Relazione Buffer Overflow

Un attacco di tipo buffer overflow consiste nell'andare a scrivere all'interno di un buffer più dati di quanti ne possa contenere, andando così a sovrascrivere la memoria che si trova oltre il buffer.

Scopo

Lo scopo principale di questo attacco è di andare a sovrascrivere il **return address** di una funzione, così che appena la funzione finisce possiamo ridirezionare il flusso del programma dove preferiamo.

Chiamata funzioni

Nell'architettura x64 quando viene fatta una chiamata a funzione, vengono usati i registri per passare i parametri (in caso sono troppi si utilizza lo stack).

In particolare l'istruzione assembly `call funzione` esegue due cose:

- Pusha nello stack il valore della prossima istruzione.
- Imposta il valore di `$rip` al primo indirizzo della funzione.
  Così facendo viene eseguita la funzione.

Alla fine di ogni funzione si trova invece l'istruzione `ret`, che nel nostro caso è equivalente ad una `pop rip`, nel senso che prende il primo elemento dallo stack, cioè il return address pushato dalla funzione chiamante, e lo mette all'interno di `$rip`, tornando così ad eseguire la funzione chiamante.

Programma vulnerabile

Come prima cosa dobbiamo creare un programma che sia vulnerabile.
La vulnerabilità principale che sfruttiamo è quella della funzione

```
// Funzione deprecata
gets(buffer)
```

La funzione `gets()` prende l'input da tastiera e lo inserisce all'interno del buffer, senza fare controlli sulla lunghezza di esso, andando quindi potenzialmente a sovrascrivere la memoria fuori dal buffer.

Compilazione

Per aggirare vari sistemi di sicurezza messi in pratica dal compilatore, dobbiamo compilare il nostro programma in questo modo.

```
gcc buffer.c -o buffer -fno-stack-protector -z execstack -std=c99
```

In particolare l'opzione `std=c99` serve per poter usare la funzione `gets()` che, oltre ad essere deprecata, non è più presente nelle ultime versioni di C.

Il resto delle opzioni di compilazioni vediamo a cosa servono dopo quando parliamo delle problematiche di questo attacco.

Attacco

L'esecuzione dell'attacco sembra all'inizio piuttosto immediata.
Alcuni strumenti utili sono:

- **GDB** (Gnu DeBugger): utile per analizzare l'esecuzione del programma e trovare gli indirizzi che ci interessano.
  - **pwndbg**: un estensione di GDB che ha vari strumenti integrati molto utili per progetti di questo tipo.
- **Linguaggio di scripting**, nel nostro caso python, per andare a costruire il ***payload*** da iniettare.

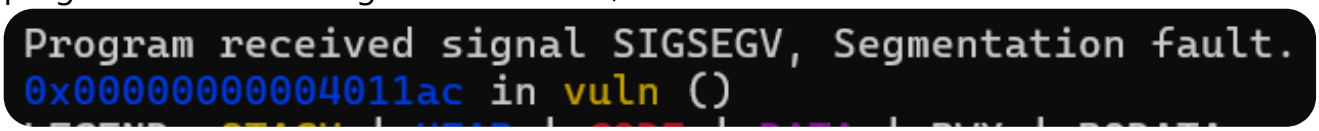
Offset del Return Address

Come prima cosa dobbiamo capire dopo quanti byte, a partire dal buffer, si trova il return address che dobbiamo andare a sovrascrivere.

