System Exploit - Buffer Overflow

Introduzione e contesto:

L'obbiettivo di questa esercitazione è descrivere il funzionamento del programma **prima** dell'esecuzione. Riprodurre ed eseguire il programma nel laboratorio e modificare il programma affinché' si verifichi un errore di segmentazione.

Buffer Overflow:

è un difetto di programmazione (una vulnerabilità) che si verifica quando un programma tenta di scrivere dati in **un'area di memoria temporanea** (chiamata buffer) che è più piccola della quantità di dati che viene effettivamente scritta.

La maggior parte dei Buffer Overflow avvengono nello **Stack** (la pila di memoria utilizzata per le chiamate di funzione).

Quando una funzione viene chiamata, il sistema operativo alloca un'area sullo stack che contiene le variabili locali della funzione (i buffer) e i dati di controllo della funzione, in particolare l'**Indirizzo di Ritorno**. L'Indirizzo di Ritorno è fondamentale: è l'istruzione che il programma deve eseguire **dopo** che la funzione corrente ha finito il suo lavoro.

Un attacco di Buffer Overflow sfrutta il fatto che i dati e le informazioni di controllo della funzione sono archiviati in settori di memoria **adiacenti**.

Il difetto nasce quando il codice non controlla la lunghezza dei dati in input prima di copiarli nel buffer.

Ad esempio, se l'utente fornisce un input di 30 caratteri { [più dei 16 dichiarati nel buffer (ad esempio "char buffer[16];") } i primi 16 caratteri andranno nel buffer, i restanti 14 traboccheranno dal buffer. I dati che traboccano dal buffer non vanno nel "vuoto", essi sovrascrivono i dati adiacenti sullo stack, che includono l'indirizzo di ritorno della funzione.

Cosa fa l'attaccante?

Costruisce strategicamente un input composto da due parti:

- **Shellcode** → ovvero la parte di codice malevolo. I dati iniziali che traboccano sono un piccolo programma che l'attaccante vuole far eseguire (ad esempio per eseguire una shell di comando).
- **Nuovo indirizzo di ritorno** → l'ultima parte dei dati in overflow rappresenta l'indirizzo esatto dove si trova il codice maligno.

Quando la funzione vulnerabile termina, invece di tornare all'istruzione legittima (Indirizzo di ritorno originale), il programma legge l'indirizzo di ritorno modificato dall'attaccante.

Il risultato è che il flusso di esecuzione del programma viene dirottato e il sistema esegue il codice Maligno (Shellcode) iniettato dall'attaccante.**Codice di partenza (←)**

Il seguente codice è un programma completo il cui scopo è leggere 10 numeri interi dall'utente, ordinarli in modo crescente e poi visualizzare il risultato.

Utilizza l'algoritmo di **Bubble Sort** per l'ordinamento:

lo scopo del Bubble Sort è quello di spostare ripetutamente l'elemento più grande non ancora ordinato alla sua posizione finale, facendo "galleggiare" (**bubble up**) i numeri grandi verso la fine dell'array tramite scambi di elementi adiacenti.

```
#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;


}
```

Il codice modificato mantiene la logica di ordinamento del codice di partenza, ma:

- ❖ introduce un **menu** e due percorsi di esecuzione: uno corretto e uno con errore forzato:

```
printf("\n-----Menu-----\n");
printf("\nCome vuoi eseguire il programma?\n"
printf("1) Con errore di segmentazione\n");
printf("2) Senza errore\n");
printf("----------------\n");
printf("Inserisci scelta: ");

if (scanf("%d", &scelta) != 1) {
    printf("Errore di input. Esco.\n");
    return 1;
}
while (getchar() != '\n');

switch(scelta) {
    case 1:
        Errore();
        break;
    case 2:
        Corretto();
        break;
    default:
        printf("SCELTA NON VALIDA!\n");
        return 0;
}
return 0;
```

Il programma si apre con la visualizzazione di un **Menu** che chiede all'utente come desidera eseguire il programma:

1. **"Con errore di segmentazione"** : Questo percorso serve esplicitamente allo scopo dell'esercitazione per innescare un crash.
2. **"Senza errore"** : Questo rappresenta il percorso robusto e corretto, tipico di un'applicazione ben progettata.

