



## RELAZIONE DI NETWORK SECURITY

---

*Studenti:*

Marco De Stefano

*N° Matricola:*

7025991

Regino Kamberaj

7164400

Kirollo Habib

7060226

*Nome Squadra:*

Scualo

*Data:* Dicembre 2024

*Id:*

73

---

# Contents

<b>1 Realizzazione del Container</b>	<b>2</b>
1.1 Scelta delle Vulnerabilità . . . . .	2
1.2 Preparazione del Container . . . . .	3
1.2.1 Struttura del Codice . . . . .	3
1.2.2 Deployment del Container . . . . .	5
1.3 Modalità di Attacco . . . . .	7
1.4 Mitigazioni alle Vulnerabilità . . . . .	9
<b>2 Attacchi eseguiti per cattura CTF</b>	<b>10</b>
2.1 LANgo Unchained . . . . .	10
2.1.1 Prima modalità di attacco . . . . .	10
2.1.2 Seconda modalità di attacco . . . . .	11
2.1.3 Terza modalità di attacco . . . . .	11
2.2 lo spagnolo . . . . .	12
2.2.1 Prima modalità di attacco . . . . .	12
2.2.2 Seconda modalità di attacco . . . . .	14
2.3 Salsiccia e Friarielli . . . . .	15
2.3.1 Primo tentativo . . . . .	15
2.3.2 Secondo tentativo - Utilizzo di OpenStego . . . . .	20
2.4 Conclusioni . . . . .	22
<b>3 Contributi</b>	<b>23</b>

# Terzo Assignment

Regino Kamberaj (7164400), Marco De Stefano (7025991), Kirolos Habib (7060226)

December 17, 2024

## Abstract

Il presente documento descrive l'esercizio di *Capture the Flag* (CTF) svolto in collaborazione con i colleghi Marco De Stefano e Kirolos Habib. L'attività è stata suddivisa in due principali obiettivi:

1. Creare un "Capture the Flag" personalizzato per i nostri colleghi, progettato per sfidarli a individuare vulnerabilità.
2. Catturare una o più bandiere nascoste nei container creati dai nostri colleghi, affrontando le vulnerabilità da loro aggiunte.

Il report è strutturato in due parti principali: la prima descrive la creazione del container e l'inserimento della vulnerabilità, mentre la seconda documenta il processo di cattura delle flag create dagli altri partecipanti, incluse le difficoltà incontrate e i tentativi falliti.

## 1 Realizzazione del Container

### 1.1 Scelta delle Vulnerabilità

La vulnerabilità selezionata è stata l'SQL Injection, una delle più diffuse e interessanti a livello applicativo. Questa scelta è motivata dalla sua rilevanza nel panorama delle minacce informatiche e dalla necessità di comprendere sia le cause che le soluzioni per mitigare tali attacchi.

**SQL Injection** è una vulnerabilità che consente a un attaccante di manipolare query SQL attraverso input **non sanitizzati**, compromettendo la sicurezza del database. Per *input non sanitizzati* si intendono quei dati forniti dall'utente a un'applicazione che non sono stati adeguatamente controllati, filtrati o processati prima di essere usati. Questo può includere input come moduli web, form, URL, cookie, o persino parametri nelle API.

Un Esempio comune di input non sanitizzato è il caso di un **Form senza controllo dei dati**, ovvero un modulo di login, oppure input di nuovi dati, che

accetta un qualsiasi valore inserito dall'utente senza rimuovere stringhe pericolosi come ' oppure --.

Quali possono essere i problemi di input non sanitizzati?

Se i dati forniti dall'utente non vengono verificati in modo sicuro, possono contenere del codice malevolo, o che portano a contenuti inaspettati.

Ad esempio sfruttando *query SQL*, queste permettono agli attacanti di iniettare codice per compromettere il comportamento del database o mostrare dati sensibili.

Evidenzieremo e faremo vedere nelle pagine successive come alcune query, possono essere sfruttate per questo tipo di attacco e l'importanza della sanitizzazione degli input come misura di prevenzione.

## 1.2 Preparazione del Container

L'applicazione è stata realizzata utilizzando il framework **Flask** per la parte applicativa e il modulo built-in **sqlite3** di **Python** per la gestione del database. L'idea è stata quella di realizzare un semplice sito di gestione degli utenti.

### 1.2.1 Struttura del Codice

Il codice dell'applicazione si divide in due file:

- **unsafeapp.py**: che contiene la logica di business dell'applicazione, ovvero tutto l'insieme di funzionalità che il sito offre.
- **usedb.py**: che contiene tutti i metodi per la gestione del database, fra cui inserimento e ricerca.

L'app contiene una schermata iniziale di login, una home, e una pagina **list** da cui sono visibili e modificabili i contenuti di una tabella interna **Workers**. La schermata di login, e la pagina **list** contengono entrambe delle form che, una volta inviate, causano l'esecuzione di una query sul database.

I due tipi di form sono:

- **Form di inserimento:** Per l'aggiunta di nuovi lavoratori inserendo il nome e il cognome del lavoratore.
- **Form di ricerca:** Dove poter cercare un lavoratore immettendo nome, cognome o entrambi.

Come DBMS abbiamo utilizzato il database **SQLite**, contenente lo *schema table*, ovvero una tabella **nativa** chiamata anche **sqlite\_master**, al cui interno sono presenti metadati sulle tabelle del nostro database.

