Esercizio S7_L3_System_Exploit_BOF

Consegna

https://drive.google.com/file/d/1nEM_FV5zFHj4hw9_Ya1PUP_xf5bLGy0I/view

Leggete attentamente il programma in allegato.
Viene richiesto di:
● Descrivere il funzionamento del programma prima dell'esecuzione.
● Riprodurre ed eseguire il programma nel laboratorio - le vostre ipotesi sul funzionamento erano corrette?
● Modificare il programma affinché si verifichi un errore di segmentazione.

Suggerimento: Ricordate che un BOF sfrutta una vulnerabilità nel codice relativo alla mancanza di controllo dell'input utente rispetto alla capienza del vettore di destinazione.

Concentratevi quindi per trovare la soluzione nel punto dove l'utente può inserire valori in input, e modificate il programma in modo tale che l'utente riesca ad inserire più valori di quelli previsti.

Bonus

Inserire controlli di input
Creare un menù per far decidere all'utente se avere il programma che va in errore oppure quello corretto

Svolgimento

Per comodità riportiamo qui sotto il codice presente nel link in allegato:

```
#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;


}
```

ANALISI CODICE

Analisi Statica

Analizzando il programma possiamo vedere che è stato sviluppato in c.
Come prima cosa il programmatore procede ad importare la libreria stdio.h.

Viene poi creata una funzione main che conterrà tutto il restante algoritmo.

la funzione inizia con una serie di variabili:

- vector, un'array contenente 10 interi
- i, j, k: solitamente variabili contatore utilizzate nei cicli for
- swap_var

Il programma chiede poi in input all'utente di inserire 10 numeri interi:

Una volta inseriti, parte un ciclo for dove il contatore i viene settato a 0 e per i < di 10 il contatore viene **incrementato di 1**.
Il ciclo prevede l'attribuzione del valore i+1 alla variabile c, stampa poi una stringa [%d] dove il **placeholder %d** prenderà il valore di **c**. Viene quindi fatto immettere dall'utente un numero decimale intero per ogni ciclo for; ogni valore viene registrato all'interno dell'array vector.

Dopo l'inserimento dei valori da parte dell'utente, il programma procede con la stampa del vettore così come è stato immesso.

Per questa operazione viene avviato un nuovo ciclo **for** in cui il contatore **i** viene **inizializzato a 0** e, fintanto che i < 10, il ciclo prosegue **incrementando di 1 ad ogni iterazione**.

All'interno del ciclo viene dichiarata una variabile **t** alla quale viene attribuito il valore i+1.
Successivamente viene stampata la stringa [%d]: %d, dove:

- il **primo %d** viene sostituito con la variabile **t**, che rappresenta la posizione dell'elemento (partendo da 1 anziché da 0);

- il **secondo %d** viene sostituito con il valore effettivo contenuto in vector[i].

In questo modo l'utente può visualizzare a schermo l'intero vettore nell'ordine esatto in cui lo ha inserito.

Una volta mostrato il vettore non ordinato, il programma passa all'ordinamento dei valori contenuti nell'array.

Questa fase viene implementata mediante due cicli **for** annidati:

- Il primo ciclo, con contatore **j, parte da 0** e si ripete fino a j < 10 - 1. Esso rappresenta il numero di passaggi necessari per portare gradualmente i valori più grandi verso la fine del vettore.

- Il secondo ciclo, con contatore k, **parte da 0** e si ripete fino a k < 10 - j - 1. Esso ha il compito di confrontare le coppie di elementi adiacenti: vector[k] e vector[k+1].

All'interno di questo secondo ciclo è presente una condizione **if** che verifica se l'elemento in posizione k è maggiore di quello in posizione k+1.
In caso positivo, i due valori vengono scambiati tra loro utilizzando la variabile **swap_var**.

Grazie a questa logica, ad ogni passaggio ***l'elemento più grande "scivola" verso destra, fino a posizionarsi correttamente in fondo al vettore***. Dopo un numero sufficiente di iterazioni, l'intero array risulta ordinato in ordine crescente.