Dopo aver visualizzato il menu, il programma tenta di leggere la **scelta** dell'utente tramite scanf("%d", &scelta).

Segue un cruciale **controllo dell'input**:

- Se la funzione scanf non riesce a leggere un intero (cioè, se (scanf("%d", &scelta) != 1)), significa che l'utente ha inserito un carattere non numerico o un errore di input. In questo caso, il programma stampa un messaggio di errore ed esce immediatamente (return 1).
- Subito dopo il controllo di scanf, c'è l'istruzione while (getchar() != '\n');. Questa riga ha lo scopo di **pulire il buffer di input** eliminando qualsiasi carattere residuo, specialmente il newline (\n) lasciato dall'immissione dell'utente, assicurando che non interferisca con le letture successive (sebbene in questo punto il programma esca in caso di errore).

Infine, il costrutto **switch** direziona il flusso di esecuzione in base al valore della variabile scelta:

- **Caso 1**: Se l'utente sceglie '1', viene chiamata la funzione **Errore()**. Questa è la funzione che sfrutta intenzionalmente la vulnerabilità per forzare l'Errore di Segmentazione.
- **Caso 2**: Se l'utente sceglie '2', viene chiamata la funzione **Corretto()**. Questa funzione esegue l'ordinamento in modo sicuro e robusto.
- **Default**: Se l'utente inserisce qualsiasi altro numero, la scelta è considerata non valida, e il programma termina con un messaggio appropriato.

Analisi del codice

Soffermiamoci adesso su un analisi più approfondita delle funzioni che costruiscono il nostro codice.

Corretto():

La funzione void Corretto() esegue la logica di base del programma in modo sicuro e controllato. Essa è stata concepita per rappresentare il **codice che ci si aspetterebbe in un'applicazione ben progettata**.

All'inizio, la funzione **dichiara** e **alloca** l'array di interi: int vector[DIM];. La dimensione di questo vettore è definita dalla costante DIM (che presumibilmente vale 10, come indicato nella documentazione generale).

```c
/* --- FUNZIONE CORRETTA --- */
void Corretto(){
    int vector[DIM];
    printf("\n--- 2. Esecuzione Corretta ---\n");
    leggiVettore(vector, DIM);
    printf("\n--- Vettore Inserito ---\n");
    stampaVettore(vector, DIM);
    bubbleSort(vector, DIM);
    printf("\n--- Vettore Ordinato ---\n");
    stampaVettore(vector, DIM);
}
```

Il flusso di esecuzione è il seguente:

1. **Input Sicuro**: Viene chiamata la funzione **leggiVettore(vector, DIM);**. Questa è l'implementazione robusta che legge i 10 numeri interi dall'utente, garantendo che l'input sia gestito correttamente e che vengano accettati solo numeri validi, grazie al **controllo degli errori e alla pulizia del buffer**. Questo assicura che il programma gestisca in modo elegante l'input non valido o inatteso senza bloccarsi.
2. **Visualizzazione**: Dopo l'inserimento, il programma stampa il vettore non ordinato chiamando **stampaVettore(vector, DIM);**.
3. **Ordinamento**: La funzione **bubbleSort(vector, DIM);** viene eseguita per ordinare gli elementi del vettore in modo crescente.
4. **Output Finale**: Infine, il programma stampa il vettore ordinato chiamando nuovamente **stampaVettore(vector, DIM)**

Errore():

```c
/* --- FUNZIONE ERRORE--- */
void Errore(){
    int vector[DIM];
    printf("\n--- 1. Esecuzione con Errore di Segmentazione ---\n");


    leggiVettore_VULNERABILE(vector, DIM);

    printf("\n--- Vettore Inserito ---\n");
    stampaVettore(vector, DIM);
    bubbleSort(vector, DIM);
    printf("\n--- Vettore Ordinato ---\n");
    stampaVettore(vector, DIM);
}
```

Strutturalmente, questa funzione segue il percorso di esecuzione della funzione Corretto(): alloca il vettore, stampa un'intestazione, legge i dati, li visualizza, li ordina e li visualizza nuovamente.

