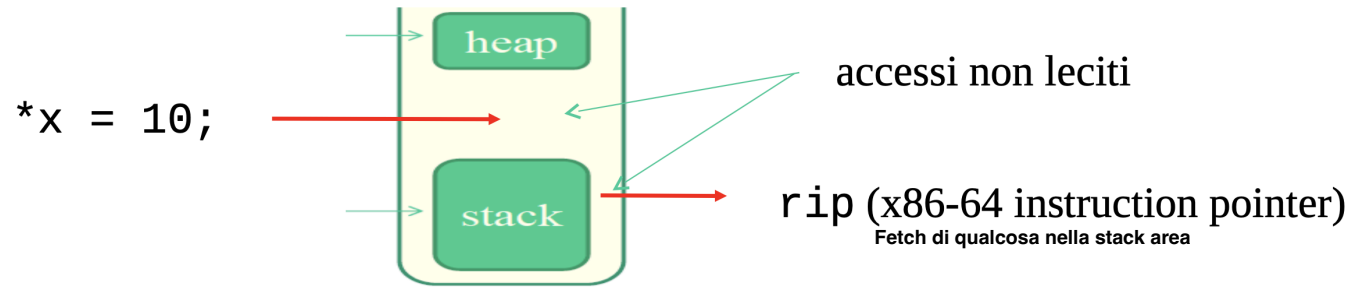
Segmentation Fault

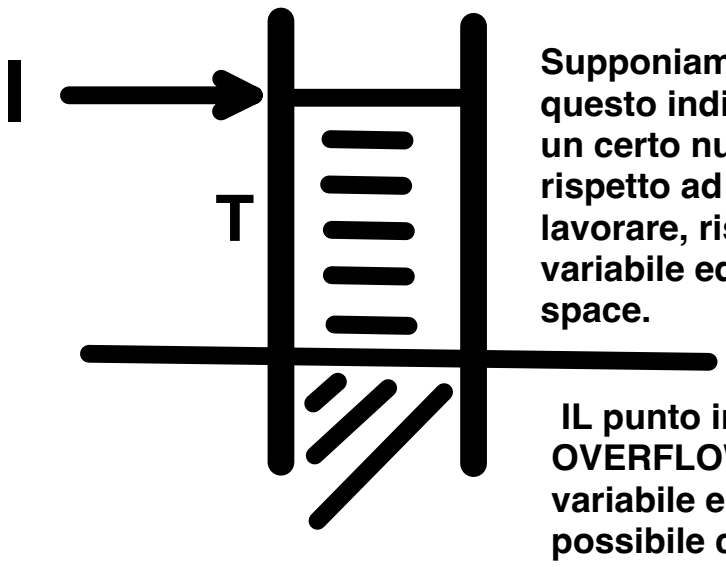
domenica 26 marzo 2023 12:17

Errori di segmentazione (segmentation fault)

- sono legati ad accesso a zone dello spazio di indirizzamento correntemente non valide



Quando abbiamo puntamento + spiazzamento e questo eventualmente ci porta in buffer overflow, è possibile che all'interno di un address space di un'applicazione, succedano cose di questo tipo:



Supponiamo di avere una locazione utilizzabile a questo indirizzo I, e questo ha una certa taglia, un certo numero di byte T. Se ci spiazziamo rispetto ad I, all'interno della memoria per lavorare, rischiamo di andare all'esterno della variabile ed entrare in un'altra zona dell'address space.

IL punto importante legato al BUFFER OVERFLOW è che quando usciamo da quella variabile e andiamo nella zona di sotto, è possibile che questa zona non sia correttamente utilizzabile ora in questo momento.

Quindi noi ci stiamo comunque muovendo all'interno dell'address space, ma magari stiamo lavorando ad un certo offset di questo address space, in cui è possibile che non ci siano delle zone realmente utilizzabili ora. Magari se chiamiamo il sistema operativo e chiediamo di poterle utilizzare, queste ci vengono date in uso. Però di fatto attualmente non sono utilizzabili.

Questo è il classico errore della memoria che si chiama "SEGMENTATION FAULT".

Noi ad esempio abbiamo un pointer, cerchiamo di accedere all'informazione effettivamente puntata, ma questo pointer casca in un punto anomalo dell'address space. Ovvero punta ad un punto che non è coperto da informazioni attualmente raggiungibili.

Un punto esterno a tutte le zone dell'address space. Per poter lavorare nel punto anomalo, l'unica cosa che si poteva fare è estendere l'heap, MA SUPPONIAMO DI NON AVERLO FATTO: Questo rimane un segmentation fault.