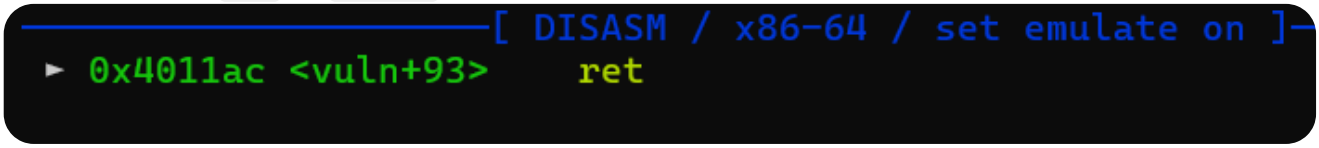
Come prima cosa andiamo, tramite la funzione `cyclic` di pwndbg, a generare una stringa sufficientemente lunga da far crashare il nostro programma.

```
pwndbg> cyclic 600
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaa
aaaakaaaaaaalaaaaaaamaaaaaaanaaaaaaaoaaaaaaapaaaaaaaqaaaaaaaraaaaaaasaaaaaaa
taaaaaaauaaaaaaavaaaaaaawaaaaaaaxaaaaaaayaaaaaaazaaaaaabbaaaaaabcaaaaaabdaaa
aaabeaaaaaabfaaaaaabgaaaaaabhaaaaaabiaaaaaabjaaaaaabkaaaaaablaaaaaabmaaaaaab
naaaaaaboaaaaaabpaaaaaabqaaaaaabraaaaaabsaaaaaabtaaaaaabuaaaaaabvaaaaaabwaaa
aaabxaaaaaabyaaaaaabzaaaaaacbaaaaaaccaaaaaacdaaaaaaceaaaaaacfaaaaaacgaaaaaac
haaaaaaciaaaaaacjaaaaaackaaaaaaclaaaaaacmaaaaaacnaaaaaacoaaaaaacpaaaaaacqaaa
aaacraaaaaacsaaaaaactaaaaaacuaaaaaacvaaaaaacwaaaaaacxaaaaaacyaaaaaac
```

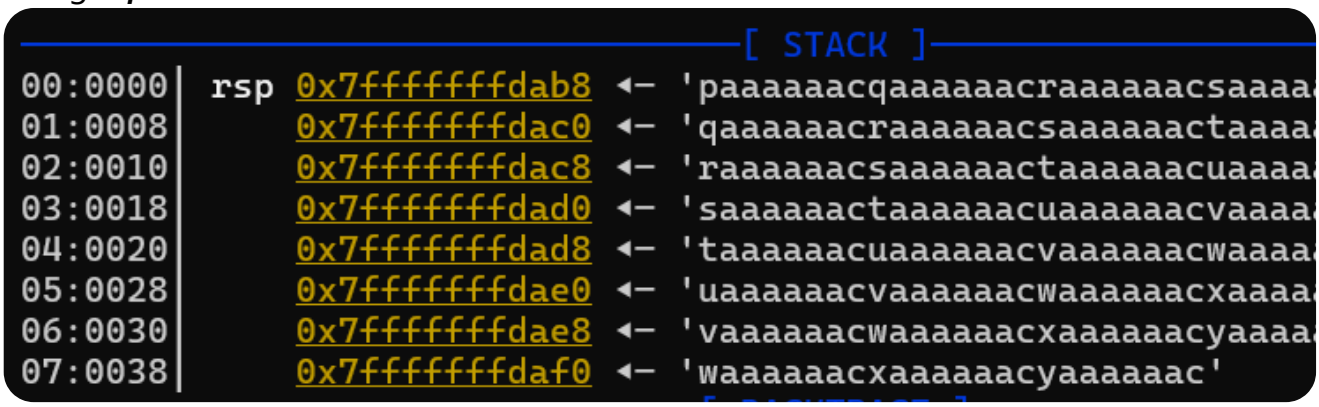
Se andiamo poi ad eseguire il programma, dando in input questa stringa, avremo che il programma andrà in Segmentation Fault, crashando.



Andando ad analizzare la situazione possiamo vedere che il programma è crashato sull'istruzione `ret` di `vuln()`.



Guardando lo stack possiamo vedere che i primi 8 byte presenti nello stack sono la stringa "***paaaaaac***"



La particolarità di questa stringa generata da cyclic è che, prendendo 8 byte ovunque nella stringa, possiamo trovare l'offset preciso dall'inizio della stringa.