La differenza cruciale risiede nell'unica riga di input:

- Al posto della funzione sicura leggiVettore(), viene chiamata **leggiVettore_VULNERABILE(vector, DIM);**.
- Questa funzione vulnerabile è intenzionalmente programmata per innescare una **violazione di accesso alla memoria** (tentativo di scrittura all'indirizzo NULL, 0x0).

Di conseguenza, se l'utente fornisce l'input che attiva la vulnerabilità , il programma viene terminato forzatamente con il SegFault , interrompendo il normale flusso di esecuzione (ordinamento e stampe successive)

LeggiVettore:

La funzione leggiVettore() è il **percorso robusto** per l'acquisizione dell'input, garantendo che vengano letti solo numeri interi validi per l'ordinamento.

All'interno di un ciclo che scorre per il numero di elementi (dim) da inserire, la funzione tenta di leggere direttamente un intero usando **scanf("%d", &v[i])**. Il valore di ritorno di scanf viene salvato in input_successo: se la lettura va a buon fine, il valore è **1**; altrimenti (se l'utente inserisce caratteri non numerici), la lettura fallisce.

Se scanf fallisce, il blocco if di gestione degli errori entra in azione:

1. Stampa un messaggio d'errore.
2. Esegue un ciclo while (getchar() != '\n') per **pulire il buffer di input**. Questa operazione fondamentale elimina tutti i caratteri errati lasciati dall'utente.

Infine, il ciclo do-while esterno si ripete finché input_successo non è pari a **1**, forzando l'utente a ripetere l'inserimento finché non fornisce un intero valido. Questa implementazione dimostra un codice sicuro e difensivo

```c
void leggiVettore(int v[], int dim) {
    int i;
    int input_successo; // Flag per il controllo di scanf

    printf("\nInserire %d interi per l'ordinamento:\n", dim);

    for (i = 0; i < dim; i++) {
        do {
            printf("Elemento v[%d]: ", i);

            // 1. Tenta di leggere l'intero
            input_successo = scanf("%d", &v[i]);

            if (input_successo != 1) {
                // 2. Se scanf fallisce (input non numerico)
                printf("!!! Errore: inserisci un numero valido !!!\n");

                // 3. Pulisce il buffer di input da tutti i caratteri errati
                while (getchar() != '\n');

            }
        } while (input_successo != 1); // Ripete finché l'input non è un intero valido
    }
    printf("\nVettore letto con successo.\n");
}
```

LeggiVettore_VULNERABILE:

Questa funzione, **leggiVettore_VULNERABILE()**, è stata creata appositamente per dimostrare come un errore nel codice possa portare al **crash del programma**, ed è il cuore dell'esperimento che abbiamo svolto.

Invece di leggere subito un numero intero, usiamo **fgets** per leggere l'input come una **stringa** di testo. fgets è una funzione che prende i caratteri digitati dall'utente e li salva in una variabile, e il fatto che possiamo specificare la dimensione massima la rende intrinsecamente più sicura contro i Buffer Overflow rispetto ad altre funzioni di input stringa. Una volta letta la stringa e rimosso l'eccesso di newline (\n), la convertiamo in un numero intero con **atoi**.

Qui arriva il momento critico, il **punto di innesco** della vulnerabilità: Abbiamo impostato una condizione (if (input_value > 1000)) che, se soddisfatta (cioè, se si inserisce un numero maggiore di 1000), esegue la parte di codice che garantisce il crash.

Questa sezione include un puntatore, ***crash_ptr**, a cui viene intenzionalmente assegnato l'indirizzo **NULL**, che corrisponde all'indirizzo di memoria **0x0**. Quando il codice tenta di scrivere un valore in quell'indirizzo (*crash_ptr = 42;), il sistema operativo lo riconosce immediatamente come una **violazione di accesso a memoria**. Dato che quell'area di memoria è riservata, il sistema interviene per proteggere la sua integrità e **termina forzatamente** l'applicazione, provocando il SegFault.