Il Database è costituito dalle tabelle:

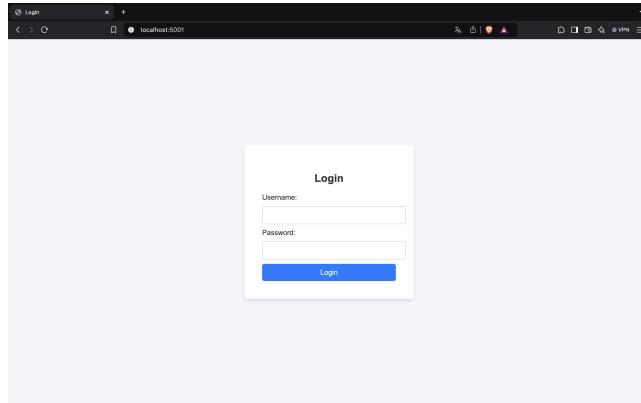


Figure 1: Pagina di Login del sito

A screenshot of a web browser window titled "Worker List Page". The title bar also shows "localhost:5001/html/list". The page has a header with three buttons: "Go to Login", "Go to Home", and "Go to List". Below the header, there is a message: "Use the forms to search or add workers. Reload the page to view any changes." There are two forms: "Add a Worker" (with fields for First Name and Last Name) and "Search for a Worker" (with fields for First Name and Last Name). Below these forms is a table titled "The List of Workers" with columns "First Name" and "Last Name". The table contains the following data:

First Name	Last Name
John	Doe
Alan	Smith
Marco	De Stefano
Bola	De

Figure 2: Pagina di inserimento e ricerca lavoratori

- **Users:** Parte della logica dell'applicazione, contiene la lista di tutte le persone autenticabili.
- **Workers:** Visibile e manipolabile dall'utente, contiene la lista dei lavoratori aggiunti dagli utenti.
- **Flags:** Nascosta, non fa parte della logica dell'applicazione, ma visibile solo tramite injection, è l'obiettivo da conquistare.

Tabelle (4)		
Workers		
id	INTEGER	
firstname	TEXT	
lastname	TEXT	
flags		
flag	TEXT	
sqlite_sequence		
users		
id	INTEGER	
username	TEXT	
password	TEXT	
role	TEXT	

Figure 3: Struttura del Database

Sono state quindi inserite **due flag** nel sistema:

- La prima flag, si trova nel percorso `/robots.txt`, generalmente utilizzato dai siti web per comunicare con i crawler dei motori di ricerca o altri bot automatizzati.
- La seconda flag, la più interessante, è raggiungibile sfruttando una vulnerabilità di SQL Injection. Si tratta di una stringa memorizzata nella tabella `flags`.

Avremmo modo di scoprire queste flags successivamente, nel capitolo *Modalità di Attacco*.

### 1.2.2 Deployment del Container

Una volta accertato il funzionamento dell'applicazione, si è creata un'immagine docker per poter girare il container anche sul computer di altri colleghi.

Per fare questo si è creata una directory `app/` con quanto necessario all'esecuzione dell'applicazione (`script`, `file database`, e `template html`).

Infine per creare l'immagine è stato utilizzato il seguente `Dockerfile`.

```
FROM python:3.7-slim-buster

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY app/ app/
WORKDIR app/

EXPOSE 5000

CMD ["flask", "--app", "unsafeapp", "run", "--debug", "--host=0.0.0.0"]
```

dove `requirements.txt` contiene le dipendenze necessarie per un applicazione `Flask` (nel nostro caso solo la libreria `Flask` per `Python`).

Vi sono alcuni dettagli da notare nel file:

- Si è scelta come immagine di partenza `python:3.7-slim-buster` per non dover creare l'ambiente `python`.
- La flag `--host 0.0.0.0` passata al comando `flask` serve a rendere l'applicazione visibile dall'esterno, se si fosse lasciato il `127.0.0.1` di default, la pagina sarebbe stata visibile solo all'interno del container.
- Nonostante l'applicazione sia esposta all'esterno, tutte le porte del container sono comunque isolate dal resto del sistema. Dunque l'istruzione `EXPOSE 5000` serve a rendere la PORTA 5000 del container (alla quale si connette di default il server debug di `flask`) visibile anche all'utente, che utilizza il container.

Preparata quindi la cartella contenente il `Dockerfile`, il file di `requirements.txt`, e la sottocartella `app/` contenente l'applicazione stessa; abbiamo creato l'immagine tramite il comando:

```
docker build -t alonzobazaar/unsafe .
```

E infine pubblicato il container su `dockerhub` tramite:

```
docker push alonzobazaar/unsafe
```

Pronto insieme ad un **Overview** con alcuni suggerimenti, ad essere attaccato da chiunque voglia.

### 1.3 Modalità di Attacco

Dopo aver definito il nostro applicativo, messo su docker, spieghiamo adesso come conquistare le due **flag**, specificando che la difficoltà di questo CTF è "facile".

Ma perchè "facile"?

Come detto precedentemente, la prima flag si trova nel percorso `/robots.txt`. Questo generalmente viene posto nella root directory di un sito web, e serve a indicare quali parti del sito dovrebbero o non dovrebbero essere scansionate, senza la necessità di alcun tipo di autenticazione.,

Nel nostro caso, se l'attaccante decide di andare sul percorso descritto si ritroverà un' ascii art di *Donkey Kong*.