Terminata la fase di ordinamento, il programma si occupa di stampare il vettore ordinato.

Per farlo utilizza un ultimo ciclo **for** in cui il contatore **j parte da 0** e si incrementa fino a j < 10.

Ad ogni iterazione viene dichiarata la variabile **g** con valore j+1.
Successivamente, tramite la stringa [%d]: %d, vengono stampati:

- la posizione dell'elemento, rappresentata da g;
- il valore ordinato corrispondente, cioè vector[j].

L'utente può così visualizzare l'intero vettore, questa volta con i valori disposti in ordine crescente.

In sintesi:

il programma sviluppato in C ha lo scopo di richiedere all'utente l'inserimento di 10 numeri interi, mostrarli così come sono stati immessi e successivamente ordinarli in ordine crescente.

La struttura prevede quindi tre fasi principali:

1. **Input**: acquisizione dei valori da tastiera e memorizzazione all'interno di un array.
2. **Output intermedio**: stampa del vettore non ordinato per permettere all'utente di verificare i dati inseriti.
3. **Elaborazione e output finale**: ordinamento del vettore tramite confronti e scambi successivi, seguito dalla stampa del vettore ordinato.

Analisi Dinamica

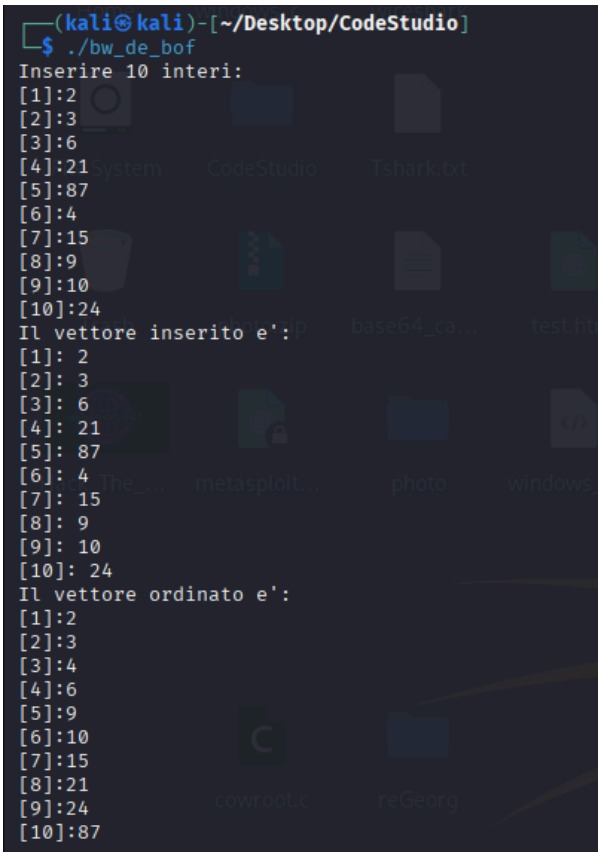
Per analizzare il comportamento del programma in maniera dinamica ho poi compilato il programma tramite il seguente comando:

gcc -Wall -Wextra -g BW_D3_BOF.c -o bw_de_bof

I due parametri utilizzati hanno le seguenti funzioni:

-Wall
Attiva il set "base" di avvisi del compilatore, pensato per segnalare i **warning più comuni** con poco rumore come variabili/non usate, funzioni non usate, parentesi/precedenze sospette, inizializzatori mancanti nei struct, uso potenzialmente non inizializzato (quando rilevabile).

-Wextra
Aggiunge **avvisi aggiuntivi** più "pignoli" che -Wall non include.

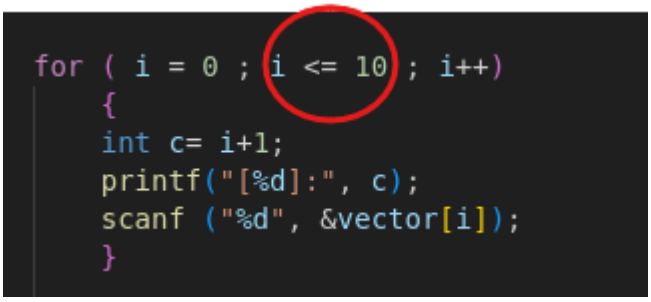


Il programma svolge esattamente quanto avevo preventivato confermando quanto spiegato in precedenza.

