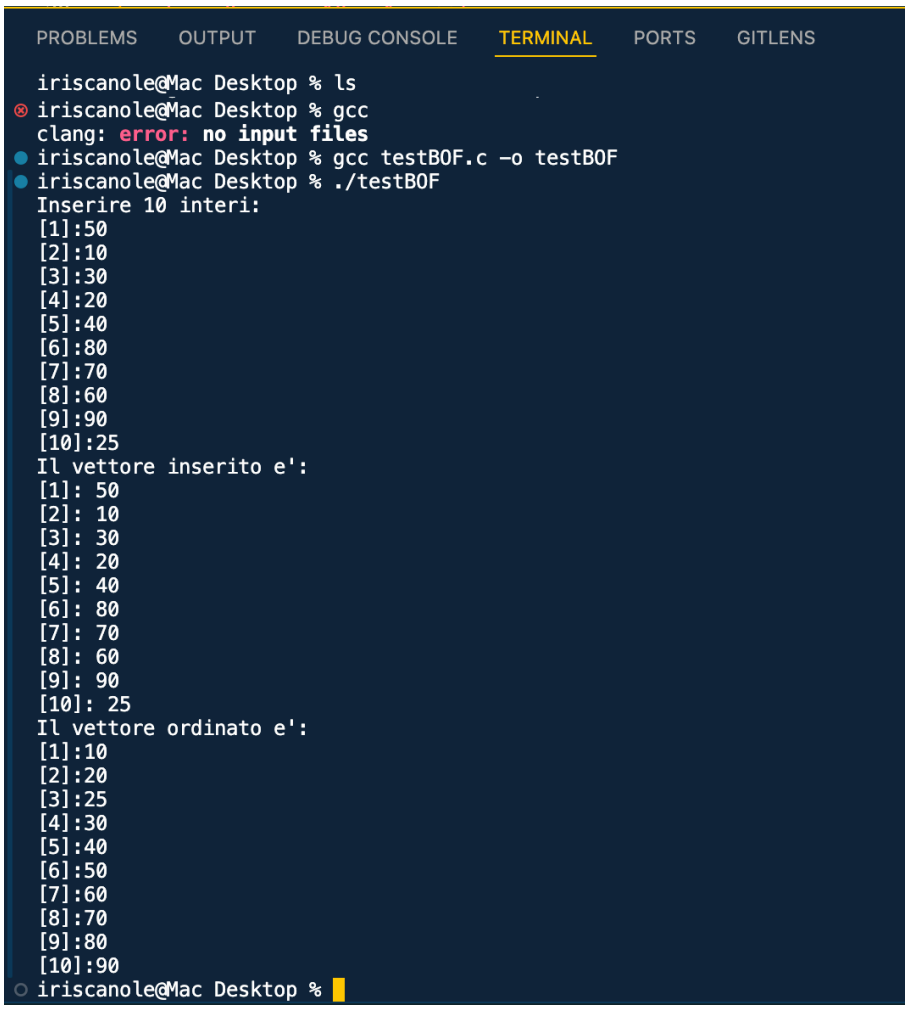
ogni passaggio

- Il risultato è un array **ordinato in modo crescente** (dal più piccolo al più grande)
- L'ultimo ciclo `for (j = 0; j < 10; j++)` stampa il vettore dopo che è stato ordinato.

Sulla base delle interpretazioni di cui sopra, quello che fa il programma è con molta probabilità un algoritmo di tipo **Bubble Sort**, il quale confronta ripetutamente coppie di elementi adiacenti e li scambia se sono nell'ordine sbagliato.

Prova del programma in un ambiente idoneo

Abbiamo provato a lanciare il programma all'interno di Visual Studio Code per vederne nel concreto il funzionamento. Dall'immagine possiamo vedere l'output, che è coerente con la nostra interpretazione del funzionamento del programma:



Come si può evincere dall'immagine:

- abbiamo inserito come prova 10 numeri casuali non messi nell'ordine corretto
- il programma come prima cosa ci stampa il vettore così come lo abbiamo inserito
- di conseguenza, dopo l'esecuzione dell'algoritmo di sorting, il programma ci stampa il vettore ordinato in modo crescente

Modifica del programma affinché si verifichi un errore di segmentazione

A questo punto, dobbiamo modificare tale programma in tal modo che si verifichi un errore di segmentazione, come da obiettivo principale.