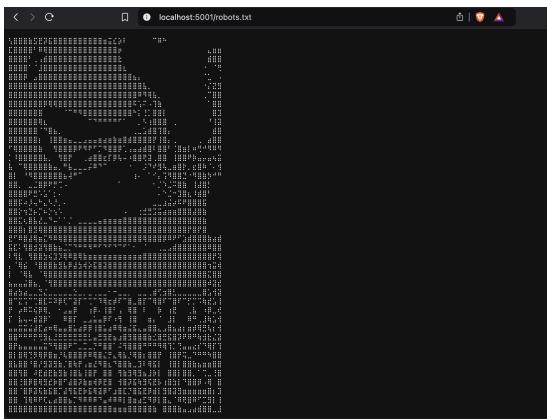


Figure 4: Prima Flag

La seconda flag è più difficile da catturare ma non abbastanza da alzare la difficoltà della cattura delle flags.

L'attaccante si ritroverà inizialmente, su una pagina di login base, che però presenta numerose vulnerabilità, ovvero:

- La tabella `Users` è popolata con un solo utente, il quale ha username e password rispettivamente: `admin` e `admin_password`.
- L'attaccante può utilizzare query mal poste per entrare nel sito.
- L'attaccante può comunque visitare ogni percorso senza essere autenticato. Può utilizzare, ad esempio, il tool `gobuster` per eseguire un *bruteforce* delle directories sul nostro applicativo e trovare quasi tutti i percorsi.

Per entrare sulla home tramite `SQL Injection` si può mettere nel campo utente la seguente query mal composta: `'' OR '1'='1' --`.

Di conseguenza al nostro database verrà formulata la seguente query:

```
SELECT * FROM users
WHERE username=' OR '1'='1' -- AND password='{password}'
```

Si nota che il carattere ' ci permette di chiudere la stringa nella quale era inserito il nome utente all'interno della query, e di inserire codice sql nel resto dell'input. Quindi OR '1'='1' --, ci permette di bypassare eventuali check sulla presenza degli input nel database di utenti, ritornando comunque VERO. Infine la parte della query che controlla la password viene totalmente bypassata poichè annullato dal marker "--".

Dunque, l'attaccante passa facilmente il login e viene portato alla home. A quel punto indirizzandosi verso la lista di lavoratori troverà i due tipi di form discussi precedentemente.

Poichè la flag è una stringa inserita in una tabella nascosta, l'attaccante deve far riferimento alla tabella `sqlite_master`, per scoprire il nome di tutte le tabelle, fra cui quella nascosta.

L'attaccante può comporre la seguente query scomposta da mettere nel form di ricerca (nel campo cognome):

```
' UNION SELECT name, name, name
FROM sqlite_master WHERE type='table'--
```

In questo caso, si chiude innanzitutto la stringa per espandere l'input, e viene usato l'operatore `union` per fare in modo che al risultato della query vengano aggiunti anche i nomi delle tabelle. Per fare questo dalla tabella `sqlite_master`, si aggiunge la condizione `where type='table'`.

`name` è presente 3 volte per non causare problemi con l'operatore `union`, in quanto la tabella `Workers`, con la quale facciamo la `Union`, contiene 3 attributi, ottendo dalla query un risultato di 3 colonne.

Il -- in fondo, anche qui rende il resto della query un commento, evitando così di causare errori di sintassi. Dunque si ottiene il seguente risultato:

The screenshot shows a web browser window titled "Search Results" with the URL "localhost:5005/worker/search". The page displays two tables of search results. The top table, titled "Matching Both", has columns "First Name", "Last Name", and "Other Data". It lists four entries: Workers, flags, sqlite\_sequence, and users. The bottom table, titled "Matching Either", also has columns "First Name", "Last Name", and "Other Data". It lists the same four entries: Workers, flags, sqlite\_sequence, and users. The "Workers" entry appears in both tables, while "flags", "sqlite\_sequence", and "users" appear only in the "Matching Both" table.

First Name	Last Name	Other Data
Workers	Workers	Workers
flags	flags	flags
sqlite_sequence	sqlite_sequence	sqlite_sequence
users	users	users

First Name	Last Name	Other Data
Workers	Workers	Workers
flags	flags	flags
sqlite_sequence	sqlite_sequence	sqlite_sequence
users	users	users

Quindi, scoperta la tabella **flags**, l'attaccante dovrà usare la stessa tipologia di query utilizzata precedentemente, ma sulla nuova tabella in modo che venga restituito il suo contenuto.

```
' UNION SELECT flag,flag,flag
FROM flags--
```

La query funziona come la precedente, ottenendo quindi il seguente risultato, e la conseguente flag nascosta:

Matching Both		
First Name	Last Name	Other Data
SEI UN CONTRABANDIERO(2 :-)	SEI UN CONTRABANDIERO(2 :-)	SEI UN CONTRABANDIERO(2 :-)
ADESSO TI MANCA DA ASCOLTARE 4000	ADESSO TI MANCA DA ASCOLTARE 4000	ADESSO TI MANCA DA ASCOLTARE 4000
VOLTE SQUALO(73	VOLTE SQUALO(73	VOLTE SQUALO(73

