

SYSTEM EXPLOIT BOF

Introduzione

La seguente relazione descrive l'attività svolta durante il terzo giorno di laboratorio dedicato all'esplorazione di vulnerabilità di tipo 'buffer overflow' (BOF) nei sistemi. L'esercizio prevede l'analisi e la modifica di un programma fornito per comprendere in dettaglio i meccanismi alla base di un potenziale exploit.

L'obiettivo iniziale è quello di analizzare il codice del programma per comprenderne il funzionamento, e successivamente modificare il codice in modo da causare un errore di segmentazione, inducendo un comportamento anomalo nel programma, per dimostrare come un buffer overflow possa portare a malfunzionamenti.

Inoltre, è compresa l'aggiunta di uno degli obiettivi bonus di inserire dei controlli di input prima dell'esecuzione del programma per garantirne la corretta esecuzione, di modo da evitare comportamenti inattesi o dannosi.

Analisi del codice

Il codice dato scrive un programma in C finalizzato a leggere un array di 10 numeri interi inseriti dall'utente. Il programma prima stampa i numeri inseriti sul terminale nell'ordine in cui sono stati immessi, li ordina in ordine crescente attraverso un algoritmo, e infine ristampa l'array ordinato.

Il codice si apre con la direttiva del processore che include le definizioni delle funzioni per l'input/output standard del programma, e la definizione della funzione principale del programma.

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

```
int main () {
```

Vengono successivamente dichiarati l'array da 10 numeri interi e le variabili di ciclo, che saranno successivamente usate all'interno di un ciclo 'for', insieme alla variabile di supporto per lo scambio di valori durante l'ordinamento delle stesse.

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

```
int swap_var;
```

Viene dunque chiesto all'utente di inserire 10 numeri interi, seguito da un blocco di codice che inizia un ciclo 'for' per leggere i valori immessi, salvandoli nell'array 'vector' e riportandoli sul terminale con un indice numerico da 1 a 10.

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

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

```
{
```

```
int c= i+1;
```

```
printf("[%d]:", c);
```

```
scanf ("%d", &vector[i]);
```

```
}
```

Al termine dell'inserimento, il blocco seguente di codice inizia un secondo ciclo 'for' per stampare i numeri nell'ordine in cui sono stati inseriti, insieme al relativo indice.

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

Una volta fatto questo, il programma inizia due cicli 'for', uno interno esterno ed uno interno, per ordinare gli input numerici inseriti in maniera crescente attraverso un algoritmo 'bubble sort'. Il ciclo 'for' interno confronta le coppie di elementi adiacenti da sinistra verso destra, scambiandoli se questi non sono in ordine crescente. In particolare, la riga 'if (vector[k] > vector[k+1])' verifica che l'elemento corrente sia maggiore del successivo, scambiando gli elementi con l'utilizzo della variabile temporanea 'swap_var' nel caso in cui la condizione sia vera.

```
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;  
        }  
    }  
}
```

L'ultimo blocco di codice stampa sul terminale gli elementi dell'array in ordine crescente, insieme all'indice corrispondente, prima di terminarsi.

```
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;  
}
```

Errore di segmentazione

Un errore di segmentazione è un tipo di errore che si verifica quando un programma tenta di accedere a una porzione di memoria che va oltre quella che gli è stata allocata. È un tipo di errore comune nei linguaggi come C, che offrono un controllo diretto sulla memoria. In particolare, quando un programma tenta di scrivere dati oltre ai limiti stabiliti da un array potrebbe accedere ad una parte di memoria non destinata a tale array (buffer overflow), portando a comportamenti imprevisti o vulnerabilità di sicurezza.

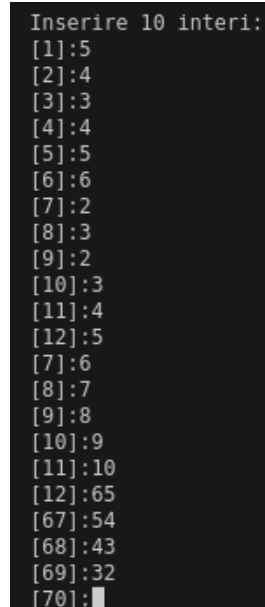
Partendo dal codice fornito, è possibile apportare una minuscola modifica che permette all'utente di inserire più valori di quelli previsti semplicemente eliminando la condizione di terminazione all'interno primo ciclo 'for' relativo all'inserimento di numeri interi dal parte dell'utente.

```
for (i = 0 ; ; i++)  
{  
    int t = i + 1;  
    printf("[%d]: %d", t, vector[i]);  
    printf("\n");  
}
```

In questo caso, mancando la condizione di terminazione ($i < 10$), il ciclo continuerà all'infinito cercando di accedere a zone di memoria che non gli sono state assegnate agli elementi dell'array anche oltre la dimensione massima di 10. Questo comportamento porta a un errore di segmentazione poiché il programma tenta di accedere a zone di memoria che non gli sono state assegnate.

