Assignment 3 - User Authentication

100 Oppnåelige Poeng

21.10.2025

Forsøk 1

Pågår

NESTE: Lever oppgave

Legg til kommentar

Ubegrenset antall forsøk tillatt

7.10.2025 til 21.10.2025

Detaljer

[Assignment on Notion (https://perara.notion.site/Assignment-User-Authentication-148911874ac748d5b3bc21780872553b?pvs=4)](https://perara.notion.site/Assignment-User-Authentication-148911874ac748d5b3bc21780872553b?pvs=4)

Objectives

<aside> 🐍 You are free to develop your solution using any language, but I recommend Python. **You can extend your previous application from the first assignment. But I recommend using GIT (if you did not do that the first assignment, now is the time to push everything on git)**

</aside>

Design and implement a user authentication system that combines OAuth2 Authorization Code Flow, conventional authentication mechanisms, a backend database, and security features such as Two-Factor Authentication (2FA).

The assignment consists of 5 tasks, where each task is weighted the same:

- Database Integration
- Basic User Authentication
- Protection Against Brute Force Attacks
- Two-Factor Authentication
- Understanding the Concepts of OAuth2

Learning Outcomes

From this assignment, you will learn about basic databases, like JSON and SQLite. You will learn how to use them securely, having a security-first mindset. You will further learn how to create basic username-password authentication and use hashing to protect credentials. Further, you will create measures to prevent rainbow table attacks from working with rate-limiting. Then we will discover how two-factor authentication works and how it can be implemented with Python. Lastly, we will learn the basics of OAUTH2.

(20) **Database Integration:**

- Integrate a lightweight database, e.g., JSON-based storage or SQLite, to persistently save user data.
- Design efficient database schemas that optimize retrieval and storage operations while ensuring data security.

(20) Basic **User Authentication:**

<aside> 💡 Remember to document security challenges and mitigations

</aside>

- Set up a standard authentication system that allows users to sign up using a username and password.
- Store user credentials securely in the database, leveraging advanced hashing and salting techniques, preferably with libraries like `bcrypt` or `hashlib`.

(20) **Protection Against Brute Force Attacks:**

<aside> 💡 Remember to document security challenges and mitigations

</aside>

- Embed a robust rate-limiting mechanism in the system to discourage repetitive password guess attempts.
- Impose a mandatory time-out after three consecutive failed login attempts.

Note: You can test this by attempting to access an API endpoint several times without the correct authentication token, or trying to log in with incorrect credentials

(20) **Two-Factor Authentication (2FA):**

<aside> 💡 Remember to document security challenges and mitigations </aside>

- Incorporate a time-based one-time password (TOTP) system for an enhanced security layer following either the OAuth2 or conventional login. Utilize the `pyotp` library.
- Upon registration, generate and display a QR code for users, allowing integration with 2FA apps like Google Authenticator.
- During the login phase, request that the user input the TOTP from their authenticator app.

(20) Understanding the Concepts of **OAuth2:**

<aside> 💡 Remember to document security challenges and mitigations. Also here, you can document the benefits of OAuth

</aside>

- Develop an OAuth2 client using the Authorization Code Flow.
- Fetch and securely store user details from the third-party provider in the database.

Sample Code

```
from flask import Flask, request, redirect, session
import os

app = Flask(__name__)
app.secret_key = os.urandom(24)  # For session handling

# Mock constants for client ID and secret.
CLIENT_ID = "YOUR_CLIENT_ID"
CLIENT_SECRET = "YOUR_CLIENT_SECRET"
REDIRECT_URI = "<http://localhost:5000/callback>"

AUTH_CODES = {}  # Temporary storage for auth codes. Use a proper database in a real-world scenario.
TOKENS = {}      # Temporary storage for access tokens.

@app.route("/auth", methods=["GET"])
def auth():
    """
    Endpoint where the client sends the user to request their authorization.
    After authorization, user is redirected back to the client with an auth code.
    """
    # TODO:
    # 1. Extract 'client_id', 'redirect_uri', 'state', etc. from the request.
    # 2. Validate 'client_id' and 'redirect_uri' against registered client details.
    # 3. Display an authorization page to the user to grant permission.
    # 4. If user grants permission, generate an authorization code.
    # 5. Save the authorization code and associated data.
    # 6. Redirect the user back to 'redirect_uri' with the 'code' and 'state'.
    pass

@app.route("/token", methods=["POST"])
def token():
    """
    Endpoint where the client exchanges the authorization code for an access token.
    """
    # TODO:
    # 1. Extract 'code', 'redirect_uri', 'client_id', 'client_secret' from the request.
    # 2. Verify that the 'code' is valid and has not expired.
    # 3. Validate 'client_id' and 'client_secret'.
    # 4. Generate an access token (and optionally, a refresh token).
    # 5. Save the access token for later validation.
    # 6. Return the access token (and optionally, a refresh token) in a JSON response.
    pass

@app.route("/protected_resource", methods=["GET"])
def protected_resource():
    """
    A protected endpoint the client can access using the access token.
    """
    # TODO:
    # 1. Extract the access token from the request's Authorization header.
    # 2. Validate the access token.
    # 3. If valid, proceed to access the protected resource and return the data.
    # 4. If invalid, return an appropriate error response.
    pass

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

Code to help on the way

The below Python script is sufficient to test that your OAUTH2 authentication flow works as expected. Note that you will have to use the exact route names and variables for this to work; otherwise, you need to adjust them. You are also recommended to take such tests yourself.

```
import requests

# Constants
```

```
CLIENT_ID = "YOUR_CLIENT_ID"
CLIENT_SECRET = "YOUR_CLIENT_SECRET"
REDIRECT_URI = "<http://localhost:5000/callback>"

# Step 1: Simulate getting the Authorization Code
auth_data = {
    'client_id': CLIENT_ID,
    'redirect_uri': REDIRECT_URI
}
response = requests.post("<http://localhost:5000/approve_auth>", data=auth_data)
# Extract auth_code from the response URL. This assumes the code is in the response URL's query parameters.
auth_code = response.url.split("code=")[1].split("&")[0]

# Step 2: Exchange Authorization Code for Access Token
token_data = {
    'code': auth_code,
    'redirect_uri': REDIRECT_URI,
    'client_id': CLIENT_ID,
    'client_secret': CLIENT_SECRET
}
response = requests.post("<http://localhost:5000/token>", data=token_data)
token_info = response.json()
access_token = token_info['access_token']

# Step 3: Access Protected Resource
headers = {'Authorization': f"Bearer {access_token}"}
response = requests.get("<http://localhost:5000/protected_resource>", headers=headers)
print(response.text)
```

Assignment Overview:

Build a secure web-based API using the Flask framework or a similar tool. Prioritize security, user data protection, error handling, and seamless integration with third-party services.

Documentation Requirement:

For each task, detail:

- **Security Challenges:** Identify challenges related to the specific feature.
- **Vulnerabilities & Mitigations:** List potential vulnerabilities and ways to counteract them.

Deliverables:

1. **Repository or Folder:** This should contain code, database schemas, templates, and other vital files.
2. **Report:** Includes:
   - **Architectural Choices:** Why they were made.
   - **Resources:** Libraries, tools, or external resources used and why.
   - **Challenges & Solutions:** Difficulties encountered and how they were resolved.
   - **Recommendations:** Suggestions for further system improvements.

Evaluation Criteria:

1. **Functionality:** How well the system operates and follows specifications.
2. **Security Excellence:** Adherence to top security practices from the course.
3. **Code Quality:** Organization, readability, and documentation of the code.
4. **Innovative Features:** Additional features enhancing user experience or security.
5. **Documentation Depth:** Clarity and thoroughness in the report and code comments.

Guidance:

While striving for a fully functional system, focus on building robust security. Develop with a security-first approach. Best of luck!

Vis vurderingsveiledning

User Authentication		
Kriterier	**Vurderinger**	**Poeng**

Kriterier	Vurderinger					Poeng
Database Integration	Excellent 20 poeng	Good 18 poeng	Satisfactory 16 poeng	Needs Improvements 12 poeng	No marks 0 poeng	/20 poeng
Basic User Authentication	Excellent 20 poeng	Good 18 poeng	Satisfactory 16 poeng	Needs Improvements 12 poeng	No marks 0 poeng	/20 poeng
Protection Against Brute Force Attacks	Excellent 20 poeng	Good 18 poeng	Satisfactory 16 poeng	Needs Improvements 12 poeng	No marks 0 poeng	/20 poeng
Two-Factor Authentication (2FA)	Excellent 20 poeng	Good 18 poeng	Satisfactory 16 poeng	Needs Improvements 12 poeng	No marks 0 poeng	/20 poeng
Understanding the Concepts of OAuth2	Excellent 20 poeng	Good 18 poeng	Satisfactory 16 poeng	Needs Improvements 12 poeng	No marks 0 poeng	/20 poeng

Husk at denne innleveringen vil være tellende for alle i din Project Groups gruppe.

Velg innsendelsestype

Last opp

Mer

Velg en fil å laste opp

eller

Webkamera-bilde

Canvas-filer

□ Denne oppgaveinnleveringen er mitt eget, originale arbeid *

Lever oppgave