

# Report Esercizio 3 - Buffer Overflow Exercise

## Introduzione

Questo report documenta l'analisi e la modifica di un programma di ordinamento per dimostrare le vulnerabilità di buffer overflow. L'esercizio ha trasformato un programma originariamente sicuro in uno vulnerabile attraverso modifiche mirate al controllo dell'input utente.

## 1. Analisi del Programma Originale

### Descrizione del Funzionamento

Il programma chiede all'utente di **inserire 10 numeri interi**, li memorizza in un vettore (array) di **dimensione fissa** e poi li ordina in ordine **crescente** usando un metodo di ordinamento. Alla fine, mostra sia il vettore così come è stato inserito dall'utente, sia lo stesso vettore una volta ordinato, applica l'algoritmo di **Bubble Sort** per riorganizzare gli elementi in ordine crescente. Infine, presenta all'utente il risultato finale mostrando la sequenza ordinata.

Il flusso di esecuzione è completamente lineare e prevedibile: **input → visualizzazione dati originali → ordinamento → visualizzazione risultato ordinato**. Non sono presenti menu di scelta, opzioni alternative o percorsi di esecuzione dinamici.

**In pratica, serve per vedere come si può leggere, stampare e ordinare una sequenza di numeri.**

### Analisi delle componenti principali

#### 1. Dichiarazioni iniziali

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

- **vector[10]** → array che contiene i 10 numeri.
- **i, j, k** → variabili di controllo per i cicli for.
- **swap\_var** → variabile d'appoggio usata per scambiare i valori durante l'ordinamento.

## 2. Inserimento dei dati

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

- **Il programma chiede 10 numeri all'utente.**
- **Con scanf** salva ogni numero nell'array.
- **La stampa [%d]:** serve a indicare all'utente quale numero deve inserire (1°, 2°, ..., 10°).

## 3. Stampa del vettore inserito

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

- **Mostra** i valori nell'ordine in cui sono stati digitati.
- **Usa un indice numerato** [1], [2], ... per renderli leggibili.

## 4. Ordinamento (Bubble Sort)

```
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;  
        }  
    }  
}
```

- Qui entra in gioco l'algoritmo di **Bubble Sort**:

- **Confronta** due numeri vicini (vector[k] e vector[k+1]).
- Se non sono in ordine, **li scambia**.
- **Ripete più volte** fino a mettere tutti i numeri in ordine crescente.

## 5. Stampa del vettore ordinato

```

    }
    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;

}

```

- Alla fine, **mostra l'array ordinato** dal più piccolo al più grande.

## In sintesi il programma:

1. Legge 10 numeri da tastiera.
2. Li stampa nell'ordine di inserimento dell'utente.
3. Li ordina con Bubble Sort.
4. Li ristampa in ordine crescente.

## Caratteristiche di Sicurezza

Il programma originale presenta diverse caratteristiche che lo rendono **sicuro** rispetto ai buffer overflow. Tutti i cicli utilizzano limiti che corrispondono esattamente alla capacità dell'array allocato. Non esistono variabili che possano influenzare dinamicamente questi limiti, **eliminando la possibilità di accessi illegali alla memoria**. La logica del programma è completamente statica e prevedibile, senza dipendenze da input utente per determinare i confini operativi.

## 2. Analisi del Programma Modificato

### Identificazione del Punto di Vulnerabilità

La traccia dell'esercizio indicava di concentrarsi "**nel punto dove l'utente può inserire valori in input**" per introdurre la vulnerabilità. Il punto critico identificato è stato **il meccanismo di controllo del ciclo di input**, dove la sostituzione di un **limite fisso** con un **valore dinamico** fornito dall'utente ha creato la condizione necessaria per il buffer overflow.