Matching Either		
First Name	Last Name	Other Data
SEI UN CONTRABANDIERO(2 :-)	SEI UN CONTRABANDIERO(2 :-)	SEI UN CONTRABANDIERO(2 :-)
ADESSO TI MANCA DA ASCOLTARE 4000	ADESSO TI MANCA DA ASCOLTARE 4000	ADESSO TI MANCA DA ASCOLTARE 4000
VOLTE SQUALO(73	VOLTE SQUALO(73	VOLTE SQUALO(73

Figure 5: Risultato della query, la seconda flag

## 1.4 Mitigazioni alle Vulnerabilità

Concludiamo la prima sezione, parlando di possibili mitigazioni alle vulnerabilità aggiunte nel nostro container:

- Sanitizzazione degli input:** Tutti i dati forniti dagli utenti devono essere validati e filtrati per rimuovere caratteri o stringhe pericolosi come ' e --.
- Query parametrizzate:** Utilizzare query preparate, per separare i dati forniti dall'utente dalla logica SQL, non permettendo l'iniezione di codice.
- Rimozione di informazioni superflue:** Evitare l'esposizione di file come `/robots.txt` che possono fornire informazioni utili agli attaccanti.
- Rendere più sicure le credenziali:** Le password devono essere salvate in formato cifrato con algoritmi di hashing sicuri, evitando che l'attaccante possa facilmente risalire alla password .

## 2 Attacchi eseguiti per cattura CTF

Nel seguente capitolo, parleremo invece dei risultati ottenuti provando a catturare le **flags** dei seguenti gruppi:

- LANgo Unchained (ID: 62) FLAG: Conquistata
  - lo spagnolo (ID: 4) FLAG: Conquistata
  - Salsiccia e Friarielli (ID: 313) FLAG: Conquistata

Illustreremo in quanti modi abbiamo raggiunto il flag e oltre alle vittorie, illustreremo i tentativi per vincere le flag che non siamo riusciti a conquistare.

## 2.1 LANgo Unchained

Questa prima CTF è di difficoltà facile e dopo aver costruito il container e averlo fatto partire, siamo entrati nella pagina Login.

### 2.1.1 Prima modalità di attacco

Abbiamo, quindi, utilizzato un tool chiamato gobuster che permette di trovare tutte le directories dell'applicativo attraverso un dizionario di parole che solitamente vengono utilizzate.

Figure 6: Risultati ottenuti attraverso il tool

Avendo scoperto le seguenti directories, abbiamo subito provato a entrare nella pagina /success.

Ottenendo la seguente schermata nella pagina:



Figure 7: Flag Catturata!

Ovviamente questo ci è stato permesso grazie alla vulnerabilità di poter entrare negli altri percorsi senza essere autentificato (era presente anche nel nostro sito).

### 2.1.2 Seconda modalità di attacco

Abbiamo avvertito, a quel punto, i ragazzi del gruppo e loro ci hanno informato che si poteva fare in un altro modo e che si doveva fare tramite comandi bash mettendoli nel form. Questo tipo di attacco si chiama **Command Injection**, ovviamente non ci hanno detto nessun comando per arrivare alla soluzione. Ma a questo indizio c'eravamo già arrivati dal risultato ottenuto con gobuster. Infatti tra i percorsi c'era anche \_cmd, che poteva suggerirci il modo per entrare nel sistema. Ma anche qui, siamo arrivati alla soluzione senza rispettare l'idea dei creatori di questo container. Mettendo nel form della password i seguenti comandi:

- con **ls** ci ha restituito questo:

```
{  
    "output": "app.py\\nrequirements.txt\\n",  
    "password_command": "ls",  
    "username": "asfas"  
}
```

- utilizzando, poi **cat app.py** si è ottenuto la flag scritta in FLAG\_CONTENT.

### 2.1.3 Terza modalità di attacco

A questo punto abbiamo cercato di arrivare alla flag nel modo giusto utilizzando i seguenti comandi bash:

- **ls -a** : ci ha elencato tutti i file e le directories nascoste e non. Notando per l'appunto una directory nascosta **.flag**.

Figure 8: Si nota la flag dentro questo messaggio

- **cd .flag; ls -a** : con questo comando concatenato siamo entrati nella directory nascosta e abbiamo visualizzato tutti i file messi qua dentro.
  - **cd .flag; cat flag.txt**: con questa abbiamo ottenuto la password dell'account(1234).
  - si è digitato la password nella schermata di login e siamo entrati nella pagina /success.

## 2.2 lo spagnolo

### 2.2.1 Prima modalità di attacco

Nella seconda CTF conquistata, abbiamo un pò seguito le orme di quello fatto prima scoprendo come prima cosa i percorsi del sito attraverso il tool gobuster, ottenendo però un risultato diverso. Infatti si è trovato solo un path:

Figure 9: Si ha un solo risultato

Questa console fa riferimento ad una console python interattiva, bloccata però dal pin della modalità debugger che è stata scritta quando il container è stato avviato. Quindi abbiamo visualizzato i path dentro l'applicativo e poi aperto il file app.py dove si è notato un percorso di nome ”/espagna”, che porta alla flag (letteralmente) della Spagna.