Come vediamo da questo output, i valori degli indici risultano errati o casuali, proprio perché il ciclo 'for', non avendo una condizione di uscita specificata, continua a incrementare la variabile *i* fino a quando non causa un errore di segmentazione o finisce per accedere a una memoria non valida.

Dal momento che un programma del genere, che richiede l'inserimento di un numero infinito di input, risulterebbe improponibile, possiamo analizzare altri modi per provocare un errore di segmentazione in un programma che effettivamente si compie a termine.



```
Inserire 10 interi:  
[1]:5  
[2]:4  
[3]:3  
[4]:4  
[5]:5  
[6]:6  
[7]:2  
[8]:3  
[9]:2  
[10]:3  
[11]:4  
[12]:5  
[7]:6  
[8]:7  
[9]:8  
[10]:9  
[11]:10  
[12]:65  
[67]:54  
[68]:43  
[69]:32  
[70]:
```

Lavoriamo su una versione modificata del codice dato che presenta alcune differenze, anche migliorando l'esperienza dell'utente (vedere [Appendice 1](#) per il codice completo). Nello specifico, le modifiche apportate coprono:

1. L'inserimento dinamico di dati: il codice modificato permette all'utente di inserire anche meno di 10 numeri, dando la possibilità di fermarsi prima di raggiungere questo limite inserendo '-1'.
2. La richiesta di ulteriori input: se l'utente inserisce meno di 10 numeri, il programma chiede ulteriori input, continuando a consentire all'utente di aggiungere numeri finché non raggiunge il limite.
3. Un secondo ciclo per i nuovi input: introduce un ciclo 'while (1)' per l'inserimento di ulteriori numeri.

Quest'ultima modifica risulta di particolare importanza per il nostro caso di studio, in quanto la possibile rimozione del controllo sul limite ($i < 10$) nel ciclo interno rende il codice suscettibile a un errore di segmentazione nel caso l'utente continui a inserire numeri oltre il limite di 10.

```
while (i < 10) {  
    printf("Hai inserito solo %d numeri. Inserisci altri numeri:\n", i);  
    while (1) {  
        int temp;  
        scanf("%d", &temp);  
        if (temp == -1) {  
            break;  
        }  
        vector[i] = temp;  
        i++;  
    }  
}
```

In particolare, il ciclo esterno '`while (i < 10)`' continua finché 'i' è minore di 10, di modo che se l'utente inserisce meno di 10 numeri e decide poi di inserire più numeri in un secondo momento, il ciclo esterno permette all'utente di continuare.

Il ciclo interno, al contrario, non ha una condizione di terminazione ('`while(1)`') e come tale permette all'utente di inserire numeri un numero potenzialmente infinito di numero prima di fermarsi: ogni volta che l'utente inserisce un numero valido, 'i' viene incrementata, tuttavia la mancanza di un controllo che limiti i a rimanere entro i confini dell'array, che è di dimensione 10, permette all'utente di inserire più numeri portando ad un errore di segmentazione quando il programma cercherà di accedere a una zona di memoria non allocata.

Come si può notare dall'output, un inserimento fuori limite, ossia più di 10 numeri senza controllare l'indice 'i', può portare a comportamenti non definiti, come in questo caso valori casuali e non ordinati nell'array.

```

Inserire fino a 10 interi (inserire -1 per fermarsi):
[1]: 1
[2]: 3
[3]: 2
[4]: 5
[5]: 4
[6]: 6
[7]: -1
Il vettore inserito è:
[1]: 1
[2]: 3
[3]: 2
[4]: 5
[5]: 4
[6]: 6
Il vettore ordinato è:
[1]: 1
[2]: 2
[3]: 3
[4]: 4
[5]: 5
[6]: 6

```

```

Hai inserito solo 6 numeri. Inserisci altri numeri:
5
4
7
7
8
6
5
4
10
13
24
46
-1
Il vettore aggiornato è:
[1]: 1
[2]: 2
[3]: 3
[4]: 4
[5]: 5
[6]: 6
[7]: 5
[8]: 4
[9]: 7
[10]: 8
[11]: 7
[12]: 6
[13]: 5
[14]: 4
[15]: 14
[16]: 47
[17]: 1
[18]: 0
[19]: -136385398
[20]: 32767
[21]: -9056
[22]: 32767
[23]: 1431654745
[24]: 21845
[25]: 1431650368
[26]: 1
[27]: -9032

