Large-Scale Software Architecture

Laboratory 3 - Security

1. Objective

The objective of this lab is to demonstrate how applying the **Limit Exposure** security tactic using the **API Gateway** architectural pattern significantly reduces the system's attack surface. This is a core principle of **Secure by Design** — building security into the architecture from the start.

2. Architectural Overview

a. Secure Architecture (With API Gateway)

Client → API Gateway (exposed) → Microservice (hidden) → Database (hidden)

Only the API Gateway is exposed to the public. All other components are hidden behind the gateway, reducing exposure and enforcing centralized access control.

b. Insecure Architecture (Without API Gateway)

Client → Microservice (exposed) → Database (hidden)

The microservice is directly accessible from the internet. This increases the attack surface and violates the Limit Exposure principle.

c. Security Tactic: Limit Exposure

What is Limit Exposure?

The **Limit Exposure** tactic focuses on **reducing the visibility and accessibility** of sensitive components. This is achieved by controlling who can interact with the system and which parts of the system are exposed. The core idea is to **hide** or **restrict** access to critical components, preventing unauthorized services or users from interacting with them.

Advantages of Limit Exposure:

- **Reduced Attack Surface**: By minimizing the points of entry, we limit the number of vectors an attacker can use.
- **Granular Access Control**: Only authorized services or users can interact with sensitive components.
- **Better Compliance**: Restricting access helps in meeting regulatory compliance (e.g., GDPR, HIPAA).

3. Prerequisites

Ensure you have the following installed:

- **Python 3.x**
- **Flask** (`pip install flask`)
- **PyJWT** (`pip install pyjwt`)

You should also have a basic understanding of:

- **Microservice architecture**
- **API Gateway** concepts
- **JSON Web Tokens (JWT)** for authentication

4. Instructions

Step 1: Create the API Gateway

The **API Gateway** will act as the entry point to your system. It will control access to the microservice and the database by enforcing authentication and IP restrictions.

api_gateway.py

```python
from flask import Flask, request, jsonify
import jwt
from functools import wraps

app = Flask(__name__)
SECRET_KEY = "your_secret_key"
AUTHORIZED_IP = "127.0.0.1"  # Only allow local access for simplicity

# Mock users for the sake of the example
USERS = {
    "user1": "password123"
}

# Function to check if the request comes from an authorized IP address
def limit_exposure(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        client_ip = request.remote_addr
        if client_ip != AUTHORIZED_IP:
            return jsonify({'message': 'Forbidden: Unauthorized IP'}), 403
        return f(*args, **kwargs)
    return decorated_function

# Function to check JWT token
def token_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        token = request.headers.get('Authorization')
        if not token:
            return jsonify({'message': 'Token is missing!'}), 403
        try:
            jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
        except:
            return jsonify({'message': 'Token is invalid!'}), 403
        return f(*args, **kwargs)
    return decorated_function
```

```
# Route for user login (returns JWT token)
@app.route('/login', methods=['POST'])
def login():
    auth = request.get_json()
    username = auth.get('username')
    password = auth.get('password')
    if USERS.get(username) == password:
        token = jwt.encode({'username': username}, SECRET_KEY,
algorithm="HS256")
        return jsonify({'token': token})
    return jsonify({'message': 'Invalid credentials'}), 401

# Protected route
@app.route('/data', methods=['GET'])
@token_required
@limit_exposure  # Apply the limit exposure tactic to this route
def get_data():
    return jsonify({'message': 'Data accessed successfully!'}), 200

if __name__ == "__main__":
    app.run(debug=True)
```

Explanation:

- The `limit_exposure` decorator ensures that only requests from the local IP (`127.0.0.1`) are allowed to access the protected route.
- The `token_required` decorator checks if a valid JWT is provided in the `Authorization` header before granting access to the `/data` route.

Step 2: Create the Microservice (MS)

The **Microservice (MS)** will be a simple service that performs some business logic. It will be protected by the API Gateway, ensuring that it cannot be accessed directly without the proper authentication and IP restrictions.

microservice.py

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/microservice')
def microservice():
    return jsonify({'message': 'This is a secure microservice'}), 200

if __name__ == "__main__":
    app.run(debug=True, port=5001)
```

Step 3: Create the Database (DB)

For this laboratory, we will simulate a simple **Database (DB)**. In a real-world application, this would be a full-fledged database, but here we will just mock the behavior of a database with a protected route.

`database.py`

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/db')
def db_access():
    return jsonify({'message': 'Database access granted'}), 200

if __name__ == "__main__":
    app.run(debug=True, port=5002)
```

Step 4: Test the Architecture

Start all services

Run the following commands to start each service in separate terminals:

1. **API Gateway**:

   ```
   python api_gateway.py
   ```

   The API Gateway will run on port `5000`.

2. **Microservice**:

   ```
   python microservice.py
   ```

   The microservice will run on port `5001`.

3. **Database**:

   ```
   python database.py
   ```

   The database will run on port `5002`.

Obtain JWT Token

Use the `POST /login` endpoint to log in and get a JWT token.

```
curl -X POST -H "Content-Type: application/json" -d '{"username": "user1",
"password": "password123"}' http://127.0.0.1:5000/login
```

This will return a token that will be used to access protected routes.

Access Protected Routes

Use the `GET /data` endpoint in the API Gateway with the JWT token in the `Authorization` header.

```
curl -X GET -H "Authorization: Bearer <your_token>"
http://127.0.0.1:5000/data
```

Limit Exposure

- If you try to access the API Gateway or microservice from an unauthorized IP (e.g., not from `127.0.0.1`), you will get a `403 Forbidden` error.

Conclusion

By incorporating **Limit Exposure** into the architecture, we have:

1. **Reduced the attack surface** by controlling who can access sensitive components.
2. Implemented **JWT-based access control** to ensure that only authorized users can interact with the system.
3. Used **IP restrictions** to further limit which clients can access the system.

These measures, when applied at the design stage (Secure by Design), provide significant advantages over applying them post-implementation. This proactive approach prevents vulnerabilities and reduces the risk of attacks.

---

5. Delivery

5.1. Deliverable

- Full name.
- The same excercise with the following improvement: expand this design by adding additional services and more complex rules for limiting exposure.

5.2. Submission Format

- The deliverable must be submitted via GitHub (Issa2025i repository).
- Steps:
    - Use the branch corresponding to your team (team1, team2, ...).

- In the folder laboratories/laboratory_3, create an **X** folder (where X = your identity document number), which must include the **deliverable**:
  - README.md with the full name and steps for executing the exercise.
  - Related files to execute.

5.3. Delivery Deadline

Friday, April 25, 2025, before 23h59.