### Modifiche Implementate

#### Introduzione della Variabile di Controllo e della Modalità

```
int num_elementi;  
int modalita;
```

**int** = tipo di dato intero

**num\_elementi** = nome della prima variabile

**modalita** = nome della seconda variabile

L'aggiunta di questa variabile **num\_elementi** introduce un elemento dinamico che rompe la staticità del programma originale. Questa variabile permette all'utente di specificare quanti elementi desidera inserire, **rimuovendo il vincolo fisso di dieci elementi**. In aggiunta la **variabile modalità**, consentirà all'utente di scegliere tra versione **sicura o vulnerabile**( Extra)

#### Nuovo Meccanismo di Input Utente

```
printf("Quanti interi vuoi inserire (1-10)? ");  
scanf("%d", &num_elementi);
```

**printf** = funzione di stampa

**Quanti interi vuoi inserire (1-10)?** = messaggio mostrato all'utente

**scanf** = funzione per leggere input dell'utente

**"%d"** = formato per leggere un numero intero

**&num\_elementi** = indirizzo della variabile dove salvare il numero inserito

Questa modifica rappresenta il nucleo della vulnerabilità. Il programma accetta un valore numerico dall'utente senza applicare alcuna validazione o controllo sui limiti. L'utente può inserire qualsiasi valore intero, inclusi numeri molto maggiori della capacità dell'array.

#### Modifica del Ciclo di Input

##### *Versione originale:*

```
for (i = 0; i < 10; i++)
```

### Versione modificata:

```
for (i = 0; i < num_elementi; i++) {
```

**for** = parola chiave per iniziare un ciclo

**i = 0** = inizializza la variabile contatore i a 0

**;** = separa le tre parti del ciclo for

**i < num\_elementi** = condizione: continua finché **i** è minore del numero di elementi

**;** = separa la seconda dalla terza parte

**i++** = incrementa i di 1 ad ogni giro

La sostituzione del limite fisso 10 con la variabile **num\_elementi** trasforma un ciclo sicuro in uno potenzialmente pericoloso. Quando **num\_elementi** supera 10, **il ciclo continua ad iterare** oltre i confini dell'array, **causando scritture in locazioni di memoria non allocate per l'array**.

### Propagazione della Vulnerabilità

La modifica del ciclo for spiegata sopra si propaga a tutti i cicli che operano sull'array:

### Visualizzazione dell'array:

```
printf("\nIl vettore inserito e':\n");  
for (i = 0; i < num_elementi; i++) {  
    int t = i + 1;  
    printf("[%d]: %d", t, vettore[i]);
```

**int** = dichiara variabile intera

**t** = nome variabile

**= >>>** operatore di assegnazione

**i + 1** = valore di **i** più 1

**printf** = funzione stampa

**"[%d]: %d"** = formato stringa con due placeholder per numeri

**t** = primo valore da stampare (posizione)

**vettore[i]** = secondo valore (elemento dell'array alla posizione i)

### Algoritmo di ordinamento:

```
for (j = 0; j < num_elementi - 1; j++) {  
    for (k = 0; k < num_elementi - j - 1; k++) {
```

**for** = inizia il primo ciclo

**j = 0** = inizializza j a 0

**;** = separa le condizioni

**j < num\_elementi - 1** = j deve essere minore di (numero elementi meno 1)

**;** = separa

**j++** = incrementa j

**for** = inizia secondo ciclo annidato

**k = 0** = inizializza k a 0

**;** = separa

**k < num\_elementi - j - 1** = k minore di (elementi meno j meno 1)

**;** = separa

**k++** = incrementa k

### Meccanismo del Buffer Overflow

Quando l'utente specifica un valore di **num\_elementi** superiore a 10, si verifica la seguente sequenza:

1. **Fase 1** (i = 0 a 9): I primi dieci elementi vengono scritti nelle **posizioni legittime** dell'array
2. **Fase 2** (i = 10 in poi): Gli elementi successivi vengono scritti in locazioni di memoria adiacenti che **non appartengono** all'array
3. **Corruzione progressiva**: Ogni elemento oltre il decimo **corrompe dati in memoria**, potenzialmente alterando altre variabili o strutture di controllo del programma

### Conseguenze della Modifica

La rimozione del controllo sui limiti dell'input trasforma completamente la natura del programma. Da un'applicazione completamente prevedibile e sicura, diventa un programma in grado di corrompere la propria memoria di esecuzione.