Il programma modificato è il seguente:

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

// Definizioni
#define SIZE 10


// FUNZIONE PER LEGGERE L'INPUT IN MODO VULNERABILE (CHIAREZZA SUL BOF)

void BOF(int *vettore, int dim) {

    int i;


    printf(">> Inserisci 10 valori.\n");
    printf("Per innescare il BOF, inserisci una stringa di 5 o piu' caratteri al primo prompt \n");

    for (i = 0; i < dim; i++) {
        int c = i + 1;
        printf("[%d]:", c);

        // VULNERABILITÀ: Lettura di una stringa (%s) nell'indirizzo di un intero (char*).
        // Un intero (int) è un buffer di soli 4 byte. Una stringa di 5+ caratteri
        // sovrascriverà l'int successivo (vettore[i+1]).
        if (scanf("%s", (char*)&vettore[i]) != 1) { //Quando scanf restituisce un valore diverso da 1, significa che il numero di elementi correttamente letti e assegnati è diverso da uno. Questo accade se l'input non corrisponde al formato specificato, causando un errore di lettura
             printf(" Errore di lettura input (non una stringa).\n");
             while (getchar() != '\n');
             i--; // Riprova
             continue;
        }

        // Nota: Il valore stampato e' il risultato della conversione ASCII della stringa
        // nei 4 byte dell'int. Un BOF palese altera i valori successivi.
        printf("  [LETTURA CORROTTA (ASCII)]: %d\n", vettore[i]);
    }

    printf("\n[TENTATIVO BOF ESEGUITO SU TUTTO IL VETTORE]\n");
}


// FUNZIONE PER LEGGERE L'INPUT IN MODO SICURO (CORRETTO con CONTROLLI DI SICUREZZA)

void leggi_input_corretto(int *vettore, int dim) {
    int i;
    printf("\n--- SICURO: Utilizzo di scanf(\"%%d\") con CONTROLLI DI SICUREZZA ---\n");
    printf("Inserire 10 interi (solo numeri accettati):\n");

    for (i = 0; i < dim; i++) {
        int c = i + 1;
        printf("[%d]:", c);

        // CONTROLLO DI SICUREZZA 1: Verifica che scanf abbia letto un intero.
        if (scanf ("%d", &vettore[i]) != 1) {
             printf(" [ERRORE DI SICUREZZA]: Input non numerico! Riprova.\n");
             vettore[i] = 0;

             // CONTROLLO DI SICUREZZA 2: Pulisce il buffer di input (stdin)
             int ch;
             while ((ch = getchar()) != '\n' && ch != EOF);

             // Decrementa l'indice per far ripetere la lettura corretta dell'elemento
             i--;
        }
    }
}

// ------------------------------------------------------------------------
// FUNZIONE PRINCIPALE
// ------------------------------------------------------------------------
int main () {

    int vector[SIZE];
    int i, j, k;
    int swap_var;
    int scelta;

    // Inizializza il vettore a zero
    memset(vector, 0, sizeof(vector));
```

```c

    printf("Selezionare l'opzione di esercizio:\n");
    printf("1. Esegui Codice Corretto (Input Sicuro con Controlli)\n");
    printf("2. Esegui Codice BOF (Input Dinamico Insecuro - Corrompe l'array)\n");
    printf("Scelta: ");

    if (scanf("%d", &scelta) != 1) {
        printf("Errore: Input non valido.\n");
        return 1;
    }
    while (getchar() != '\n');

    // --- PUNTO DI INGRESSO (SCELTA) ---
    switch (scelta) {
        case 1:
            leggi_input_corretto(vector, SIZE);
            break;
        case 2:
            BOF(vector, SIZE);
            break;
        default:
            printf("Scelta non valida.\n");
            return 1;
    }


    // Stampa del Vettore
    printf ("\n Il vettore inserito e' (dopo l'input):\n");
    for (i = 0; i < SIZE; i++) {
        int t = i + 1
        printf("[%d]: %d", t, vector[i]);
        printf("\n");
    }

    // Algoritmo di Ordinamento a bolle (Bubble Sort)
    printf("\n Avvio dell'algoritmo di Bubble Sort...\n");
    // Se la memoria e' stata corrotta (Opzione 2), il bubble sort potrebbe fallire.
    for (j = 0; j < SIZE - 1; j++) {
        for (k = 0; k < SIZE - j - 1; k++) {
            if (vector[k] > vector[k+1]) {
                swap_var = vector[k];
                vector[k] = vector[k+1];
                vector[k+1] = swap_var;
            }
        }
    }

    printf("\nIl vettore ordinato e':\n");
    for (j = 0; j < SIZE; j++) {
        int g = j + 1;
        printf("[%d]:", g);
        printf("%d\n", vector[j]);
    }