Buffer Overflow

Il prossimo obiettivo della consegna consiste nel far raggiungere al programma lo stato di buffer overflow.

Per fare ciò è stato sufficiente variare l'iterazione del primo ciclo for per ripetere l'inserimento dei valori da parte dell'utente una volta in più.



Il problema generato sta nel fatto che il programma proverà poi, durante il primo **ciclo for**, a scrivere l'undicesimo valore nello spazio di memoria già occupato dal decimo valore.

Procedo dunque a compilare il programma tramite il seguente comando:

gcc -std=c11 -Wall -Wextra -Wpedantic -O0 -g -fsanitize=address,undefined BW_D3_BOF_test.c -o bw_de_bof_test

In questo caso sono stati aggiunti i parametri:

-Wpedantic
Avvisa su tutto ciò che **non è conforme strettamente allo standard C**

-O0
Nessuna ottimizzazione. Il binario resta più vicino al codice che è stato scritto.

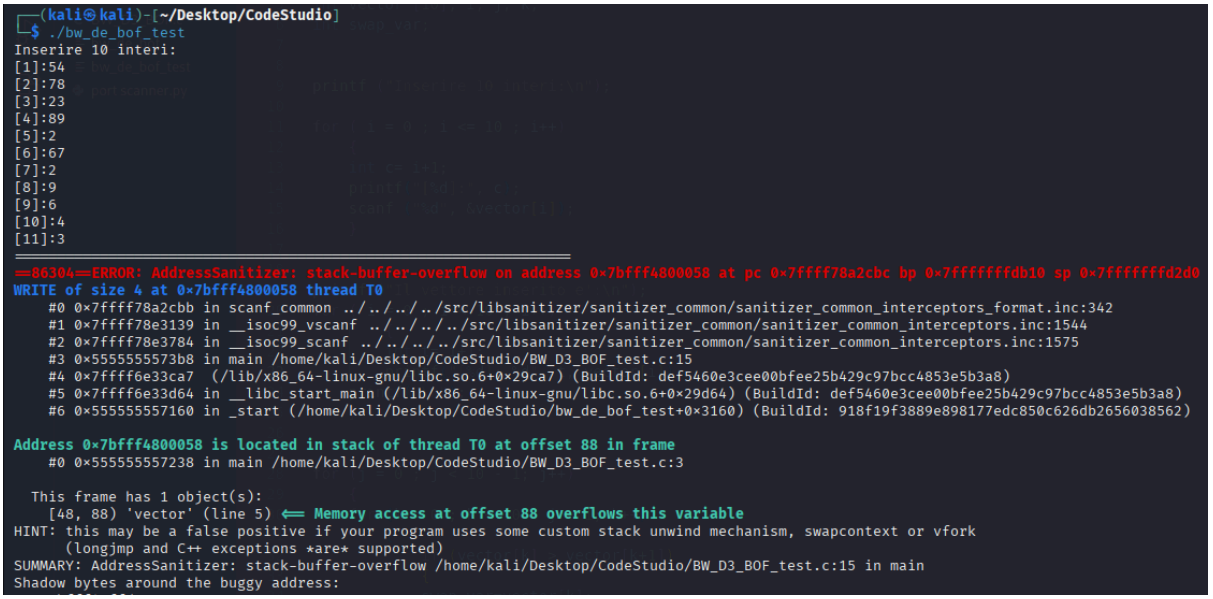
-g
Inserisce le **informazioni di debug** (simboli, mapping file:riga). Serve a mostrare **la riga esatta** del problema.

-fsanitize=address,undefined
Attiva due sanitizzatori a runtime:

- **AddressSanitizer (ASan)**: intercetta **stack/heap buffer overflow**, use-after-free, double free, overflow su VLA, ecc.; stampa un **report dettagliato** e normalmente **aborta**.