### Interactive Console

In this console you can execute Python expressions in the context of the application. The initial namespace was created by the debugger automatically.

```
>>> import os; os.listdir('.')
['docker-compose.yml', 'requirements.txt', 'Dockerfile', 'app.py']
>>> with open('Dockerfile', 'r') as f:
...     print(f.read())
...
app = Flask(__name__)

# Fixed password for demonstration purposes
CORRECT_PASSWORD = "123456"

@app.route('/', methods=['GET', 'POST'])
def login():
    if 'redirect_url' in request.args:
        redirect_url = request.args.get('redirect', None)
        if redirect_url:
            return f"<script>window.location.href='{redirect_url}'</script>"

    # Capture form inputs
    username = request.form.get('username', '')
    password = request.form.get('password', '')
    message = ""

    if request.method == 'POST':
```

Brought to you by DON'T PANIC, your friendly Werkzeug powered traceback interpreter.



### Interactive Console

In this console you can execute Python expressions in the context of the application. The initial namespace was created by the debugger automatically.

```
return page
@app.route('/espana')
def espana():
    return """
<!DOCTYPE html>
<html>
<head>
<title>Espana</title>
<style>
body {
    font-family: Arial, sans-serif;
    background-color: #f1f1f1;
    text-align: center;
    padding: 20px;
}
h1 {
    color: #333;
}
img {
    width: 100px;
    margin-top: 20px;
}
</style>
</head>
```

Brought to you by DON'T PANIC, your friendly Werkzeug powered traceback interpreter.



Welcome to Espana!



Congratulations on finding the hidden page!

Figure 10: Flag trovata

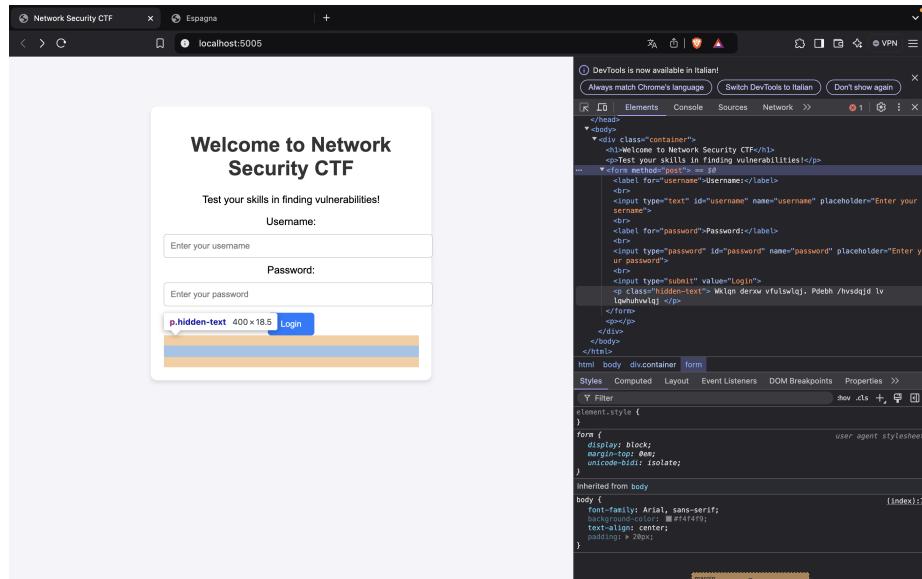
## 2.2.2 Seconda modalità di attacco

Ispezionando la pagina /login, si può notare un **hidden-text** con una scritta incomprensibile.

Questa scritta recitava:

*Wklqn derxw vfulswlqj. Pdebh /hvsdqjd lv lqwhuhvwlgj*

Qui sotto si mostra l'immagine:



Abbiamo quindi provato con il cifrario di Cesare e ci abbiamo azzeccato. Infatti viene fuori una frase del genere:

*"Think about scripting. Maybe /espagna is interesting"*

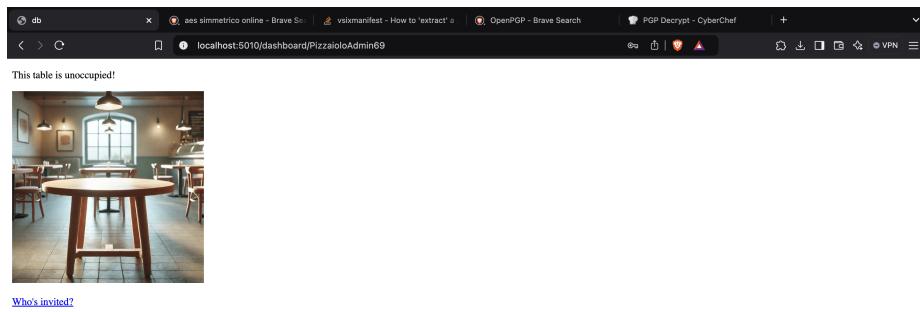
Provando questo percorso, riusciamo ad ottenere il solito risultato di prima (ovvero la flag).

## 2.3 Salsiccia e Friarielli

Come ultimo CTF da provare a conquistare si ha quello del gruppo di Salsiccia e Friarielli che è basata su SQL Injection e Steganografia. Dove quest'ultima è la pratica di nascondere le informazioni all'interno di un altro messaggio per evitare che vengano individuate.