    return 0;
}
```

Andiamo ad analizzarlo:

- Il programma ha due modalità: una modalità **sicura** che legge 10 interi con `scanf("%d", ...)` e controlli, e una modalità **vulnerabile** (`BOF`) che legge una stringa con `scanf("%s", (char*)&vettore[i])` direttamente sull'indirizzo di un `int` dell'array. La scelta tra le due viene fatta all'avvio dall'utente.
- **Linea critica**: nella funzione `BOF` la riga `25 - scanf("%s", (char*)&vettore[i])` legge una stringa illimitata e la scrive in memoria a partire dall'indirizzo di un `int`. Questo è dichiarato anche come commento nel file.
- **Perché è pericoloso**: `scanf("%s", ...)` non impone limiti di lunghezza — se inserisci più byte di quanti ne contiene il `int` (tipicamente 4), la scrittura continuerà negli indirizzi memoria successivi (cioè `vettore[i+1]`, `vettore[i+2]`, e possibilmente nelle altre variabili locali o nel frame di ritorno). Questo è un classico *stack-based buffer overflow / memory corruption*.
- **Corruzione dell'array**: i primi 4 byte di ogni `vettore[i]` verranno reinterpretati come `int` (`ASCII` → valore numerico) e stampati come tali nella riga `printf("  [LETTURA CORROTTA (ASCII)]: %d\n", vettore[i]);`. Quindi vediamo subito valori "strani" al termine dell'input.

- Quali potrebbero essere le possibili conseguenze di tutto ciò?
    - valori corrotti all'interno dell'array (e quindi output errati)
    - sovrascrittura delle altre variabili locali (`i, j, k, swap_var, scelta`) definite dopo l'array. Ciò potrebbe alterare il loro comportamento
    - sovrascrittura dei puntatori di frame o dell'indirizzo di ritorno: con un overflow sufficientemente grande si può far fallire il programma (segmentation fault) o, in casi estremi, alterare il controllo di esecuzione.

Procedura di riproduzione

1. Avvio del programma
2. Scelta della modalità "BOF / input insicuro" (`opzione 1` oppure `opzione 2`)
3. Al primo prompt, **inserire una stringa con ≥5 caratteri**
4. Completare gli altri input (anche numerici)
5. Osservare la stampa dell'array "dopo l'input": valori incoerenti/insensati indicano corruzione.
6. Quando parte il **bubble sort**, la memoria corrotta può portare a **SIGSEGV** (dipende dall'input e dallo stato dello stack).

Di seguito possiamo vedere come il programma trascrive l'output con "LETTURA CORROTTA (ASCII)":

```
Selezionare l'opzione di esercizio:
1. Esegui Codice Corretto (Input Sicuro con Controlli)
2. Esegui Codice BOF (Input Dinamico Insecuro - Corrompe l'array)
Scelta: 2
>> Inserisci 10 valori.
Per innescare il BOF, inserisci una stringa di 5 o piu' caratteri al primo prompt
[1]:AAAAAAA
  [LETTURA CORROTTA (ASCII)]: 1094795585
[2]:343
  [LETTURA CORROTTA (ASCII)]: 3355699
[3]:53225
  [LETTURA CORROTTA (ASCII)]: 842150709
[4]:333
  [LETTURA CORROTTA (ASCII)]: 3355443
[5]:222
  [LETTURA CORROTTA (ASCII)]: 3289650
[6]:233
  [LETTURA CORROTTA (ASCII)]: 3355442
[7]:421
  [LETTURA CORROTTA (ASCII)]: 3224116
[8]:21
  [LETTURA CORROTTA (ASCII)]: 12594
[9]:21
  [LETTURA CORROTTA (ASCII)]: 12594
[10]:2
  [LETTURA CORROTTA (ASCII)]: 50

[TENTATIVO BOF ESEGUITO SU TUTTO IL VETTORE]
```

Impatto e rischi

Un overflow così banale dimostra come **validazione input assente + tipi non coerenti** possa compromettere integrità dei dati e stabilità del processo. In contesti reali, lo stesso pattern (%s su buffer non adeguato) può portare ad **esecuzione di codice arbitrario** o **data leakage**.

Possibile mitigazione:

La variante sicura usa `scanf("%d", &vettore[i])` con **controllo del valore di ritorno** e **pulizia del buffer di input** per scartare dati non numerici e ripetere la lettura. Questo previene l'overflow e l'iniezione di byte arbitrari nell'array di interi.

Conclusioni

L'errore di segmentazione è stato **indotto volontariamente** corrompendo la memoria durante la fase di input (lettura di stringa dentro un `int`). La prova evidenzia come una singola riga "sbagliata" nella gestione dell'input basti a compromettere l'intero flusso: i dati risultano incoerenti già in output e l'algoritmo successivo (bubble sort) può terminare con crash. Il confronto con la versione che valida l'input mostra che **controlli minimi** (formato corretto, scarto del buffer residuo) sono sufficienti a eliminare il problema alla radice.