- **UndefinedBehaviorSanitizer (UBSan)**: segnala **UB "logici"** (es. overflow su interi **signed**, shift fuori range, divisione per zero su interi, indici fuori range su VLA, allineamenti invalidi, ecc.).

Una volta lanciato il programma notiamo per l'appunto che si presenta un problema di buffer overflow: il ciclo di input consente di inserire più di 10 valori e, all'**11° inserimento**, scanf tenta di scrivere in vector[10] (oltre i limiti di vector[0..9]). L'esecuzione viene intercettata da **AddressSanitizer** che segnala uno **stack-buffer-overflow** (*WRITE of size 4*) sulla variabile vector in main (riga incriminata), evidenziando un **off-by-one.**



Segmentation fault

Per generare il *Segmentation fault* ho modificato il programma affinché chiedesse all'utente quanti interi inserire (n), lasciando però **l'array fisso a 10 celle**. Se n > 10, il ciclo di input scrive in vector[10] e oltre (accesso fuori dai limiti): questo causa un **buffer overflow** sullo stack.

Le successive stampe e l'ordinamento effettuano ulteriori accessi illegali; ***quando l'overflow raggiunge un'area di memoria non mappata, il processo termina con Segmentation fault.***

Il codice modificato è il seguente:

```c
#include <stdio.h>

#define CAP 10  // capienza reale del vettore

int main(void) {
    int vector[CAP], i, j, k;
    int swap_var;
    int n;  // numero di elementi richiesti all'utente (non limitato a CAP)

    // --- INPUT: numero di elementi ---
    printf("Quanti interi vuoi inserire? ");
    if (scanf("%d", &n) != 1) {
        return 1;
    }

    // --- INPUT: valori ---
    // VULNERABILITÀ: se n > CAP si scrive fuori dai limiti di vector (BOF)
    printf("Inserire %d interi:\n", n);
    for (i = 0; i < n; i++) {
        int c = i + 1;
        printf("[%d]:", c);
        scanf("%d", &vector[i]);  // <-- write OOB quando i >= CAP
    }

    // --- STAMPA: vettore come inserito ---
    // Anche qui si può leggere fuori dai limiti se n > CAP
    printf("Il vettore inserito e':\n");
    for (i = 0; i < n; i++) {
        int t = i + 1;
        printf("[%d]: %d\n", t, vector[i]);  // <-- read OOB quando i >= CAP
    }

    // --- ORDINAMENTO sui "n" inseriti ---
    // Accessi OOB anche qui se n > CAP
    for (j = 0; j < n - 1; j++) {
        for (k = 0; k < n - j - 1; k++) {
            if (vector[k] > vector[k + 1]) {
                swap_var = vector[k];
                vector[k] = vector[k + 1];
                vector[k + 1] = swap_var;
            }
        }
    }

    // --- STAMPA: vettore ordinato ---
    printf("Il vettore ordinato e':\n");
    for (j = 0; j < n; j++) {
        int g = j + 1;
        printf("[%d]: %d\n", g, vector[j]);  // <-- read OOB quando j >= CAP
    }

    return 0;
}
```

Ho poi compilato il programma tramite gcc:

```
gcc -O0 -g -Wall -Wextra -fno-stack-protector -no-pie BW_D3_BOF_test.c -o bw_de_bof_test
```

