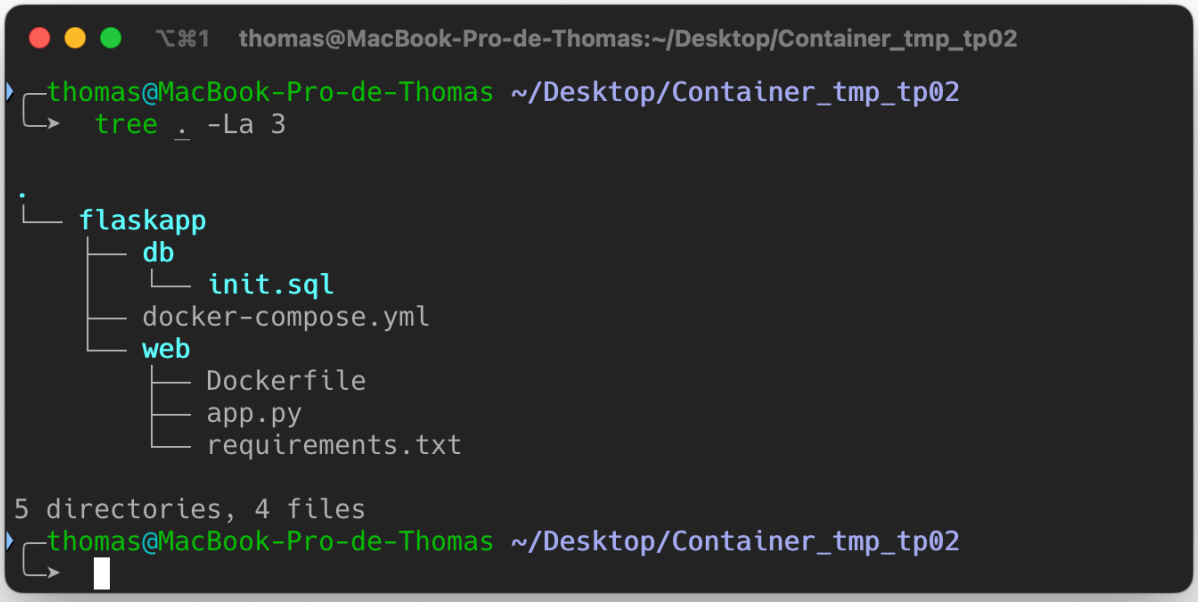
Rendu TP02

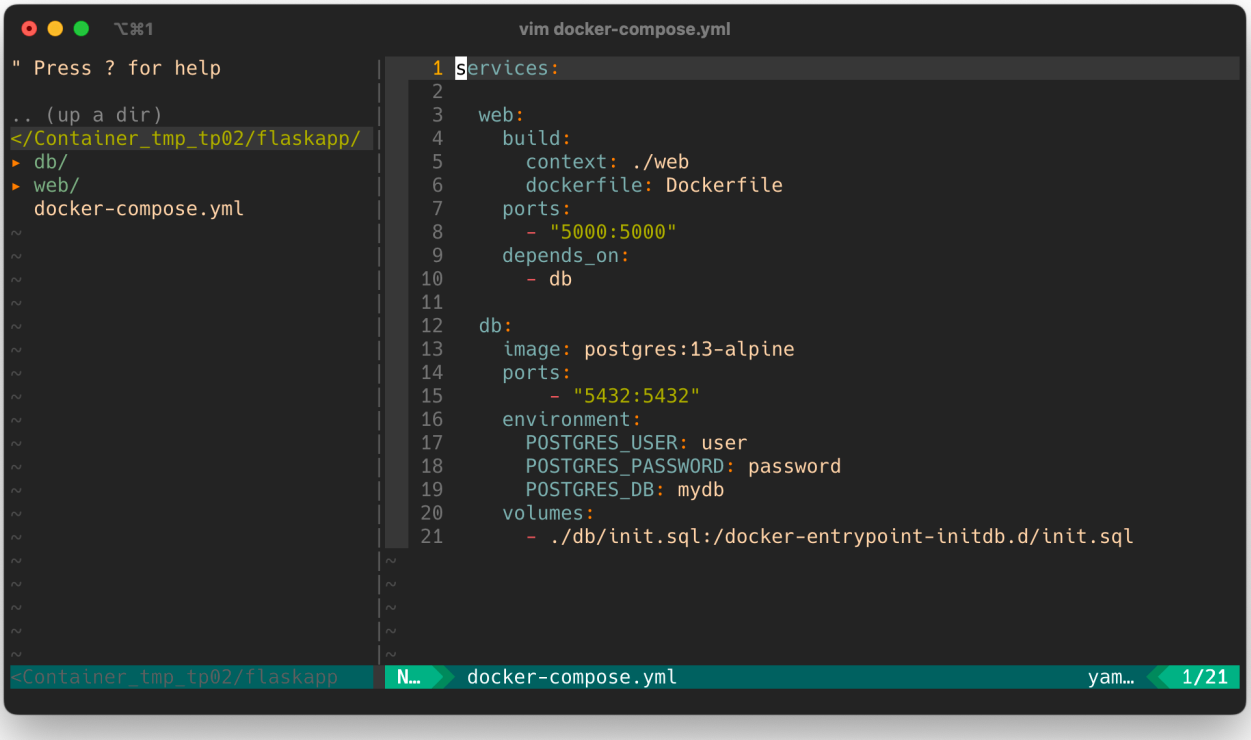
Rendu du TP02 par `Vincent LAGOGUE` et `Thomas PEUGNET`.

Mise en place de l'environnement de travail