### 2.3.1 Primo tentativo

Per prima cosa, abbiamo provato a inserire una query nella pagina login del container. La query è sempre la solita, ovvero abbiamo messo nel campo utente la stringa: '' OR '1'='1' -- mentre nel campo password una qualsiasi stringa. Premendo invio si entra su una pagina strana, formata in questo modo:



Si è quindi scaricata questa immagine e abbiamo provato a vedere se vi fosse nascosta qualche informazione. Le seguenti operazioni che abbiamo fatto su questa foto sono:

- abbiamo controllato le informazioni exif della foto (attraverso **exiftool**).
- abbiamo utilizzato **binwalk** per scansionare file e vedere se fossero presenti dei file compressi e estrarli.
- si è utilizzato **zsteg**. Ovvero uno strumento per la Steganografia nei file png dove si usa per trovare dati nascosti dentro le immagini.
- abbiamo provato ad applicare filtri sulla foto tramite StegOnline. Come filtri Full Red,Green,Blue, Inverse RGB e LSB Half

Nonostante tutti questi test, non siamo riusciti a trovare niente. Quindi abbiamo deciso, di guardare nel file "app.py" i percorsi possibili dell'applicativo accedendo dalla console interattiva( come si è fatto per gli attacchi precedenti). Facendo così abbiamo trovato le seguenti credenziali degli account:

```

c = conn.cursor()
# Creazione della tabella se non esiste
c.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY,
    username TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    dashboard_file TEXT NOT NULL
)
''')
# Inserimento degli utenti (4 vulnerabili e uno sicuro)
users = [
    (1, 'PizzaioloAdmin69', 'SfGfathewin', 'db_utenti_steg.html'),
    (2, 'HINT1', 'HINT1', 'HINT1.html'),
    (3, 'Omero1', 'Omero1', 'Omero1.html'),
    (4, 'Amavoigairam', 'Amavoigairam', 'Amavoigairam.html'),
    (5, 'HINT@02', 'HINT@02', 'hINT@02.html'),
    (6, 'brain', 'heart', 'symCrypt_AES.html')
]
# Salva i cambiamenti
c.executemany("INSERT OR IGNORE INTO users (id, username, password, dashboard_file) VALUES (?, ?, ?, ?)", users)
conn.commit()

```

Accedendo, quindi, ad ogni account segnato si sono ottenuti le seguenti pagine html.



Figure 11: Pensiamo che la hint fosse sulla l'SQL Injection fatta all'inizio

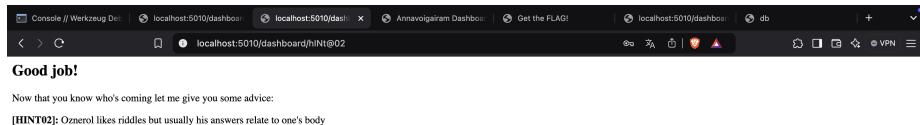


Figure 12: Questa hint era per farci capire che Oznerol era l'invitato della pagina