### 3. Sistema di Controllo e Menu

#### Implementazione del Menu di Scelta

Per soddisfare i requisiti dell'esercizio, è stato implementato un sistema di menu che permette all'utente di scegliere tra modalità sicura e vulnerabile:

```
printf("1. Modalità SICURA (con controlli di input)\n");  
printf("2. Modalità VULNERABILE (senza controlli - possibile SEGFAULT)\n");
```

Questo approccio permette di dimostrare chiaramente la differenza tra un'implementazione sicura e una vulnerabile utilizzando lo stesso codice base.

#### Controlli di Input (Modalità Sicura)

Nella modalità sicura abbiamo implementato un meccanismo ulteriore di validazione che previene ulteriormente il buffer overflow:

```
do {  
    printf("Quanti interi vuoi inserire (1-10)? ");  
    scanf("%d", &num_elementi);  
  
    if (num_elementi < 1 || num_elementi > 10) {  
        printf("ERRORE: Il numero deve essere tra 1 e 10!\n");  
    }  
} while (num_elementi < 1 || num_elementi > 10);
```

Il ciclo **do-while** garantisce che il programma **non proceda** finché l'utente non fornisce un valore compreso **nell'intervallo sicuro**. Questo controllo rafforza maggiormente la sicurezza del programma mantenendo la flessibilità di permettere all'utente di scegliere un numero di elementi inferiore a dieci.

### 4. Test e Risultati

#### Test con Modalità Sicura

Utilizzando la modalità sicura, il programma si **comporta correttamente** indipendentemente dai tentativi dell'utente di inserire valori fuori range. **I controlli di validazione aggiunti** impediscono l'avanzamento del programma fino a quando non viene fornito un input valido.

Se inseriamo 15 per esempio il programma darà errore ed il seguente messaggio :

**ERRORE: il numero deve essere tra 1 e 10!**

## Test con Modalità Vulnerabile - Overflow Contenuto

Il primo test della modalità vulnerabile è stato condotto inizialmente con **100 elementi**. Il programma ha accettato l'input e ha proceduto con l'esecuzione, ma durante la visualizzazione **sono emersi chiari segni di corruzione dei dati**. Valori anomali come **-9984** sono apparsi nei risultati, **indicando che il programma stava leggendo e manipolando dati corrotti in memoria**.

## Test con Modalità Vulnerabile - Overflow Massiccio

Il test definitivo è stato eseguito utilizzando **10.000 elementi**. Questa quantità massiccia di dati ha superato le capacità di gestione del sistema, causando il **Segmentation Fault** desiderato. Il crash del programma ha confermato che il buffer overflow era stato implementato correttamente e che la vulnerabilità era effettivamente sfruttabile.

Vediamo ogni fase:

### Avvio del programma e comparsa del menù

```
PROGRAMMA DI ORDINAMENTO CON GESTIONE BUFFER OVERFLOW
=====
Scegli la modalità di esecuzione:

1. Modalità SICURA (con controlli di input)
2. Modalità VULNERABILE (senza controlli - possibile SEGFAULT)
3. Esci

Inserisci la tua scelta (1-3): █
```

Inseriamo la modalità vulnerabile per eseguire il test, la selezioniamo digitando il numero 2

```
Inserisci la tua scelta (1-3): 2

=== MODALITÀ VULNERABILE ATTIVATA ===
*** ATTENZIONE: NESSUN CONTROLLO DI INPUT ***
Inserire più di 10 elementi causerà BUFFER OVERFLOW!

Quanti interi vuoi inserire? █
```