```

Utilizzando lo stesso codice, è inoltre possibile causare un errore di segmentazione anche con una minuscola modifica, ossia la non indicizzazione della variabile 'i' all'inizio del programma.

```

Exception has occurred. x
Segmentation fault

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

[1286]: 6648417
[1287]: 1598506072
[1288]: 1179537219
[1289]: 1147094857
[1290]: 1028870729
[1291]: 1668572463
[1292]: 1734637615
[1293]: 1329879552
[1294]: 1380533332
[1295]: 1869098813
[1296]: 1798268269
[1297]: 6909025
[1298]: 1162758476

```

Questo succede perché in C una variabile che viene dichiarata ma non inizializzata può contenere un valore casuale (altrimenti detto 'valore spazzatura'). Quando si usa questa variabile in un ciclo while, come in questo caso `'while (i < 10)'`, il suo comportamento è imprevedibile e il ciclo potrebbe non essere eseguito correttamente, facendo accedere il programma a posizioni di memoria non valide dell'array, provocando un errore di segmentazione.

Controlli di input

I controlli di input sono fondamentali per garantire la robustezza e la sicurezza del programma in quanto non solo prevengono il crash del programma a causa di input non validi, ma proteggono anche la memoria del programma da accessi non autorizzati, contribuendo al mantenimento di un comportamento prevedibile e sicuro.

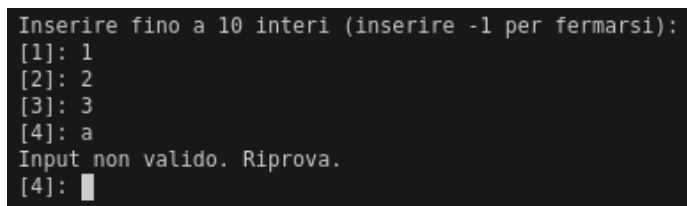
In particolare, possono essere implementati due strumenti di controllo dell'input per rendere più sicura l'esecuzione del programma (vedere [Appendice 2](#) per il codice completo):

1. La validazione dell'input: nel blocco di codice che prende input dall'utente attraverso la funzione 'scanf', prima che il valore venga accettato e memorizzato, il programma verifica che l'input sia effettivamente un numero intero.

```
if (scanf("%d", &temp) != 1) {  
    printf("Input non valido. Riprova.\n");  
    while (getchar() != '\n');  
    continue;  
}
```

Il formato '%d' indica che il programma si aspetta infatti un numero intero: se l'input fornito dall'utente è valido (cioè corrisponde al tipo di dato richiesto), l'input viene registrato con successo ed è memorizzato nella variabile 'temp'. Al contrario, se l'input non è valido (ad esempio, una lettera o un simbolo invece di un numero), non sarà registrato dal programma e non sarà memorizzato in 'temp'. Questo comportamento è fondamentale per la gestione degli errori nel programma.

In caso di input non valido, il programma notifica l'utente con un messaggio di errore e utilizza un ciclo '`while (getchar() != '\n')`' per pulire il buffer, rimuovendo eventuali caratteri rimanenti fino a raggiungere la fine della linea, aiutando a prevenire un ciclo infinito di errori di input.



```
Inserire fino a 10 interi (inserire -1 per fermarsi):  
[1]: 1  
[2]: 2  
[3]: 3  
[4]: a  
Input non valido. Riprova.  
[4]:
```

2. Controllo dell'overflow dell'array: prima di memorizzare un numero nel vettore, il programma controlla se la variabile 'i', che tiene traccia del numero di elementi inseriti è inferiore a 10. Se 'i' è già pari a 10, il programma informa l'utente che ha raggiunto il limite massimo di inserimenti e non consente ulteriori inserimenti, di modo da evitare di scrivere oltre i confini dell'array per non portare a errori di segmentazione o comportamenti imprevisti.

```
if (i < 10) {  
    vector[i] = temp;  
    i++;  
} else {  
    printf("Limite massimo raggiunto. Non puoi inserire più di 10 numeri.\n");  
}
```

```
177:
Hai inserito solo 7 numeri. Inserisci altri numeri:
3 4 5 6 7 8
Limite massimo raggiunto. Non puoi inserire più di 10 numeri.
Il vettore aggiornato è:
[1]: 3
[2]: 4
[3]: 5
[4]: 6
[5]: 7
[6]: 8
[7]: 9
[8]: 3
[9]: 4
[10]: 5
Il vettore ordinato aggiornato è:
[1]: 3
[2]: 3
[3]: 4
[4]: 4
[5]: 5
[6]: 5
[7]: 6
[8]: 7
[9]: 8
[10]: 9
```

Conclusioni

Il programma, progettato per leggere e ordinare un array di 10 numeri interi dall'utente, mostra come sia possibile manipolare e visualizzare i dati in modo efficiente attraverso l'implementazione di un algoritmo di tipo 'bubble sort'. L'analisi mette tuttavia in luce i rischi connessi a un approccio che non prevede un adeguato controllo degli input, in questo particolare caso non ponendo limiti ai dati inseribili in un array. L'assenza di un limite nei cicli di input può portare a situazioni in cui il programma tenta di accedere a memoria non a lui allocata, evidenziando i potenziali pericoli di buffer overflow.

Le modifiche proposte al codice originale, in particolare l'introduzione della possibilità di inserire un numero indefinito di input, mettono in risalto l'importanza di gestire correttamente le condizioni di terminazione dei cicli. È infatti fondamentale implementare controlli sugli input per garantire che non venga superato il limite predefinito dell'array, poiché senza tali controlli, il rischio di buffer overflow aumenta, portando a comportamenti imprevedibili del programma, vulnerabilità di sicurezza e potenziali danni ai dati.

Appendice 1

```
#include <stdio.h>

int main() {

    int vector[10], i = 0, j, k;

    int swap_var;

    printf("Inserire fino a 10 interi (inserire -1 per fermarsi):\n");

    while (i < 10) {

        printf("[%d]: ", i + 1);

        int temp;

        scanf("%d", &temp);

        if (temp == -1) {

            break;

        }

        vector[i] = temp;

        i++;

    }

    printf("Il vettore inserito è:\n");

    for (j = 0; j < i; j++) {

        printf("[%d]: %d\n", j + 1, vector[j]);

    }

    for (j = 0; j < i - 1; j++) {

        for (k = 0; k < i - 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 è:\n");

    for (j = 0; j < i; j++) {

        printf("[%d]: %d\n", j + 1, vector[j]);

    }

    while (i < 10) {
```



```
printf("Hai inserito solo %d numeri. Inserisci altri numeri:\n", i);
while (1) {
    int temp;

    scanf("%d", &temp);

    if (temp == -1) {
        break;
    }

    vector[i] = temp;

    i++;
}

printf("Il vettore aggiornato è:\n");
for (j = 0; j < i; j++) {
    printf("[%d]: %d\n", j + 1, vector[j]);
}

for (j = 0; j < i - 1; j++) {
    for (k = 0; k < i - 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 aggiornato è:\n");
for (j = 0; j < i; j++) {
    printf("[%d]: %d\n", j + 1, vector[j]);
}

return 0;
}
```

Appendice 2

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

```
int main() {
```

```
    int vector[10], i = 0, j, k;
```

```
    int swap_var;
```

```
    printf("Inserire fino a 10 interi (inserire -1 per fermarsi):\n");
```

```
    while (i < 10) {
```

```
        printf("[%d]: ", i + 1);
```

```
        int temp;
```

```
        if (scanf("%d", &temp) != 1) {
```

```
            printf("Input non valido. Riprova.\n");
```

```
            while (getchar() != '\n');
```

```
            continue;
```

```
        }
```

```
        if (temp == -1) {
```

```
            break;
```

```
        }
```

```
        if (i < 10) {
```

```
            vector[i] = temp;
```

```
            i++;
```

```
        } else {
```

```
            printf("Limite massimo raggiunto. Non puoi inserire più di 10 numeri.\n");
```

```
        }
```

```
    }
```

```
    printf("Il vettore inserito è:\n");
```

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

```
        printf("[%d]: %d\n", j + 1, vector[j]);
```

```
    }
```

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

```
        for (k = 0; k < i - 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 è:\n");
for (j = 0; j < i; j++) {
    printf("[%d]: %d\n", j + 1, vector[j]);
}
while (i < 10) {
    printf("Hai inserito solo %d numeri. Inserisci altri numeri:\n", i);
    while (1) {
        int temp;

        if (scanf("%d", &temp) != 1) {
            printf("Input non valido. Riprova.\n");
            while (getchar() != '\n');
            continue;
        }

        if (temp == -1) {
            break;
        }

        if (i < 10) {
            vector[i] = temp;
            i++;
        } else {
            printf("Limite massimo raggiunto. Non puoi inserire più di 10 numeri.\n");
            break;
        }
    }
    printf("Il vettore aggiornato è:\n");
    for (j = 0; j < i; j++) {
        printf("[%d]: %d\n", j + 1, vector[j]);
    }
    for (j = 0; j < i - 1; j++) {
        for (k = 0; k < i - 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 aggiornato è:\n");  
for (j = 0; j < i; j++) {  
    printf("[%d]: %d\n", j + 1, vector[j]);  
}  
}  
  
return 0;  
}
```