In pratica, abbiamo creato un interruttore che, se attivato dall'utente, dimostra l'impatto distruttivo di un accesso illegale alla memoria, un principio fondamentale che si trova anche dietro ai più complessi attacchi di Buffer Overflow.

```c
void leggiVettore_VULNERABILE(int v[], int dim) {
    int i;
    char input_str[100];
    int input_value;

    printf("\nInserire %d interi per l'ordinamento:\n", dim);

    for (i = 0; i < dim; i++){
        printf("Elemento v[%d]: ", i);

        // 1. Usa FGETS per leggere la stringa
        if (fgets(input_str, sizeof(input_str), stdin) == NULL) {
            printf("Errore di input. Esco.\n");
            return;
        }

        // Rimuove il newline ('\n') da fgets
        input_str[strcspn(input_str, "\n")] = 0;

        // 2. Converte la stringa in intero
        input_value = atoi(input_str);

        // **VULNERABILITÀ / TRIGGER**
        if (input_value > 1000) {
            // Salviamo il valore
            v[i] = input_value;

            // **LA LINEA CHE GARANTISCE IL CRASH**
            // Accesso a memoria 0x0 -> SegFault GARANTITO.
            int *crash_ptr = NULL;
            *crash_ptr = 42;
        }

        // 3. Se il valore è sicuro, lo salviamo
        v[i] = input_value;
    }
    printf("\nVettore letto con successo (il crash non è avvenuto).\n");
}
```

Dimostrazione pratica

Dimostrazione 1:

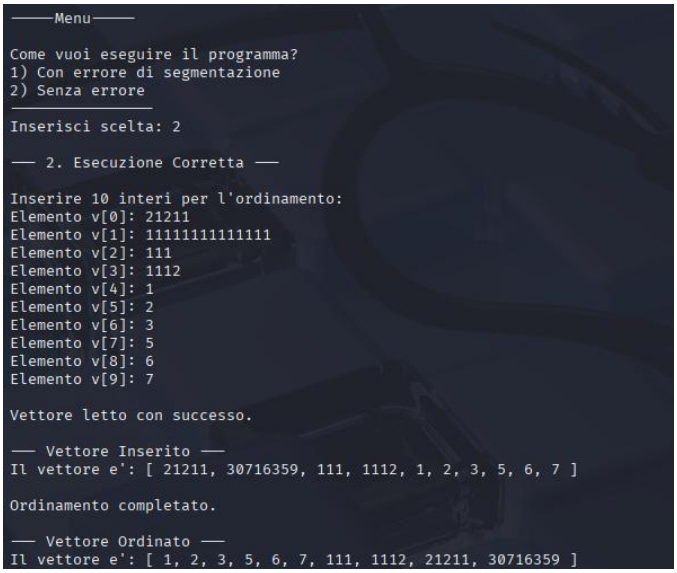
Dopo aver eseguito il programma (./a.out) da terminale, l'utente sceglie l'opzione **1) Con errore di segmentazione**. Questo avvia la funzione vulnerabile.

Quando il programma chiede l'input per il primo elemento (Elemento v[0]), l'utente inserisce intenzionalmente un numero estremamente grande: **2121212212121121212**.

Poiché questo input supera la soglia di controllo impostata nel codice, esso attiva la violazione di accesso alla memoria. Il sistema operativo rileva l'azione illegale e termina immediatamente il programma, restituendo l'errore **zsh: segmentation fault ./a.out**.

L'esecuzione dimostra quindi che il programma si è bloccato prima di poter continuare con le altre operazioni (come l'ordinamento), confermando l'innesco voluto del SegFault.

Dimostrazione 2:

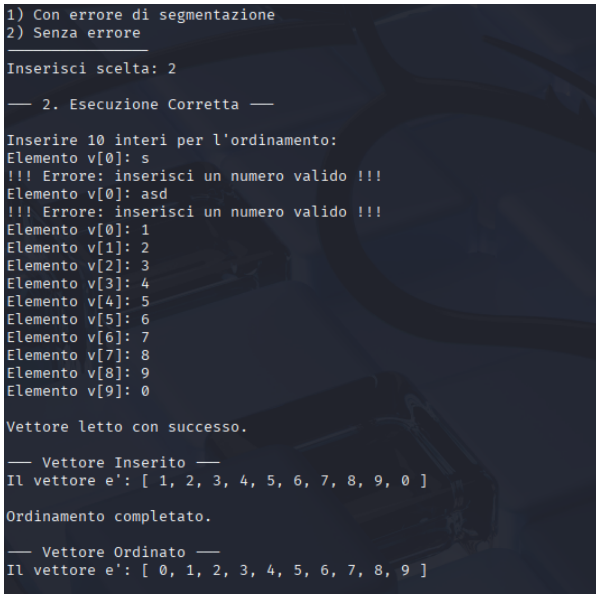


L'utente ha scelto l'opzione **2) Senza errore**, attivando la **Esecuzione Corretta** (la funzione Corretto()).

Durante l'inserimento, l'utente ha fornito numeri estremamente grandi, come **21211** e **111111111111**. Poiché il programma utilizza la funzione leggiVettore() (quella robusta), non si è verificato alcun errore di input. Tuttavia, la console mostra che il valore 111111111111 è stato memorizzato come **30716359**. Questo accade perché il numero originale **supera il limite massimo** che una variabile intera standard (int) può contenere, causando un **overflow numerico** e memorizzando il valore troncato risultante.

Nonostante la presenza di questi valori molto grandi o *overflowed*, il programma rimane **stabile**. Il Bubble Sort viene eseguito correttamente e il **Vettore Ordinato** finale mostra i numeri in sequenza crescente: [1, 2, 3, 5, 6, 7, 111, 1112, 21211, 30716359].

Dimostrazione 3:



La dimostrazione conclude con l'utente che sceglie l'opzione "2) Senza errore" dal menu, avviando il percorso di Esecuzione Corretta.

Durante la fase di inserimento dei 10 interi, il codice dimostra la sua robustezza:

- Per il primo elemento (v[0]), l'utente tenta due volte di inserire un input non numerico (s e asd).

- Il programma reagisce correttamente, stampando "!!! Errore: inserisci un numero valido !!!" in entrambi i tentativi e forzando l'utente a ripetere l'inserimento.

Dopo questi errori, l'utente inserisce con successo i 10 numeri, che vengono letti e il programma conferma: "Vettore letto con successo".

Infine, il programma visualizza il Vettore Inserito non ordinato: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]. Dopo l'ordinamento, viene mostrato il Vettore Ordinato in sequenza corretta: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. L'output finale è "Ordinamento completato".

Conclusioni:

L'evoluzione dal codice di partenza al codice custom non è solo un insieme di modifiche, ma una vera e propria trasformazione funzionale del programma. L'obiettivo primario non è più l'ordinamento in sé, ma l'illustrazione didattica di una vulnerabilità critica del software.

Il cambiamento più evidente è l'introduzione di un menu d'avvio.

Il codice originale procedeva in modo lineare: chiedeva 10 numeri, li ordinava e li mostrava. Il nuovo codice, invece, offre all'utente la possibilità di scegliere tra due percorsi:

1. Il Percorso Corretto() (Robusto): Rappresenta il codice che ci si aspetterebbe in un'applicazione ben progettata. La funzione chiave, leggiVettore(), è stata resa sicura e robusta grazie all'uso di **scanf** e all'implementazione di un meccanismo di controllo degli errori e pulizia del buffer. Questo assicura che il programma gestisca in modo elegante l'input non valido o inatteso senza bloccarsi.
2. Il Percorso Errore(): Questo percorso serve esplicitamente allo scopo dell'esercitazione. Qui risiede la vera modifica concettuale, che sfrutta la funzione leggiVettore_VULNERABILE().

Assegnando a un puntatore l'indirizzo NULL (0x0) e tentando poi di scriverci, il programma compie una violazione di accesso alla memoria. Il sistema operativo interviene immediatamente per proteggere la sua integrità, terminando forzatamente l'applicazione e generando il SegFault.

In conclusione, la trasformazione ha convertito un codice funzionale e didattico sull'ordinamento in uno strumento per osservare l'impatto distruttivo di un accesso illegale alla memoria, un principio fondamentale condiviso con il più complesso meccanismo degli attacchi Buffer Overflow.