Nous créons notre environnement et obtenons l'architcture suivante.



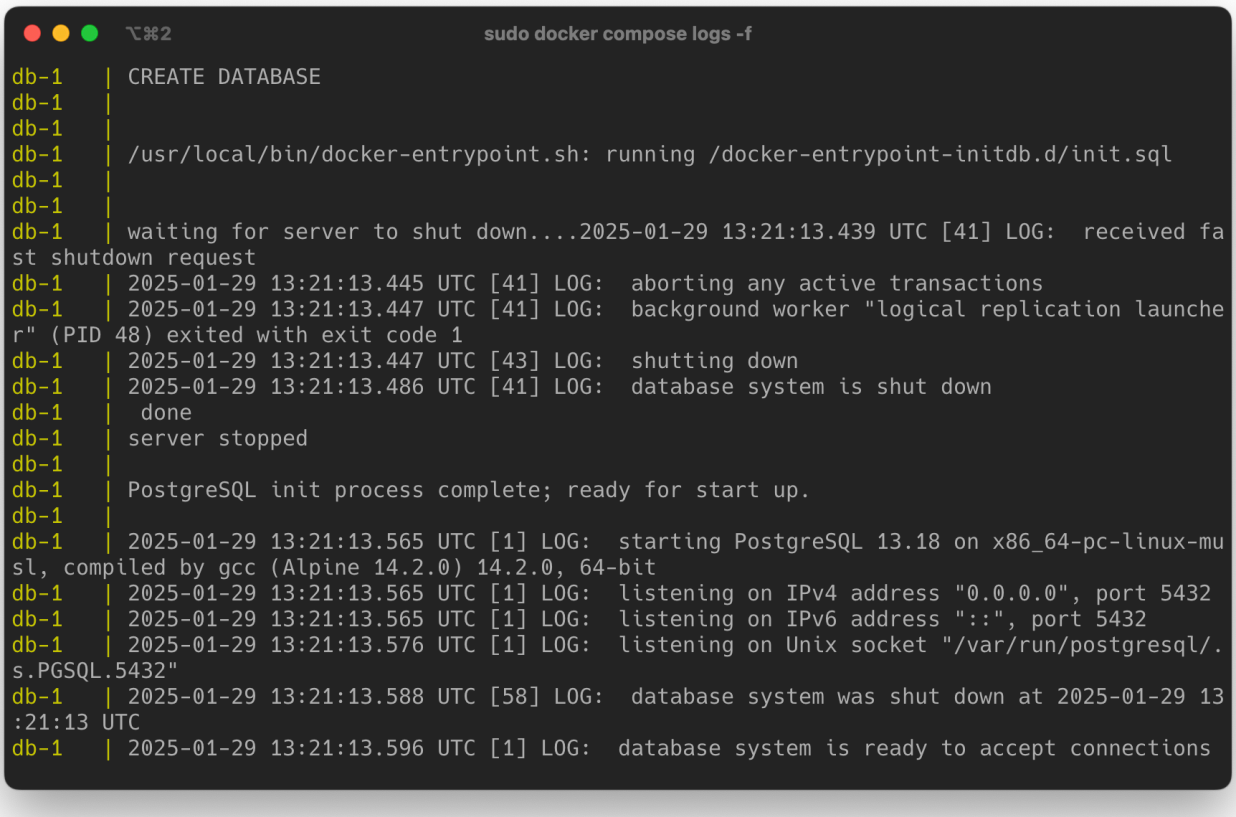
Notre `docker-compose.yml` a le contenu suivant.