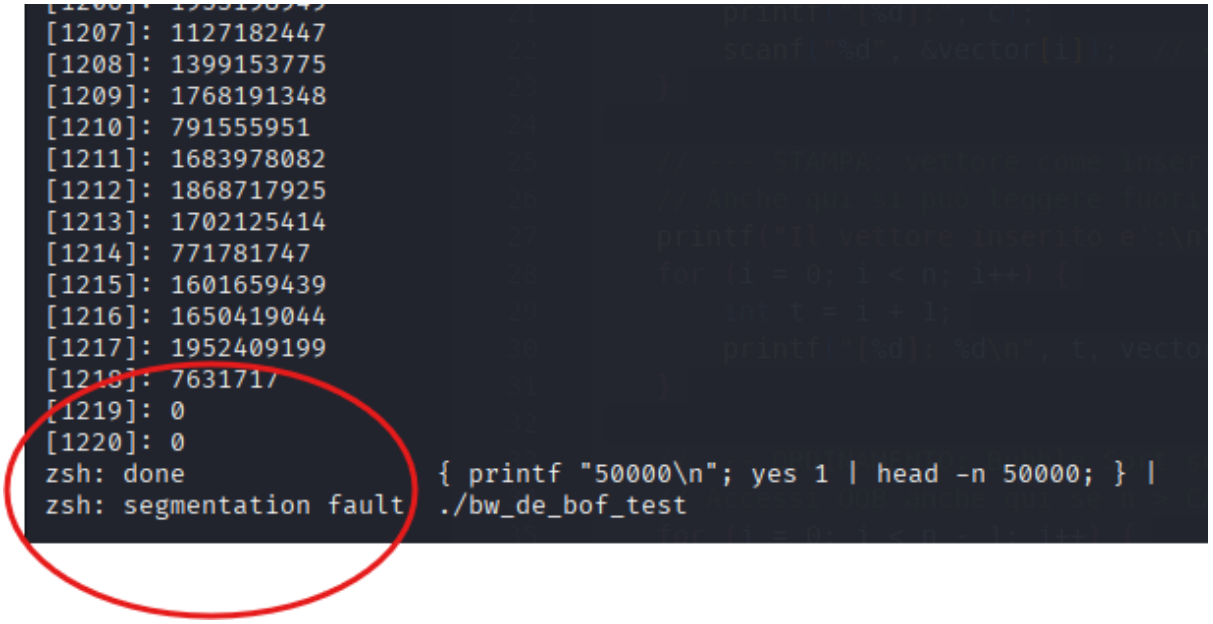
Ed ho infine disattivato l'ASLR, una mitigazione che **randomizza gli indirizzi** di memoria di processi ad ogni esecuzione allo scopo di **rendere imprevedibili** gli indirizzi; viene utilizzato affinché un exploit (es. BOF) che punta a un offset fisso tenda a fallire.

```
sudo sysctl -w kernel.randomize_va_space=0 # disabilita ASLR
```

Fatto ciò ho avviato il seguente comando affinchè fornisse in automatico i 50000 numeri una volta avviato il programma.

```
{ printf "50000\n"; yes 1 | head -n 50000; } | ./bw_de_bof_test
```

Come risultato otteniamo che dopo aver tentato di riordinare 1221 numeri il programma va in **segmentation fault**:



Il numero di elementi dopo il quale avviene il crash ***non è fisso****: dipende dal layout dello stack, dalle flag di compilazione, dall'ASLR, ecc.*

Il segfault si verifica quando le scritture oltre l'array raggiungono una ***zona di memoria non mappata****.*

UNIFICAZIONE DELLE DUE VERSIONI

Come ultimo compito ho unito il programma originale e quello vulnerabile in un unico eseguibile; all'avvio un menù permette all'utente di scegliere se eseguire la **versione originale** (10 input fissi) oppure la **versione vulnerabile** (numero di input arbitrario, che può causare BOF/segfault).

Implementare controlli di Input

Nel programma **finale con menù** ho poi implementato una funzione di lettura **robusta** degli interi, basata su così da validare *davvero* ciò che l'utente digita ed evitare i classici problemi di scanf.

La funzione è stata denominata leggi_int_sicuro e contiene:

1. **Lettura linea intera** con fgets() in un buffer a dimensione fissa → niente overflow del buffer di input.
2. **Parsing numerico** con strtol():
    - salta spazi iniziali;
    - controlla che **almeno una cifra** sia stata letta (altrimenti input non valido);
    - verifica **overflow/underflow** con errno == ERANGE;
3. **Rifiuta "spazzatura"** dopo il numero: accetta solo spazi/newline; se trova caratteri extra (es. 12abc) richiede di **riprovarci**.
4. **Vincolo di range**: accetta il valore solo se cade nell'intervallo [min..max] passato come parametro (messaggio d'errore chiaro, poi ripete la richiesta).
5. **Loop di ripetizione**: finché l'utente non inserisce un intero valido nel range, **ripropone il prompt**.