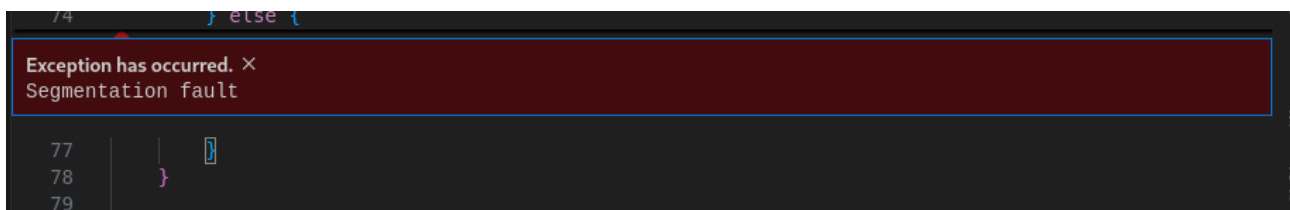
Decido di inserire un numero più alto per tentare un overflow massiccio : 10 mila interi

```
Quanti interi vuoi inserire? 10000█
```

```
Inserire 10000 interi:
[1]:█
```

Inseriti i 10000 interi e dato l'invio, il programma inizia a stampare i risultati e poco dopo si verificherà il **Segmentation Fault**:



A screenshot of a terminal window with a dark background. A red error message box is displayed in the center, containing the text "Exception has occurred. X" and "Segmentation fault". Below the error message, a portion of a C++ code file is visible, showing line numbers 74, 77, 78, and 79. Line 74 contains a closing curly brace and the text "else {". Line 77 contains a closing curly brace. Line 78 contains a closing curly brace. Line 79 contains a closing curly brace.

## 5. Analisi dei Risultati

### Comprensione del Comportamento del Sistema con Overflow Contenuto

Il fatto che il programma non sia crashato immediatamente con **100 elementi** è dovuto alle **protezioni moderne implementate** nei sistemi operativi e nei compilatori attuali. Queste protezioni includono **stack canaries**, **ASLR (Address Space Layout Randomization)** e altre tecniche che **mitigano gli effetti dei buffer overflow**, pur non eliminando completamente la vulnerabilità.

### Manifestazione della Corruzione

**La presenza di valori corrotti nell'output ( -9984) dimostra che il buffer overflow si è verificato effettivamente.** Questi valori rappresentano dati casuali presenti in memoria nelle locazioni adiacenti all'array, che sono stati interpretati come numeri interi dal programma durante le operazioni di lettura e ordinamento.

### Comprensione del Comportamento del Sistema con Overflow Massiccio

L'utilizzo di **10.000 elementi** ha superato la capacità delle protezioni di sistema, causando un accesso a memoria sufficientemente illegale da attivare la terminazione forzata del processo. Questo dimostra che, nonostante le protezioni moderne, le vulnerabilità di buffer overflow rimangono pericolose e possono ancora causare crash di sistema.

## 6. Conclusioni

### Obiettivi dell'Esercizio

L'esercizio ha raggiunto tutti gli obiettivi prefissati: è stata dimostrata la trasformazione di un programma sicuro in uno vulnerabile attraverso una modifica mirata, è stato ottenuto un **segmentation fault effettivo**, e sono stati implementati controlli di sicurezza efficaci. Il sistema di menu permette di **confrontare direttamente i due comportamenti**.

### Importanza della Validazione dell'Input

L'esperimento evidenzia come la **manca di controlli sull'input utente** possa compromettere completamente la sicurezza di un programma. Una singola omissione nella

validazione può trasformare un'applicazione affidabile in un vettore di vulnerabilità potenzialmente sfruttabile da attaccanti.

### **Applicazioni Pratiche**

Le competenze acquisite attraverso questo esercizio sono direttamente applicabili allo sviluppo sicuro di software, alla revisione del codice per identificare vulnerabilità simili, e alla comprensione delle tecniche utilizzate in ambito di sicurezza informatica per identificare e sfruttare questo tipo di problematiche.

### **Lezione Principale**

La modifica apparentemente minima del programma originale ha dimostrato come piccole modifiche architetturali possano avere conseguenze di sicurezza significative. Questo sottolinea l'importanza della sicurezza del software, dove ogni input utente deve essere considerato potenzialmente pericoloso e validato di conseguenza.