Nous lançons notre application avec le démarrage de nos conteneurs.

```
docker compose up -d --build
```

Nous constatons que l'application tourne correctement et que la base de données fonctionne correctement.



Nous modifions notre fichier `init.sql` pour avoir le contenu suivant. Ce contenu a été généré volontairement par Mistra AI.

```sql
-- Empty initialization file for now
-- Création de la base de données si elle n'existe pas
-- CREATE DATABASE mydb;
\c mydb;

-- Création de la table des utilisateurs avec une vulnérabilité classique (pas de préparation de requête)
DROP TABLE IF EXISTS users;
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    password VARCHAR(255) NOT NULL,  -- Mauvaise pratique : stockage en clair
    email VARCHAR(100),
    role VARCHAR(20) DEFAULT 'user'
);

-- Insertion de quelques utilisateurs
INSERT INTO users (username, password, email, role) VALUES
('admin', 'admin123', 'admin@example.com', 'admin'),
('user1', 'password', 'user1@example.com', 'user'),
('user2', '123456', 'user2@example.com', 'user'),
('evil_hacker', 'p@ssw0rd', 'hacker@darkweb.com', 'user');
```

```sql
-- Table des logs (idéal pour tester l'injection dans les recherches)
DROP TABLE IF EXISTS logs;
CREATE TABLE logs (
    id SERIAL PRIMARY KEY,
    action TEXT,
    user_id INT REFERENCES users(id),
    timestamp TIMESTAMP DEFAULT NOW()
);

-- Exemples d'entrées de logs
INSERT INTO logs (action, user_id) VALUES
('User admin logged in', 1),
('User user1 changed password', 2),
('User user2 attempted login', 3),
('Evil hacker tried SQL injection', 4);
```

Nous modifions également notre fichier `app.py` pour avoir le contenu suivant.

```python
from flask import Flask, request, jsonify
import psycopg2
import logging

logging.basicConfig(filename='/app/logs/access.log', level=logging.INFO)

app = Flask(__name__)

# Connexion à PostgreSQL (⚠️ vulnérable car pas de paramètre sécurisé)
DB_CONFIG = {
    "dbname": "mydb",
    "user": "user",
    "password": "password",
    "host": "db",
    "port": 5432
}

def get_db_connection():
    return psycopg2.connect(**DB_CONFIG)

@app.route('/')
def home():
    return "🚀 API Flask vulnérable aux injections SQL"

# 🔥 1️⃣ Authentification vulnérable 🔥
@app.route('/login', methods=['POST'])
def login():
    data = request.json
    username = data.get('username', '')
    password = data.get('password', '')
```

```
    conn = get_db_connection()
    cursor = conn.cursor()

    # ⚠️ Vulnérable aux injections SQL
    query = f"SELECT id, username, role FROM users WHERE username = '{username}' AND
password = '{password}'"
    logging.info(f"[DEBUG] Query exécutée: {query}")  # Pour voir ce qui est injecté
    cursor.execute(query)
    user = cursor.fetchone()

    cursor.close()
    conn.close()

    if user:
        return jsonify({"message": "Connexion réussie", "user": user}), 200
    return jsonify({"message": "Échec de l'authentification"}), 401

# 🔥 2️⃣ Recherche d'utilisateurs vulnérable 🔥
@app.route('/search', methods=['GET'])
def search_users():
    search = request.args.get('q', '')

    conn = get_db_connection()
    cursor = conn.cursor()

    # ⚠️ Vulnérable aux injections SQL
    query = f"SELECT id, username, email FROM users WHERE username LIKE '%{search}%'"
    logging.info(f"[DEBUG] Query exécutée: {query}")
    cursor.execute(query)
    results = cursor.fetchall()

    cursor.close()
    conn.close()

    return jsonify(results), 200

# 🔥 3️⃣ Affichage des logs vulnérable 🔥
@app.route('/logs', methods=['GET'])
def get_logs():
    filter_log = request.args.get('filter', '')

    conn = get_db_connection()
    cursor = conn.cursor()

    # ⚠️ Vulnérable aux injections SQL
    query = f"SELECT id, action, timestamp FROM logs WHERE action LIKE '%
{filter_log}%'"
    logging.info(f"[DEBUG] Query exécutée: {query}")
```

```
    cursor.execute(query)
    logs = cursor.fetchall()

    cursor.close()
    conn.close()

    return jsonify(logs), 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

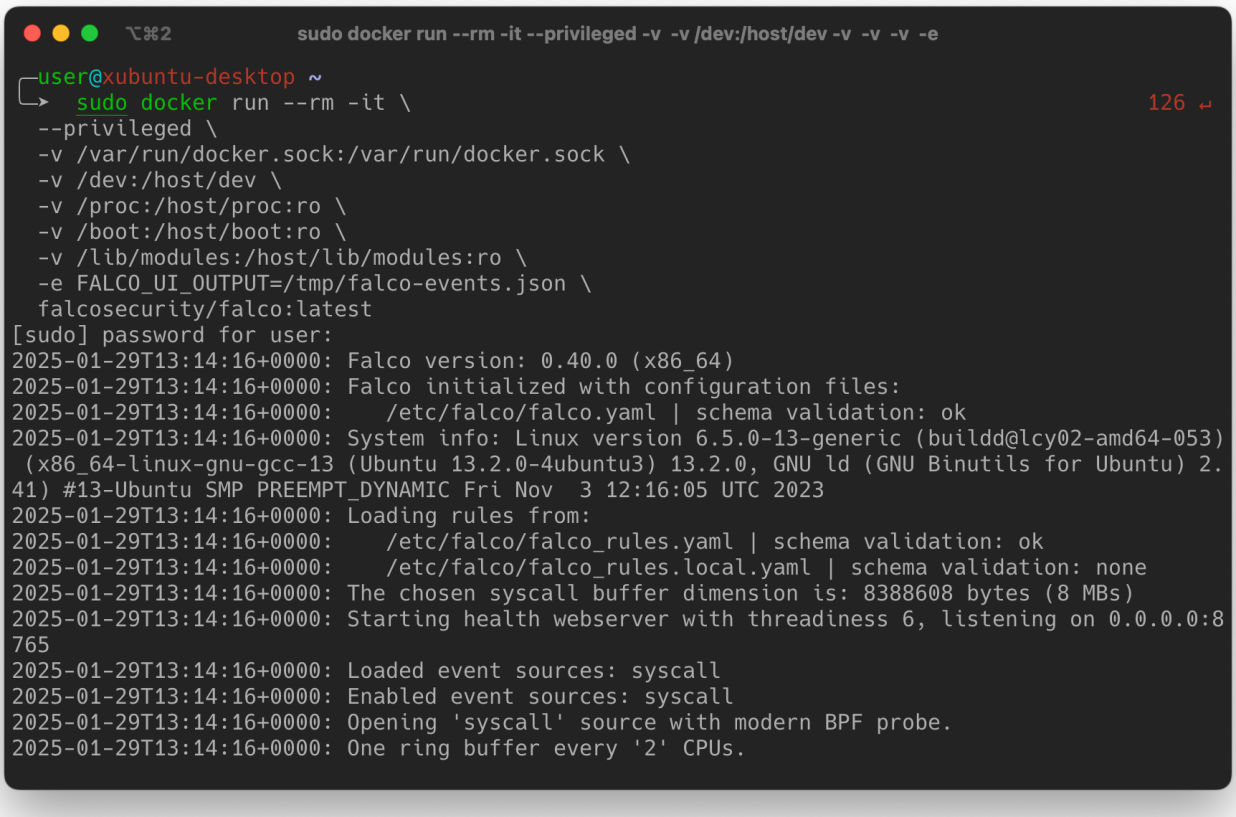
Falco

Nous lançons Falco à l'aide de la commande suivante.

```
docker run --rm -it \
  --privileged \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v /dev:/host/dev \
  -v /proc:/host/proc:ro \
  -v /boot:/host/boot:ro \
  -v /lib/modules:/host/lib/modules:ro \
  -e FALCO_UI_OUTPUT=/tmp/falco-events.json \
  falcosecurity/falco:latest
```

Nous remarquons que, après plusieurs tentatives, `falco` ne peut pas fonctionner sur un système macOS. Nous nous connectons donc à une VM sur un hyperviseur Proxmox pour continuer la suite du TP.

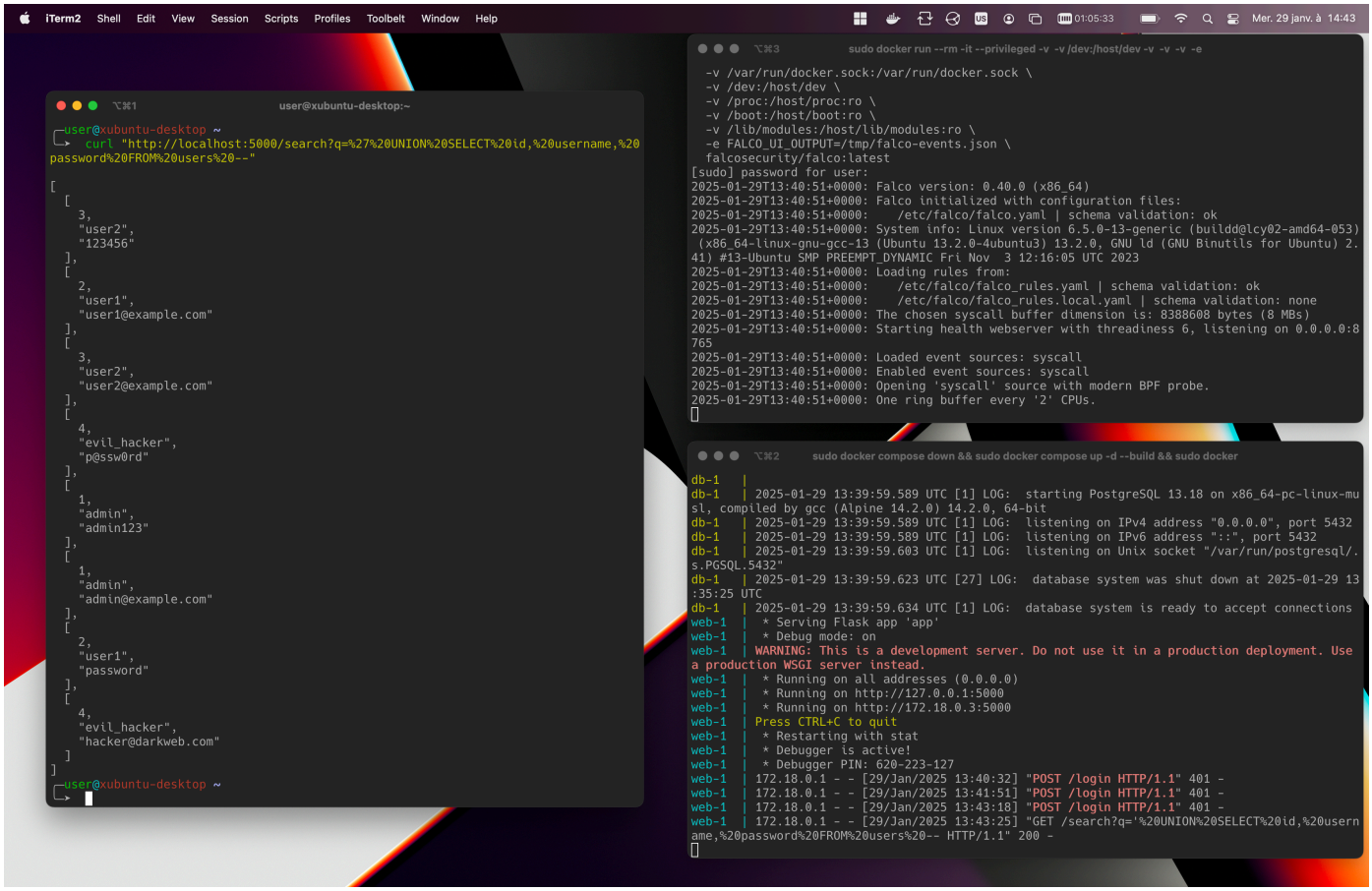
Nous relançons la commande Docker.



Attaques

Injection SQL via HTTP

Nous lançons une première attaque, étant la requête `curl` suivante. Cette attaque fait une requête SQL au travers d'une requête POST faite à notre serveur web.

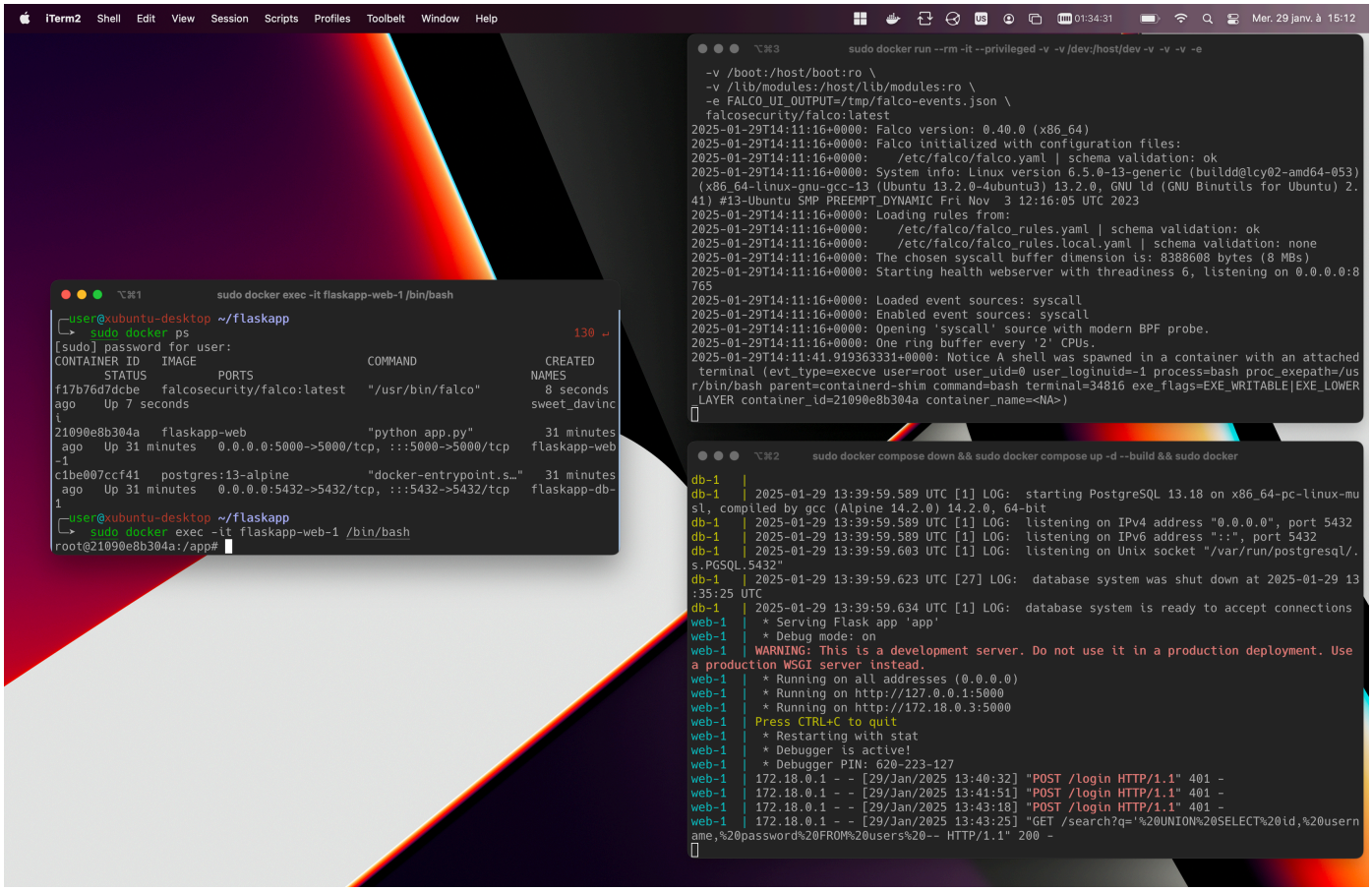


Nous constatons que l'attaque n'a pas été détectée par Falco (absence de changement dans les logs, fenêtre en haut à droite).

Connexion en shell

Nous effetuons maintenant une connexion via un `docker exec` sur le conteneur de notre application web à l'aide de la commande suivante.

```
sudo docker exec -it flaskapp-web-1 /bin/bash
```



Nous constatons cette fois-ci que cette manipulation a bien été détectée par Falco, nous avons le résultat suivant.

```
2025-01-29T14:11:41.919363331+0000: Notice A shell was spawned in a container with an attached terminal (evt_type=execve user=root user_uid=0 user_loginuid=-1 process=bash proc_exepath=/usr/bin/bash parent=containerd-shim command=bash terminal=34816 exe_flags=EXE_WRITABLE|EXE_LOWER_LAYER container_id=21090e8b304a container_name=<NA>)
```

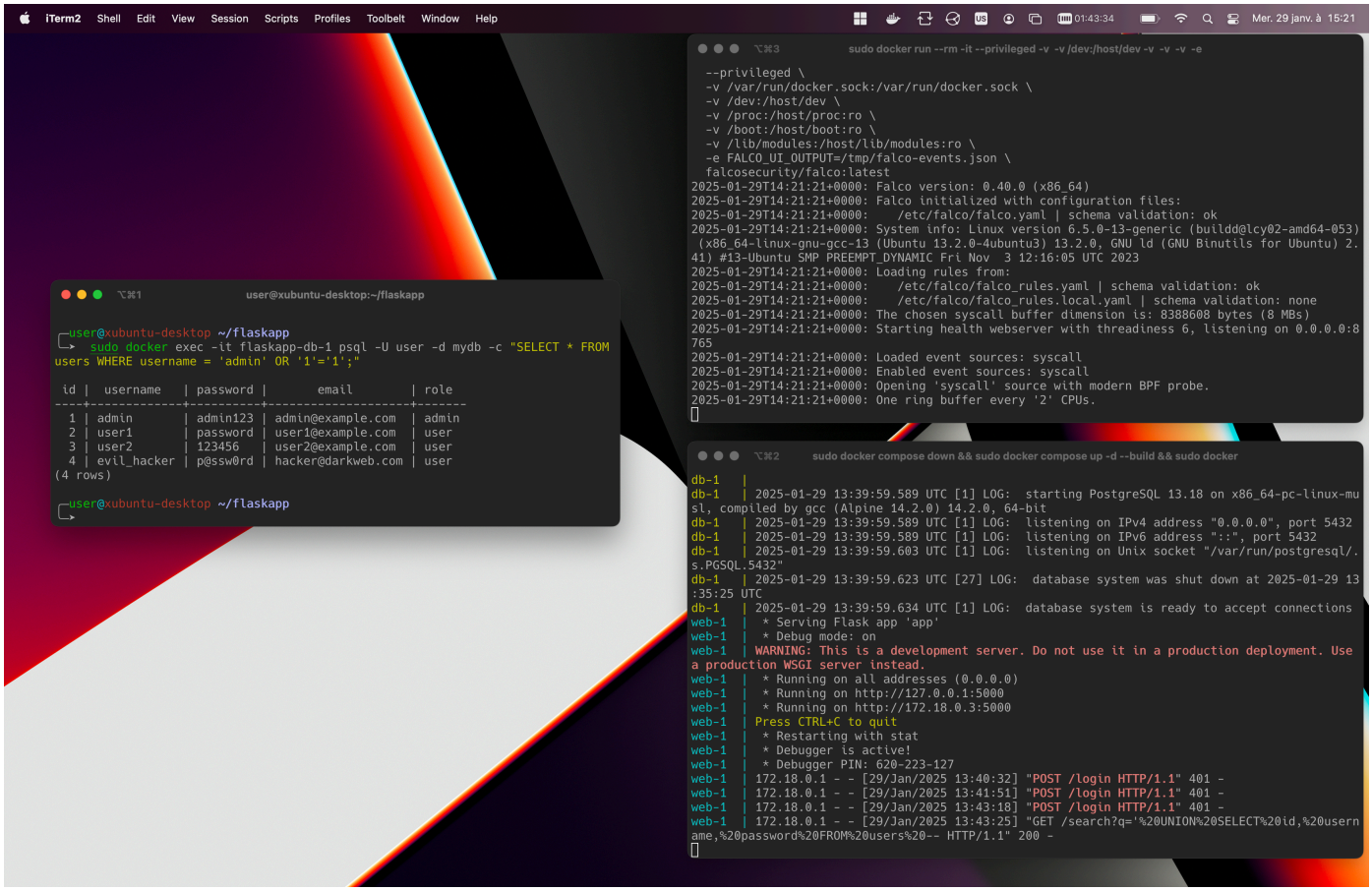
Un shell (`bash`) a été lancé à l'intérieur d'un conteneur.

- L'utilisateur `root` (`user=root user_uid=0`) a exécuté cette commande, ce qui indique un accès avec des privilèges root.
- Le processus parent est `containerd-shim`, ce qui signifie que le shell a été lancé directement via Docker ou un runtime de conteneur.
- Le terminal est attaché (`terminal=34816`), ce qui veut dire que la session est interactive.
- Le fichier exécutable (`proc_exepath=/usr/bin/bash`) est situé dans `/usr/bin/` .

Injection SQL via un Shell

Nous tentons une nouvelle façon de faire une injection, avec une commande `docker exec` .

```
sudo docker exec -it flaskapp-db-1 psql -U user -d mydb -c "SELECT * FROM users WHERE username = 'admin' OR '1'='1';"
```



Nous constatons là encore que l'attaque n'est pas détectée, malgré une réussite évidente (liste de tous les users et leur mot de passe).

Conclusion

Il semblerait donc que Falco, en fonctionnant avec ses règles par défaut ne soit pas capable de détecter des injections SQL passant par une requête HTTP.

Nous avons longuement investigué sur la façon de gérer les règles pour améliorer ses capacités de détection, mais avons constaté que c'était le programme du TP03.

Nous ajoutons simplement une note ici, permettant de monter notre fichier de règles depuis notre host vers notre conteneur Falco.

```
sudo docker run --rm -it \
  --privileged \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v /dev:/host/dev \
  -v /proc:/host/proc:ro \
  -v /boot:/host/boot:ro \
  -v /lib/modules:/host/lib/modules:ro \
  -v /home/user/flaskapp/falco_rules.local.yaml:/etc/falco/falco_rules.local.yaml:ro \
  -v /home/user/flaskapp/logs/access.log:/var/log/falco/flask_access.log:ro \
  -e FALCO_UI_OUTPUT=/tmp/falco-events.json \
  falcosecurity/falco:latest
```

À noter que la suite de nos tentatives étaient de rediriger l'output des logs de notre application Flask vers un fichier à lire par Falco lors de la création et du lancement du conteneur. Ensuite, nous aurions fait une règle analysant le contenu de notre fichier de log et vérifiant si une regex est validée par une des lignes du fichier de logs.

D'après la capture d'écran suivante, il ne serait pas forcément très difficile de détecter ce genre d'intrusion.