Abbiamo così trovato che il return address si trova dopo **520 byte** a partire dall'inizio del buffer.

Shellcode

Il nostro scopo è quello di inserire come return address un indirizzo che fa proprio parte del buffer, così che possiamo, insieme al return address, iniettare anche il codice che noi vogliamo, così che il programma esegua del codice arbitrario.

Come codice da iniettare abbiamo progettato un codice molto corto, di 13 byte + 8 byte della stringa "/bin/sh".
Questo codice consiste in una syscall alla funzione `execve()` passando come parametro ***/bin/sh***, così da ottenere una shell.

NOP Slide

Visto che i programmi quando vengono caricati nello stack possono avere un po' di "gioco" ed essere posizionati in punti leggermente diversi.
Andiamo quindi ad inserire come offset una serie di operazioni "NOP", che in x64 equivalgono al codice esadecimale **0x90**, e sono lunghe quindi un singolo byte.

Come indirizzo di ritorno inseriamo poi l'indirizzo a metà del NOP slide, così che anche se dovessero muoversi un po', finiremmo sempre ad eseguire quelle NOP, arrivando così alla fine dello slide, dove si trova il nostro shellcode.

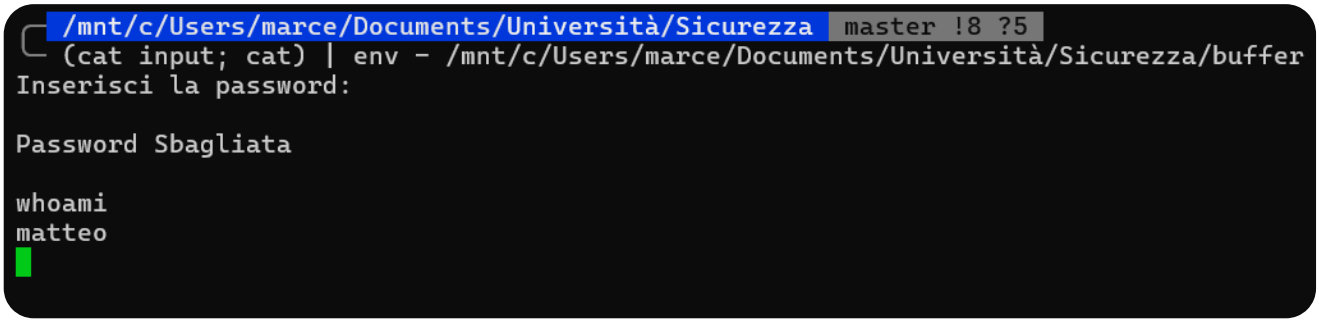
Esecuzione attacco

Per l'esecuzione finale dell'attacco dobbiamo poi usare questo comando da terminale:

```
(cat input; cat) | env - /full/path/to/buffer
```

Dobbiamo farlo così per simulare le condizioni di quando lo eseguiamo con GDB, in particolare:

- Inseriamo il full path dell'eseguibile, perché è quello che fa GDB.
- Usiamo `env -` per rimuovere tutte le variabili d'ambiente, che possono andare a spostare il nostro programma all'interno dello stack, e così il nostro return address non punterebbe più alla NOP slide, ma in una parte casuale dello stack.



Problematiche

Come detto prima, per rendere questo attacco possibile siamo andati a compilare il nostro codice in maniera particolare.
Andiamo a vedere ognuna delle cose che abbiamo fatto.

Canary

L'opzione che abbiamo inserito `-fno-stack-protector` serve a disattivere lo StackGuard, o canary.
Questa protezione consiste nell'inserire 8 byte, di cui i primi 7 randomici e l'ultimo un null byte.