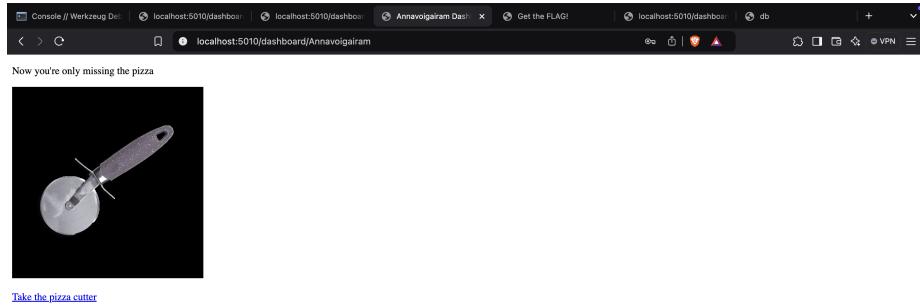


Figure 13: Si ha un'altra foto sospetta

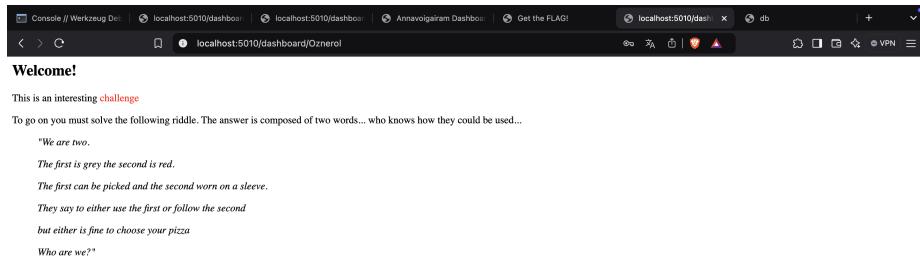


Figure 14: La risposta a questo indovinello è cervello e cuore

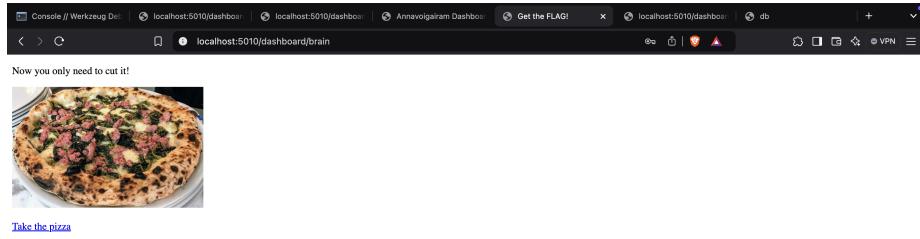


Figure 15: Si accede mettendo le risposte in inglese dell'indovinello precedente

Abbiamo trovato due foto e analizzando le seguenti immagini, si è notato che c'erano delle informazioni nascoste. Si è quindi provveduto a fare lo zsteg su queste foto ottenendo il seguente output per la foto sulla pizza:

```
imagedata      .. text: "qpka_vqmrqlld\\\_XabZef^libnicslfvmhwifod``\\S^[
R`]Tc`WdaZc`YdaZhe^fc^c`[`\\Y^ZW[ZVZYUXWUWVR00GPPFQQGRRHOOCHH<B@4<:.<.:;/=;/
;=/<:.<:.<8,C;0C9-B8,A7+B8,E;/I=1K?3K>5L?6PB9SE<SE<RD;SD=TE>UD<K=4G810B:3(`\
90)>5.B;3<5+>8,@:.A;+B;+@9'?7$=6#71"
b1,bgr,lsb,xy   .. text: "6nfWrn/y"
b2,r,lsb,xy     .. file: VISX image file
b2,g,lsb,xy     .. file: VISX image file
b2,g,msb,xy     .. file: SoftQuad DESC or font file binary - version 13989
b2,b,msb,xy     .. file: VISX image file
b2,rgb,msb,xy   .. file: OpenPGP Public Key
b2,bgr,lsb,xy   .. file: OpenPGP Secret Key
b3,g,msb,xy     .. text: "|vN_-P$I"
b3,b,lsb,xy     .. text: "{ Y<0jtV"
b4,r,lsb,xy     .. file: VISX image file
b4,b,msb,xy     .. file: VISX image file
```

Dopo aver deciso di ignorare i file `visx` e `softquad`, prendendoli per false positive, abbiamo provato a estrarre le chiavi pgp, che, a detta di `zsteg`, erano presenti nel file, per fare questo abbiamo fatto

```
zsteg -E b2,rgb,msb,xy > public
zsteg -E b2,bgr,lsb,xy > secret
```

Un controllo con `file` ha confermato che queste fossero chiavi pgp:

```
$ file public
public: OpenPGP Public Key
$ file secret
secret: OpenPGP Secret Key
```

Date queste chiavi si sarebbe potuto quindi decifrare un qualche payload. Tra i dati ritrovati ,che potessero sembrare un payload cifrato, abbiamo scelto come candidati l'`imagedata`:

```
"qpkavqmrqlldd\_\_`XabZef^libnicslfvmhwifod``\\S^ [R`]Tc`WdaZc`YdaZhe^fc^c` [``\\
Y^ZW[ZVZYUXWUWVROOGPPFQQGRRHOOCHH<B@4<.:.=;/=;/=;/<.:.<8,C;0C9-B8,A7+B
8,E;/I=1K?3K>5L?6PB9SE<SE<RD;SD=TE>UD<K=4G810B:3(\"90)>5.B;3<5+>8,@: .A;+B;+09
'?'7$=6#71"
```

riportato da `zsteg` nel file flag `5ym_AES_f14g.png`, e l'`imagedata`:

```
"wq}ukxunztlxwp}ys"
```

riportato da `zsteg` nel file key `5ym_AES_k3y.png`. Dati questi candidati si è quindi provato a usare le chiavi ricavate per vedere se potessero essere decifrati con un output sensato, si è quindi provato a importare le chiavi

```
gpg --import public
gpg --import private
```

Purtroppo, gpg non ha riconosciuto le chiavi come tali, portando a errore in entrambi i casi.

Non abbiamo trovato altro modo per utilizzare le chiavi, e ci siamo fermati.

### 2.3.2 Secondo tentativo - Utilizzo di OpenStego

Tramite un indizio fornito dal gruppo, abbiamo scaricato un software in grado di estrarre dalle foto qualcosa di più leggibile rispetto a quello che abbiamo provato fino ad adesso.

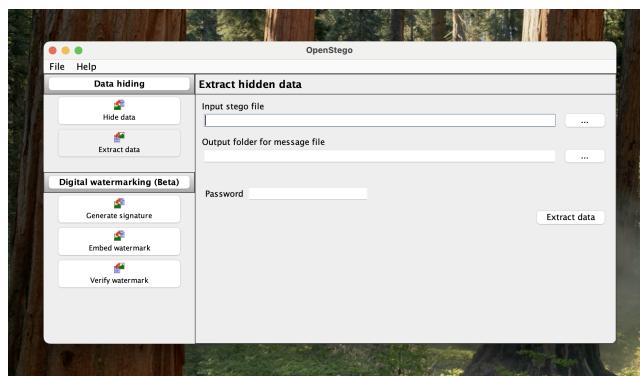


Figure 16: OpenStego è un software per estrarre dati da file

Questo software si chiama OpenStego e grazie ad esso abbiamo estratto dalle due immagini due stringhe:

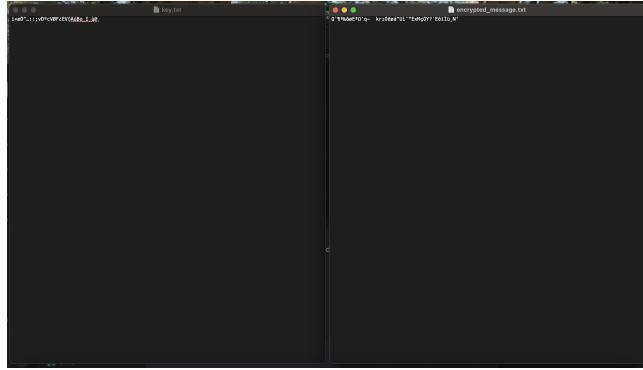


Figure 17:

Non siamo riusciti a decodificare il messaggio fino a quando non ci è stato detto che dovevamo creare uno script Python anziché utilizzare tool preesistenti. Di conseguenza, abbiamo sviluppato il seguente script in Python:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

def AESDecrypt(ciphertext_file, key_file, output_file):
    with open(key_file, 'rb') as f:
        key = f.read()

    with open(ciphertext_file, 'rb') as f:
        iv = f.read(16)
        ciphertext = f.read()

    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted = unpad(cipher.decrypt(ciphertext), AES.block_size)

    with open(output_file, 'wb') as f:
        f.write(decrypted)

def main():
    ciphertext_file = '/Users/marcodestefano/Desktop/cartella senza nome/encrypted_message.txt'
    key_file = '/Users/marcodestefano/Desktop/cartella senza nome/key.txt'
    output_file = 'decrypted_message.txt'

    AESDecrypt(ciphertext_file, key_file, output_file)
    print("Successo, controlla cartella ->", output_file)
```

```
if __name__ == "__main__":
    main()
```

Inizialmente avevamo posizionato i file da decriptare nella stessa directory dello script, ma l'esecuzione restituiva errori di padding. Abbiamo quindi rimosso gli spazi dai nomi delle cartelle, ma il problema persisteva. A quel punto, abbiamo rigenerato i due file .txt con OpenStego in una nuova cartella e li abbiamo aperti utilizzando il percorso completo. Finalmente, siamo riusciti a ottenere la nostra flag:

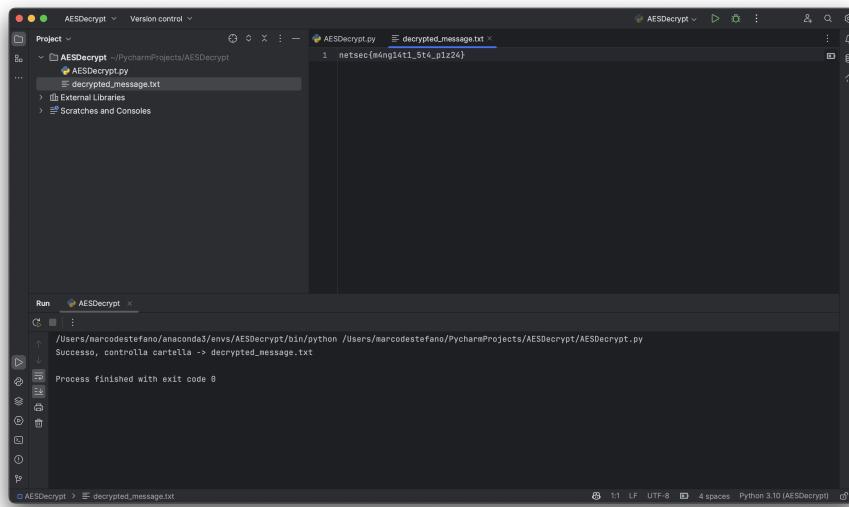


Figure 18: Flag: netsec{m4ng14t1\_5t4\_p1z24}

## 2.4 Conclusioni

Abbiamo scelto queste CTF perché risultano interessanti e rapide da attaccare. Ad esempio, il CTF del gruppo “lo spagnolo” è stato catturato in poco tempo mentre particolarmente coinvolgente è stato il CTF di ”LANgo Unchained”, che presentava una vulnerabilità nel campo password, permettendo di esplorare tutte le risorse del container tramite comandi bash.

Infine, abbiamo selezionato “Salsiccia e Friarielli” come ultima CTF, ritenendola inizialmente più semplice da catturare. Tuttavia, riteniamo che la difficoltà indicata nel documento Excel non sia stata rispettata, poiché gli indizi per la cattura ci sono sembrati poco chiari. Nonostante ciò, è stato interessante e stimolante scoprire come un’immagine possa nascondere informazioni di quel tipo.

### **3 Contributi**

Nel seguente capitolo, vedremo nella nostra squadra chi ha fatto cosa.

- Habib Kirolos (7060226): ha contribuito nella realizzazione front-end back-end del nostro applicativo. Facendo anche il deploy del nostro container. Inoltre ha scritto alcuni paragrafi della relazione e ha dato una mano nel CTF del gruppo Salsiccia e Friarielli.
- Regino Kamberaj (7164400): ha scritto metà della relazione e ha contribuito alla CTF di LANgo Unchained.
- Marco De Stefano (7025991): ha contribuito in tutti gli attacchi CTF(LANgo Unchained, Salsiccia e Friarielli e lo spagnolo), ha scritto l'altra metà della relazione e ha contribuito alla realizzazione del nostro applicativo.