I controlli di input sono stati appplicati a:

- **Menù iniziale**: scelta vincolata a [1..2] (solo "Vulnerabile" o "Originale").
- **Modalità Originale (sicura)**: richiede **esattamente 10** interi; **ogni valore** è validato (intero con segno, niente caratteri extra, nessun overflow). Non si chiede n.

- **Modalità Vulnerabile**:
    - n viene letto in modo robusto e vincolato a [1..INT_MAX] (quindi niente input non numerici o fuori range dell'int);
    - i **valori** vengono letti e validati come sopra;
    - **non limitiamo** n a CAP=10 **di proposito** per conservare la vulnerabilità: se n > 10, i cicli scrivono/leggono oltre vector[10] → **BOF** (e possibile segfault).
      In pratica, l'input è "pulito", ma la **mancanza di controllo sulla capienza** resta intenzionale per la dimostrazione del bug.

Criticità scanf("%d", &x)**?**

- scanf gestisce male input con caratteri extra e lascia **residui nel buffer** creando comportamenti imprevisti nei prompt successivi.

In sintesi:

- E' stato **validato il contenuto** (numero, formato, range) e **normalizzato il flusso di input** per entrambe le modalità.
- Nella modalità **Originale** questo elimina errori di input.
- Nella modalità **Vulnerabile**, i controlli rendono l'esperimento riproducibile (niente input sporchi), ma **non sanano** la vulnerabilità strutturale (mismatch n vs CAP), che resta intenzionalmente per mostrare il **buffer overflow**.

Ho poi nuovamente compilato il programma:

```
gcc -std=c11 -Wall -Wextra -Wpedantic -O0 -g \
   -fsanitize=address,undefined -fno-omit-frame-pointer \
   BW_D3_BOF_finale.c -o bw_d3_bof_finale
```

Ed ho infine ripristinato l'ASLR al termine dei test:

```
sudo sysctl -w kernel.randomize_va_space=2
```

Il file finale può essere trovato nella stessa directory GitHub dove è presente questo report.

Conclusioni

Obiettivi centrati: *Ho analizzato il programma originale (10 interi, stampa e ordinamento), ne ho verificato il comportamento in esecuzione e ho individuato il punto critico: l'assenza di un vincolo tra* ***input dell'utente*** *e* ***capienza dell'array****.*

Dimostrazione del BOF: *Ho mostrato un* ***buffer overflow sullo stack*** *sia con l'errore off-by-one sia con la variante che usa n controllato dall'utente.*

Dal BOF al segfault: *Nella versione vulnerabile (array vector[10] ma n arbitrario) gli accessi oltre i limiti provocano, al crescere di n, un possibile* ***Segmentation fault*** *quando si tocca memoria non mappata.*

- **Bonus implementato.** Ho realizzato un **unico eseguibile con menù ed ho implementato una funzione di validazione degli input**:
  - **Originale (sicura)** — 10 input fissi con **validazione robusta** degli interi (fgets+strtol, controllo del formato e del range).
  - **Vulnerabile** — n arbitrario letto in modo robusto ma **non limitato alla capienza**: la vulnerabilità rimane intenzionalmente per dimostrare BOF/segfault.
- **Lezioni chiave.**
  - Il **BOF è la causa**, il **segfault è un possibile effetto**.
  - I **sanitizzatori** sono essenziali per evidenziare subito accessi fuori limite e localizzare il bug.
  - Le **mitigazioni** (ASLR, canary, PIE/NX) influenzano osservabilità e sfruttabilità; vanno disattivate solo in laboratorio e **ripristinate** a fine test.

In sintesi, *l'esercizio dimostra come un semplice mismatch tra* ***dimensione del buffer*** *e* ***quantità di input*** *conduca a un* ***buffer overflow*** *e, in condizioni realistiche, a un* ***segmentation fault****. Il menù consente di confrontare in modo immediato la versione corretta con quella vulnerabile, mentre i controlli di input mostrano come rendere il programma robusto senza alterarne la logica.*