Questa protezione viene inserita nello stack, tra lo spazio dedicato alle variabili di una funzione, e il return address dove dovrà tornare.
Subito prima di andare a fare l'istruzione ***ret*** viene inserita una routine che controlla se il valore del canary è cambiato, e in caso sia cambiato il processo viene abortito.

In questo modo se, come nel nostro caso, qualcuno dovesse operare un attacco di tipo buffer overflow, andrebbe a sovrascrivere anche il canary, e quindi il processo viene abortito prima che possano essere fatti danni.

Aggirare

Un modo possibile di aggirare questa protezione, anche se tutt'altro che facile, consiste nello sfruttare una qualche altra vulnerabilità, ad esempio della funzione `puts()` per andare a leggere dalla memoria il valore del canary in tempo reale, e andare a modificare il payload in modo che lasci il canary invariato.

Anche se questo risulta più complicato di quello che sembra, perché il primo byte del canary è un null byte, se dovessimo quindi inserire questo byte all'interno della stringa che diamo alla **gets()** la funzione smetterebbe di leggere lì la stringa, senza quindi scrivere sulla canary o anche sul return address.

NX Bit

L'opzione di compilazione `-z execstack` serve a settare l'NX bit, quello che ci dice se lo stack è eseguibile o meno, a 0.

Se dovessimo lasciare l'NX bit settato avremmo che lo stack non è eseguibile e quindi, quando proviamo a ridirezionare il flusso del programma all'interno dello stack il programma si accorge di star eseguendo delle istruzioni presenti nello stack, e crasherebbe.

Aggirare

Aggirare questa limitazione risulta leggermente più facile di aggirare il Canary, ma rimane comunque insidiosa.

Per risolvere questa problematica si può fare uso di una tecnica chiamata **ROP** (Return Oriented Programming), che consiste nel far uso di porzioni di codice, dette ***gadget***, già

presenti all'interno del codice del programma, per fare quello che vogliamo.

Questi gadget hanno la particolarità che terminano tutti con l'istruzione `ret`, se quindi iniettiamo nello stack una serie di indirizzi di questi gadget, possiamo eseguirli uno dopo l'altro, senza mai eseguire nessuna istruzione presente nello stack, poiché tutti i gadget sono all'interno del **.text**.

Per trovare i gadget esistono vari tool come `ROPGadget` o `radare2`, che permettono con semplicità di trovare tutti i gadget all'interno di un binario.

Un attacco molto comune di questo tipo è detto **ret2system** o **ret2libc** il cui scopo è quello di eseguire la funzione

```
system("/bin/sh")
```

il cui risultato è equivalente a quello della `execve`.

Per farlo bisogna prima trovare:

- l'indirizzo della system all'interno della libc
- l'indirizzo della stringa "/bin/sh" sempre dentro la libc
- l'indirizzo di un gadget `pop rdi` nel nostro eseguibile

Andando così a concatenare l'indirizzo del gadget, l'indirizzo della stringa e poi l'indirizzo della system, così che mettiamo dentro rdi (il registro usato per passare il primo parametro) l'indirizzo della stringa "/bin/sh", e poi facciamo la chiamata a system().

ASLR

Infine, uno dei problemi più grandi è l'ASLR (Address Space Layout Randomization) che, ogni volta che si esegue un programma, lo posiziona in porzioni diverse dello stack, così da avere sempre indirizzi diversi, rendendo ogni volta diverso il return address da inserire.

Per disabilitarlo, su Linux, basta modificare il file **/proc/sys/kernel/randomize_va_space**, normalmente settato a 2, e mettere 0, così da disattivare l'ASLR.

Versione 2 NX Bit

Nella "versione 2" dell'attacco sono andato a riabilitare il bit NX, quindi rendendo lo stack ***non eseguibile***.

Questo attacco è stato fatto diversamente, invece di iniettare lo shellocode nel buffer, è stato fatto un attacco di tipo **ret2libc**.

Trovare l'indirizzo della libc

Come prima cosa dobbiamo vedere dove si trova in memoria la ***libc***, per fare ciò possiamo usare il comando `ldd buffer`, che cerca le librerie usate dal nostro eseguibile, tra cui appunto la libc, e ci dice dove si trovano in memoria.

```
/mnt/c/Users/marce/Documents/Università/Sicurezza/progetto master !8 ?6
ldd bufferEXEC
        linux-vdso.so.1 (0x00007ffff7fc7000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007ffff7dae000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x00007ffff7fc9000)
```

Da questo vediamo dove si trova nel sistema la libc, cioè in `/usr/libc/libc.so.6` e vediamo inoltre che in memoria si trova all'indirizzo `0x00007ffff7dae000`.

A partire poi dalla base di libc possiamo recuperare le due cose che ci servono

Funzione System

Noi faremo uso della funzione `system()`, quindi dobbiamo trovare dopo quanto la "base" di libc si trova, e per fare ciò possiamo usare `readelf` sulla libc per analizzare l'eseguibile.

```
/mnt/c/Users/marce/Documents/Università/Sicurezza/progetto master !8 ?6
readelf -s  /usr/lib/libc.so.6 | grep "system"
1050: 0000000000050f10    45 FUNC    WEAK   DEFAULT   15 system@@GLIBC_2.2.5
```

Stringa "/bin/sh"

Come parametro dovremo passare alla funzione system la stringa "**/bin/sh**", che possiamo trovare sempre all'interno della libc, stavolta tramite `strings`.

```
/mnt/c/Users/marce/Documents/Università/Sicurezza/progetto master !8 ?6
strings -t x /usr/lib/libc.so.6| grep /bin/sh
1aae28 /bin/sh
```

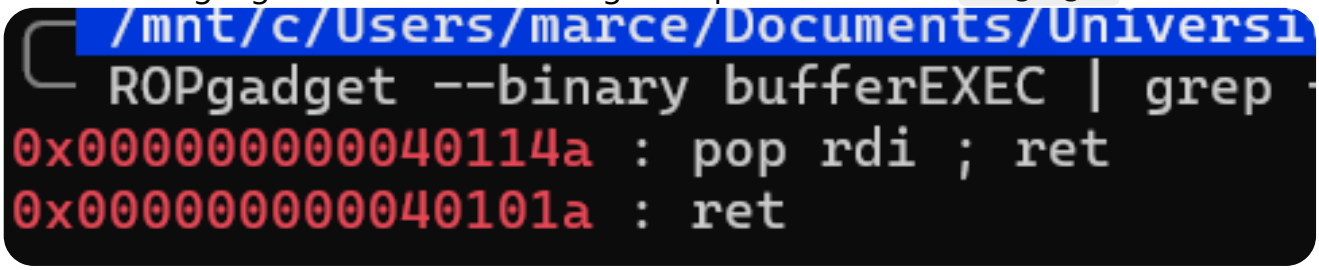
Gadget

Trovato l'indirizzo della funzione da chiamare, e l'indirizzo del parametro da passargli, dobbiamo caricare il parametro, la stringa ***/bin/sh***, all'interno di RDI, il registro usato per passare il primo parametro.
Per fare ciò useremo un gadget.

> **Gadget**

Sono detti gadget tutte quelle porzioni di codice assembly, già presenti all'interno del binario che stiamo eseguendo, che terminano con l'istruzione `ret`.
Il fatto che terminino con `ret` ci permette di concatenare vari gadget uno dopo l'altro per alterare l'esecuzione del programma come preferiamo.

Per trovare i gadget all'interno di un eseguibile possiamo usare `ROPgadget`.



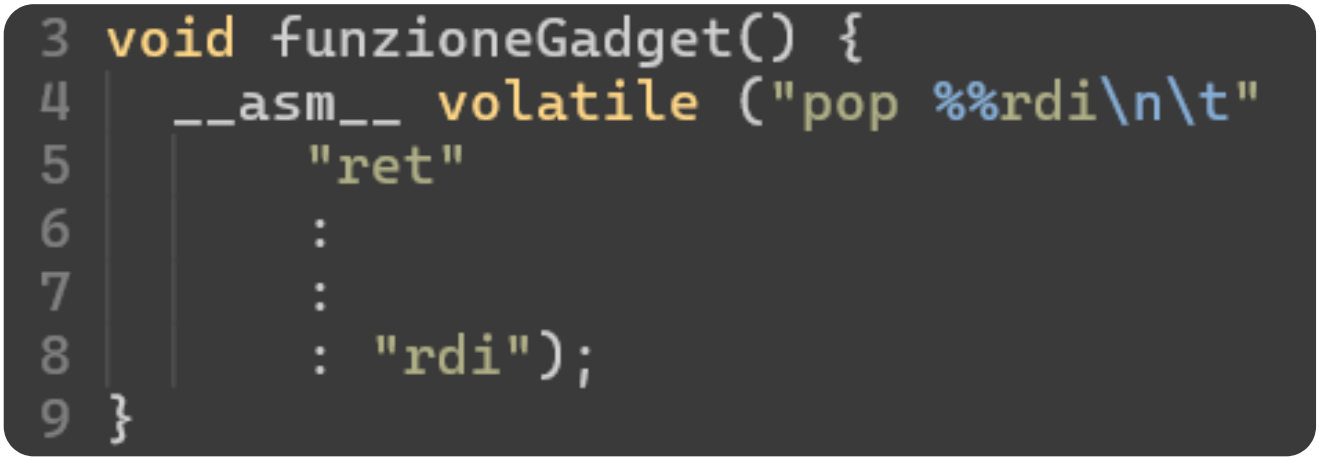
Inserire `pop rdi` gadget

In questo caso specifico il nostro programma non conteneva nessun gadget del tipo:

```
pop rdi
ret
```

che è fondamentale per poter passare il parametro alla funzione `system`.

Siamo andati quindi ad "inserire" il gadget artificialmente nel binario tramite la seguente funzione.



Specifichiamo il fatto che questa azione non è di solito necessaria, è stata necessaria in questo caso vista la dimensione esigua del programma, e quindi i pochi gadget generati.
Di solito, in eseguibili più grandi, si possono trovare migliaia di gadget, e quindi la probabilità di trovare il gadget giusto per la situazione aumentano.

```
/mnt/c/Users/marce/Documents/Università/Sicurezza/progetto  master !8 ?6                                SEGV
└ (cat inputEXEC; cat) | env - /mnt/c/Users/marce/Documents/Università/Sicurezza/progetto/bufferEXEC
Benvenuto in questo sistema di sicurezza super sicuro!
Inserisci la password:

Password Sbagliata
whoami
matteo
ls
'Relazione Buffer Overflow.pdf'   bufferCANARY   bufferCANARY.c   bufferEXEC   bufferEXEC.c   inputCANARY.py   inputEXEC   inputEXEC.py
```

Versione 2 canary

Come detto, il canary è abbastanza arduo da aggirare, poiché richiede un'altra vulnerabilità all'interno del codice.

Leakkare il canary

In questo caso quindi siamo andati a sfruttare la vulnerabilità di un codice leggermente modificato.

Questa vulnerabilità consiste nello stampare le stringhe inserite da un utente direttamente con la `printf()`, senza formattarle tramite `%s`.

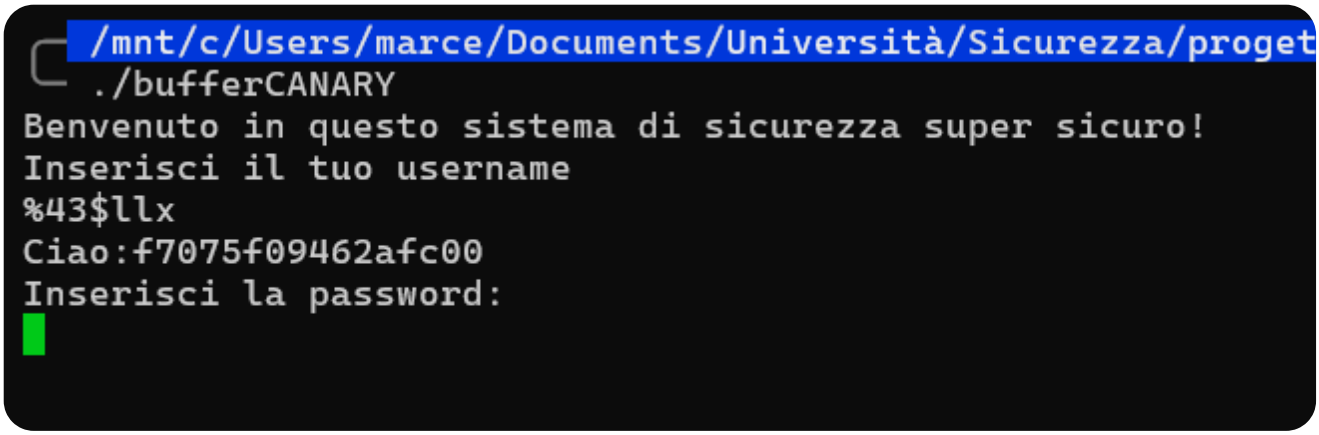
```c
13      char buf[64];
14      printf("Inserisci il tuo username\n");
15
16      gets(buf);
17      printf("Ciao:");
18      printf(buf);
19      printf("\n");
```

In questo caso l'istruzione vulnerabile è quella a riga 18.
Andando a stampare direttamente la nostra stringa, possiamo iniettare dei modificatori che ci permettono di andare a leggere dentro lo stack, ovvero `%llx`.

```
/mnt/c/Users/marce/Documents/Università/Sicurezza/progetto  master !8 ?6              INT x  34s  23:49:34
└ ./bufferCANARY
Benvenuto in questo sistema di sicurezza super sicuro!
Inserisci il tuo username
%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,
%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx
Ciao:6f616943,0,0,4057a0,0,6c6c252c786c6c25,252c786c6c252c78,786c6c252c786c6c,6c252c786c6c252c,2c786c6c252c786c,6c6c252c786c6c25,252c786c6c252c78,786c6c252c786c6c,6c252c7
86c6c252c,2c786c6c252c786c,6c6c252c786c6c25,252c786c6c252c78,786c6c252c786c6c,6c252c786c6c252c,2c786c6c252c786c,6c6c252c786c6c25,252c786c6c252c78,786c6c252c786c6c,6c252c7
86c6c252c,2c786c6c252c786c,6c6c252c786c6c25,252c786c6c252c78,786c6c252c786c6c,6c252c786c6c252c,2c786c6c252c786c,6c6c252c786c6c25,252c786c6c252c78,786c6c252c786c6c,6c252c7
86c6c252c,786c6c252c786c,4f38633,66637502,4f38633,0,0,0,0,ed5915659a3d1800,37f,7ffff7f915c0,7ffff7f8f050,7ffff7e38ca5,7ffff7f915c0
Inserisci la password:
```

In questo caso notiamo che la stringa sottolineata è formata proprio come è fatta un canary, quindi è composta da 8 byte, di cui l'ultimo è un null byte, e i primi 7 sembrano random, e infatti cambiano ogni volta ad ogni esecuzione.

Possiamo quindi salvarci che numero di stringa sia quella, in questo caso la 43, così che possiamo stampare direttamente il canary.



Troviamo così ogni volta quale è il canary.
Tramite l'uso di **pwntools**, una libreria di python, possiamo poi andare ad interagire con il processo, leggendo la canary e potendo poi riscriverla al posto giusto.