- sono anche legati ad accessi allo spazio di indirizzamento in modalita' non conforme alle regole che il sistema operativo impone
  - ✓ `.text` e' configurato read/exe
  - ✓ `.stack` e' tipicamente configurato read/write (ma non exe, almeno su processori moderni, e.g. x86-64, e senza flag di compilazione `-z execstack`)

ESEMPIO BASIC BUFFER OVERFLOW

basic-buffer-overflow.c

```c
#include <stdio.h>

int control_variable;
int v[10];

int main(int argc, char * argv[]){

    int index, value;

#ifdef IN_STACK
    int control_variable;
    int v[10];
#endif

```

Abbiamo un settaggio ad 1 di una variabile che potrebbe essere una variabile globale, oppure se noi compiliamo definendo IN_STACK, il setting della variabile va a toccare la definizione all'interno dell'IF-DEF.

Questa zona di codice esisterà o non esisterà nella versione eseguibile di questa

```c
    control_variable = 1;

    while (control_variable == 1){
        scanf("%d%d",&index,&value);
        v[index] = value;
        printf("array elem at position %d has been set to value %d\n",index,v[index]);


    }

    printf("exited cycle\n");

    return 0;
}
```

applicazione, a seconda del valore che noi scegliamo per IN_STACK.
Se definiamo IN_STACK in compilazione avremo questa variabile locale e v[10].

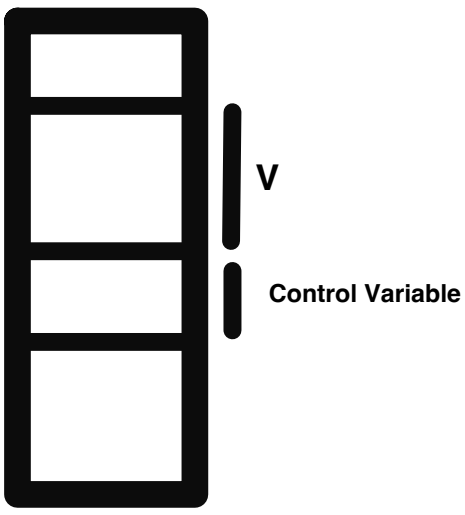
Finché control_variable==1, noi andiamo ad eseguire una scanf e cerchiamo di prelevare due interi, uno lo andiamo a scrivere esattamente nella variabile indice e un altro all'interno della variabile value, di cui ovviamente passiamo gli indirizzi di memoria alla funzione scanf().

Andiamo a cambiare il valore dell'array v, in base all'index che abbiamo acquisito precedentemente.

Scanf() non esce fuori dalle aree di memoria nel momento in cui dovrà eseguire il lavoro di consegnare le informazioni al chiamante.
Però con l'istruzione dopo scanf, stiamo adottando uno schema in cui eseguiamo puntamento alla memoria tramite V, identifichiamo l'indirizzo index, ci spiazziamo, ed accediamo.
Qua rischiamo il buffer overflow, perché l'indice potrebbe essere un valore che viene letto sempre da scanf() che potrebbe però andare all'esterno del numero di elementi (10) che caratterizzano l'array V.

Noi ci aspettiamo che il while non termini mai, abbiamo control_variable==1 e rimarrà ad 1 sempre.
PERÒ TRAMITE IL BUFFER OVERFLOW È POSSIBILE CHE NOI ANDIAMO A TOCCARE QUESTA CONTROL_VARIABLE.

Quando noi andremo a compilare questo programma, in realtà, nell'address space le locazioni di memoria di control_variable e dell'array V, saranno una accanto all'altra.



L'offset pari a 10 ci porta all'esterno dell'array, perché le celle di memori vanno da 0 a 9.

Se noi andiamo in V, prendendone l'indirizzo e cerchiamo di fare un'assegnazione al di fuori di v, usando questo approccio di buffer overflow, rischiamo di andare a sovrascrivere il valore di control variabile.

Se noi passiamo un indice che ci porta all'esterno dell'array V andiamo sull'intero successivo.

Questo è quello che andremo a fare

```
PLES/C-BASICS> ./a.out
7 8
array elem at position 7 has been set to value 8
```

```
10 0
array elem at position 10 has been set to value 0
exited cycle
```

Settiamo a zero un elemento della memoria che però non è più un elemento dell'array.
Inseriamo indice 10 e value 0.

Ovviamente abbiamo un'uscita

dal ciclo perché stiamo cambiando quella variabile di controllo.

Questo è un esempio di buffer overflow leggermente diverso a quelli già visti, ad esempio supponiamo di avere un'area che è quella dell'array V, sto sfruttando il fatto che a partire da V, posso calcolare un altro indirizzo per andare direttamente a scrivere nella nuova locazione, V[10] fa esattamente questo: senza toccare quello che ho in mezzo: