**Develop an OAuth-based Authentication System and Attempt to Bypass it using Burp Suite**

Submitted by

**Sambani Kushala -AP22110011206**

**Department of Computer Science and Engineering**

**Specialisation in CyberSecurity**

**School of Engineering and Sciences**

**A picture containing text

Description automatically generated**

Under the Guidance of

**Dr. Bhaskara Santhosh Egala**

#### **Assistant Professor**

**Department of Computer Science and Engineering**

**SRM University–AP**

**Neerukonda, Mangalagiri, Guntur**

**Andhra Pradesh – 522 240**

**April,2025**

### **Table of Contents**

**Abstract**

1. **Introduction**  
    1.1. Open Authorization (OAuth)……………………………………………….3  
    1.2. Key Roles in OAuth Flow……………………………………………………4  
    1.3. Working of OAuth……………………………………………………………4
2. **Implementation of OAuth-Based Authentication System**   
    2.1. Client-Side Implementation……………………………………………… 5-7  
     2.1.1. Index.html (Login Page) ……………………………………………….6  
     2.1.2. Dashboard.html (User Dashboard) ……………………………………7  
    2.2. Server-Side Implementation (Backend - Express.js) …………………….7-9  
     2.2.1. Authorization Endpoint (/authorize) ………………………………..7  
     2.2.2. Token Endpoint (/token) ………………………………………………8  
     2.2.3. User Info Endpoint (/userinfo) …………………………………….…9
3. **Security Testing and Bypass Simulation Using Burp Suite**   
    3.1. Attack Overview: Token Reuse Attack in OAuth 2.0……………………10  
    3.2. Vulnerabilities Exploited……………………………………………………11  
    3.3. Impact of the Attack…………………………………………………………12  
    3.4. Mitigation Strategies…………………………………………………………12
4. **Conclusion**

**Abstract**

This project focuses on the development of a mock OAuth 2.0-based authentication system designed for a student portal. The implementation simulates an OAuth flow, including authorization, token generation, and protected resource access. Furthermore, the project explores potential security loopholes in OAuth 2.0 by simulating a bypass attack using Burp Suite. The goal is to understand common misconfigurations and weaknesses, and to learn how attackers might exploit them. The outcomes highlight the importance of secure implementation practices in modern authentication mechanisms.

1. **Introduction**

In today’s digital landscape, secure authentication systems are critical to ensuring user privacy and application integrity. OAuth 2.0 has become a widely adopted protocol for delegated authorization. This project aims to build a mock OAuth 2.0 authentication system mimicking a student portal where users log in through a simulated third-party provider.

The project demonstrates the OAuth flow in a simplified web application and emphasizes learning through simulation and security testing. Using tools like Burp Suite, we attempt to bypass the system to explore real-world vulnerabilities such as token manipulation, CSRF attacks, and improper validation. This report outlines the development process, attack simulations, and key takeaways regarding secure OAuth integration.

* 1. **Open Authorization**

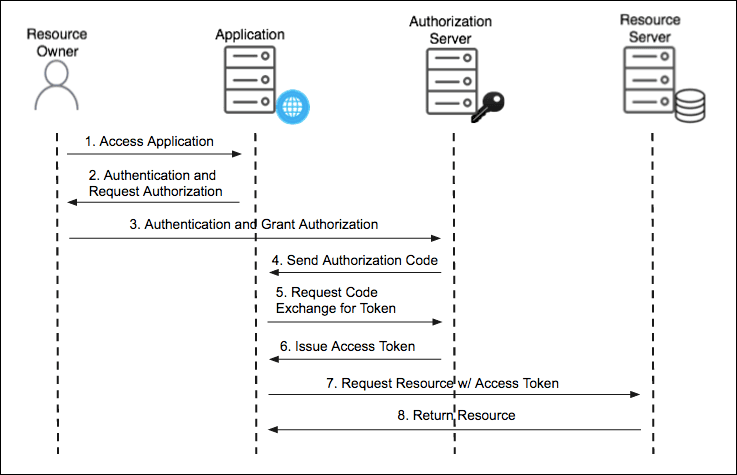
OAuth 2.0 is an open standard for access delegation, commonly used as a way for users to grant websites or applications limited access to their information without revealing their passwords. It acts as a bridge between the user and the application requesting access, facilitating trust-based communication. This mechanism allows third-party applications to obtain limited access to a web service, on behalf of the user.

Instead of asking for usernames and passwords—which increases the risk of data breaches—OAuth issues access tokens that authorize specific actions for a limited time. These tokens are issued by an authorization server and are passed to a resource server, where they validate the user’s identity and access rights.

A familiar example is "Login with Google" or "Sign in using Facebook." Behind this seemingly simple experience, OAuth is quietly coordinating the secure sharing of credentials, ensuring that the user’s private data remains protected and under their control.

* 1. **Key Roles in OAuth Flow**
* **Resource Owner**: The user who owns the data and grants access to a third-party application.
* **Client (Application)**: The application requesting access to the resource owner’s protected resources.
* **Authorization Server**: Validates the user’s identity and issues tokens.
* **Resource Server**: Hosts the protected resources and responds to requests authenticated with tokens
  1. **Working of OAuth**

One of the most widely adopted OAuth flows is the **Authorization Code Flow**, which is used by web and mobile apps where the client (application) interacts securely with an authorization server and resource server.



*Fig 1 : OAuth Authorization Code Flow*

The OAuth 2.0 Authorization Code Flow begins when the **Resource Owner**, typically the user, attempts to access a third-party **Client Application**. To gain access to protected resources, the client redirects the user to the **Authorization Server** for authentication and authorization. The user logs in and is prompted to grant permission to the application. Once the user approves, the Authorization Server responds by redirecting the user back to the client with an **Authorization Code** embedded in the URL.

The client then extracts this code and sends it in a secure server-to-server request to the Authorization Server, along with its own client credentials (like the client ID and secret). This is a crucial step, as it prevents exposing sensitive tokens to the user’s browser. Upon verifying the request, the Authorization Server issues an **Access Token**, and sometimes a **Refresh Token**, to the client.

With the Access Token, the client can now make authenticated requests to the **Resource Server**, including the token in the HTTP Authorization header. The Resource Server verifies the token's validity and, if valid, returns the requested user data or protected resource to the application. This entire process ensures that the client can access resources on behalf of the user without directly handling the user's credentials.

1. **Implementation of OAuth-Based Authentication System**

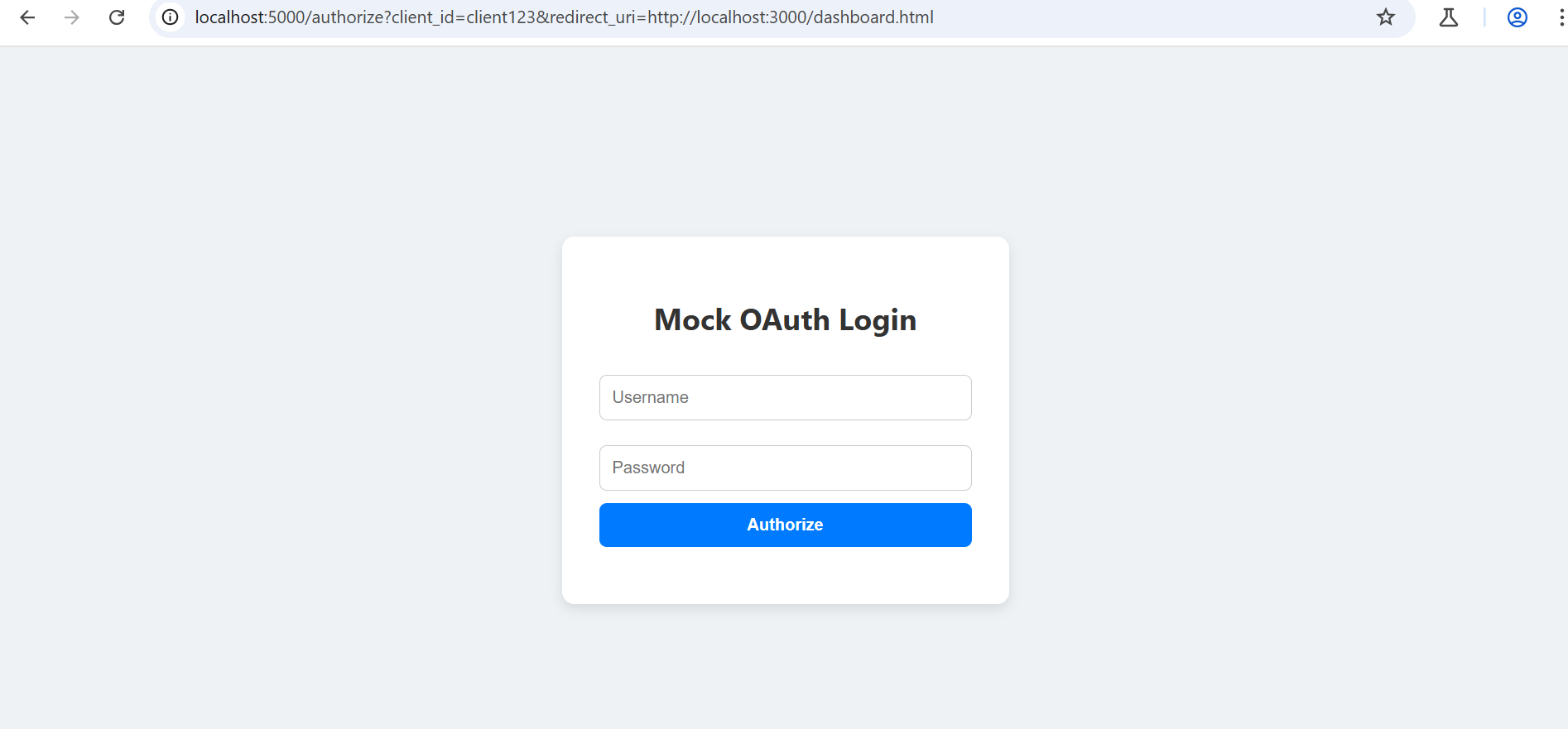
The OAuth 2.0 Authorization Code flow was implemented in a controlled environment as part of this project. The system consists of a simple web client (built with HTML, CSS, and JavaScript) and a backend authorization server (built using Node.js and Express). The OAuth flow was simulated to allow users to log into a student portal through a third-party authentication mechanism.

The following sections explain the implementation of both the client-side and server-side components of the OAuth system.

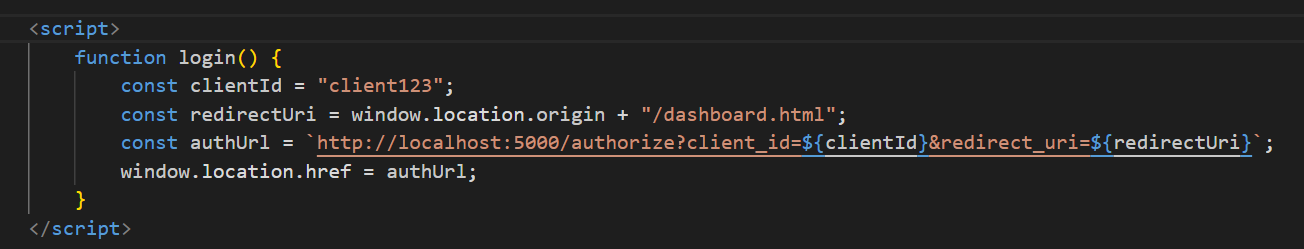
* 1. **Client-Side Implementation**

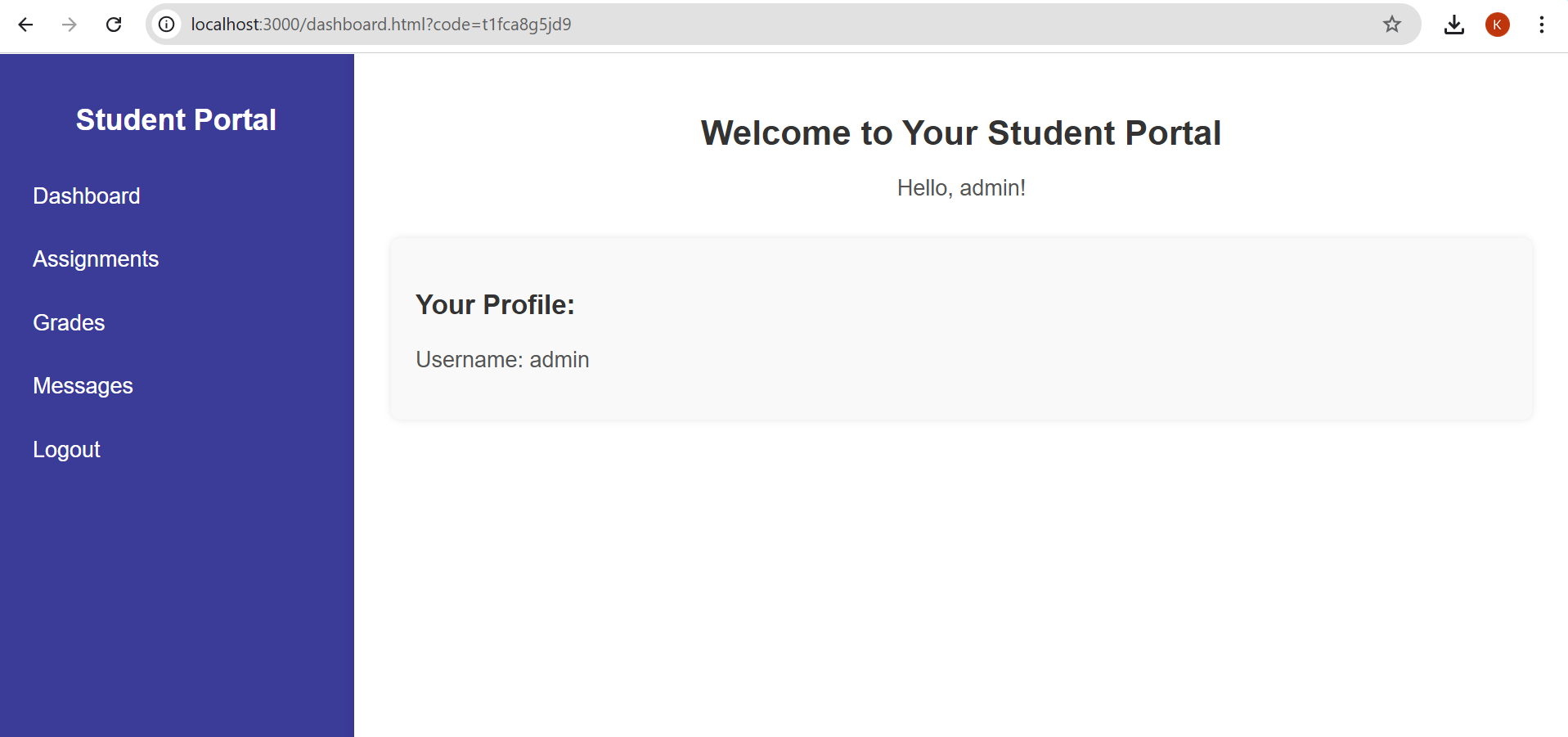
The client-side application simulates the student portal where users can log in using OAuth 2.0 authorization. The key components are:

**2.1.1. Index.html (Login Page)**  
 This is the initial landing page where users are prompted to log in via the OAuth mechanism. Upon clicking the "Login with OAuth" button, the user is redirected to the authorization server to grant access.



**Key Features**:

* + The login() function creates the authorization URL with the client\_id and redirect\_uri parameters.
  + The user is redirected to the authorization server (<http://localhost:5000/authorize>).

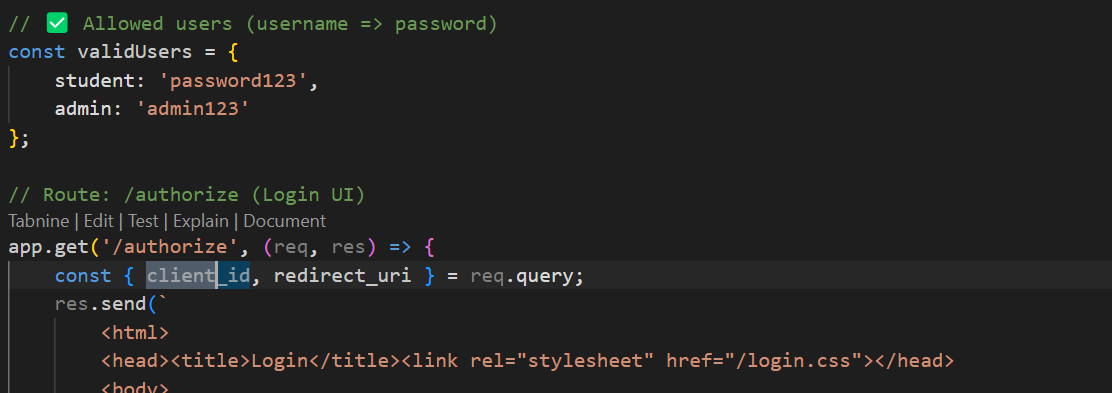
****

**2.1.2. Dashboard.html (User Dashboard)**  
 After the user successfully grants authorization, the authorization server redirects them to the dashboard.html page with an authorization code in the URL. The client then exchanges this code for an access token.

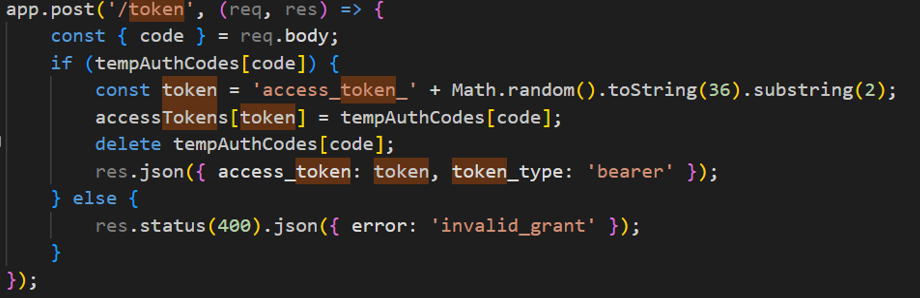
* **Key Features**:
  + The code parameter from the URL is extracted and used to request an access token from the /token endpoint.
  + The access token is then used to request user information from the /userinfo endpoint, which is displayed on the dashboard.
  1. **Server-Side Implementation (Backend - Express.js)**

The server-side application simulates an OAuth authorization server and resource server. The implementation uses Express.js to expose endpoints for authorization, token issuance, and user information retrieval.

**2.2.1. Authorization Endpoint (/authorize)**  
 This endpoint handles user login and approval. After the user grants permission, the server generates an authorization code and redirects the user back to the client with the code.



* **Key Features**:
  + The client\_id is validated, and if correct, an authorization code is generated and sent back to the client’s redirect\_uri.

**2.2.2. Token Endpoint (/token)**  
This endpoint accepts the authorization code and exchanges it for an access token. The server validates the code and issues a token.

* **Key Features**:
  + The authorization code received from the client is validated.
  + A mock access token is returned if the code is valid.

**2.2.3. User Info Endpoint (/userinfo)**  
This endpoint returns user information, but only if a valid access token is provided.



* **Key Features**:
  + The access token is checked against the expected value.
  + User information is returned only if the token is valid.

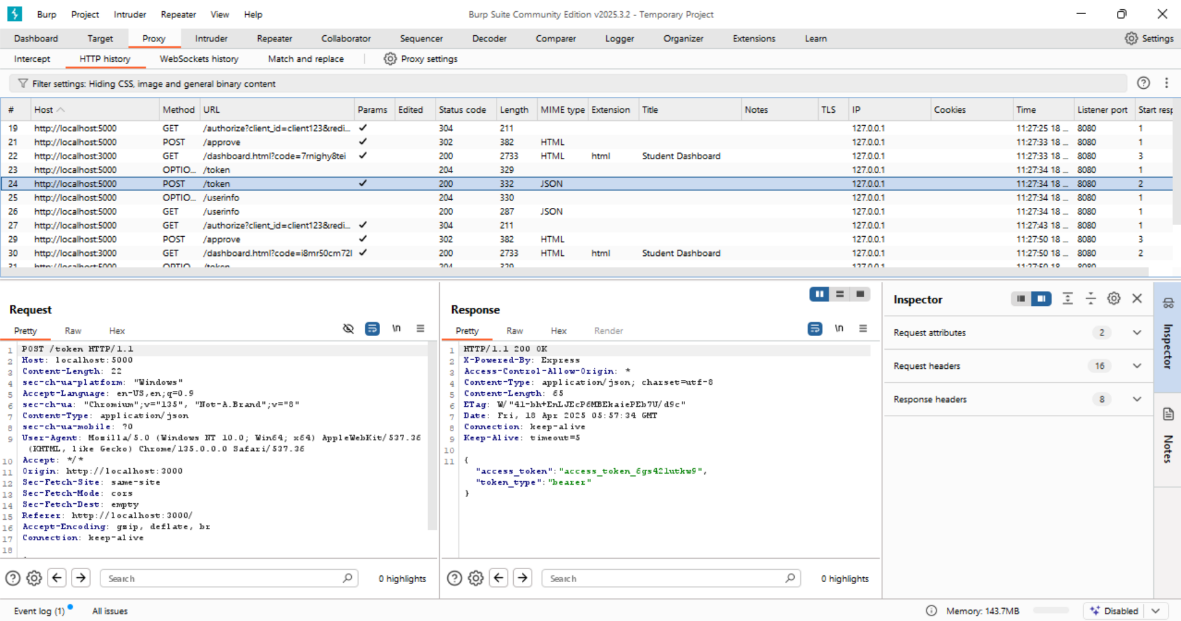
1. **Security Testing and Bypass Simulation Using Burp Suite**

### **Attack:** Token Reuse Attack in OAuth 2.0

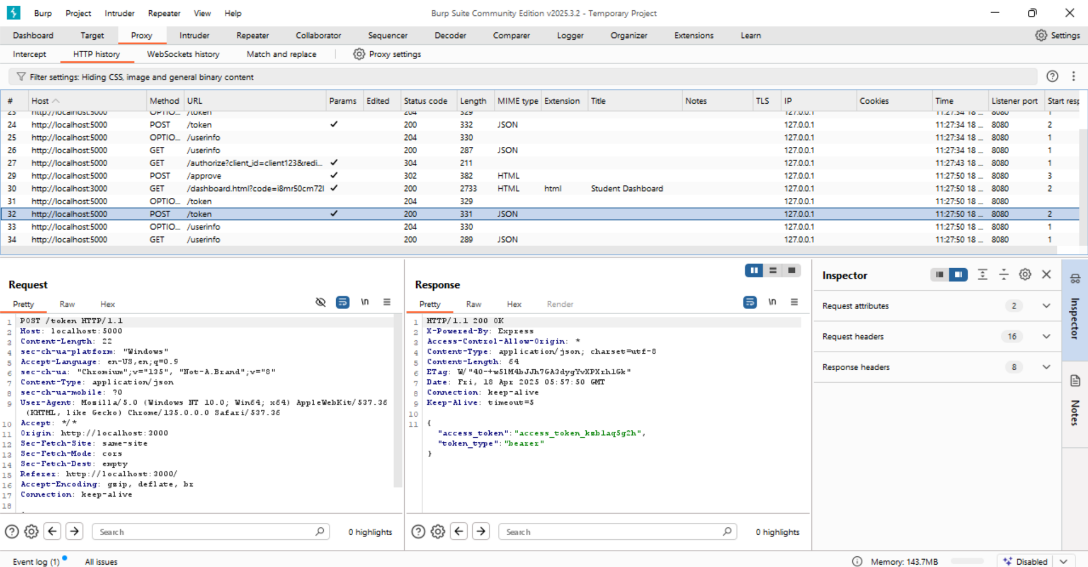
* 1. **Attack Overview**:

In this attack scenario, the access token obtained by logging in as an **Admin** user is **reused** to access resources intended for a **Student** user. The attack exploits the lack of proper token validation for user roles and demonstrates the risks associated with improperly managed OAuth tokens. By reusing the Admin's access token, an attacker can bypass authorization checks and gain unauthorized access to student-specific resources.

#### **Steps Involved in the Attack:**

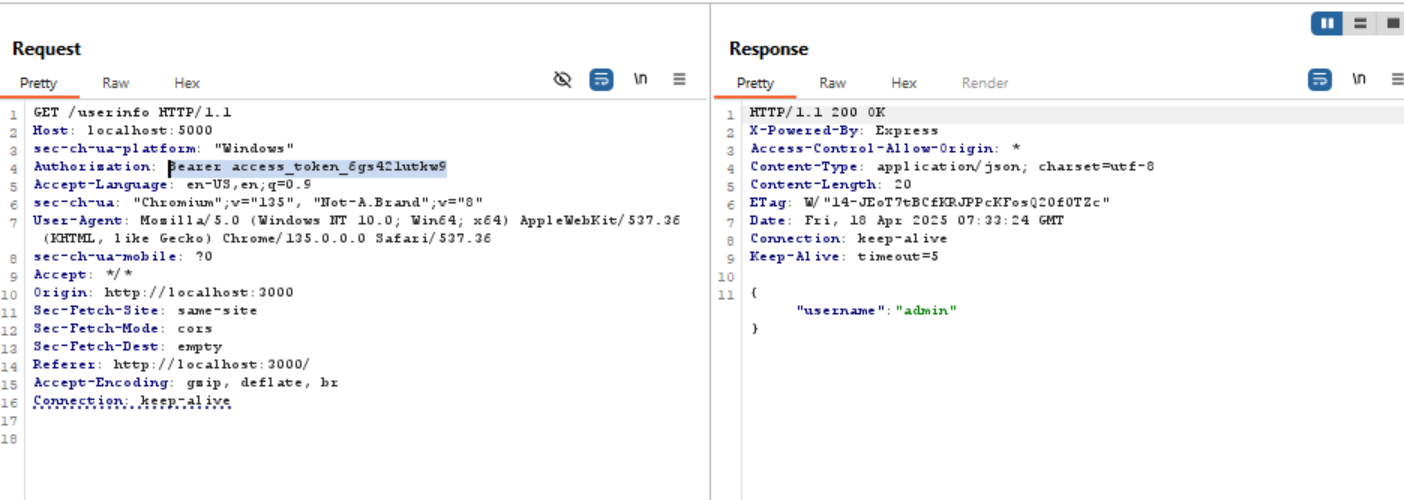
1. **Admin Login:**
   * First log in as an **Admin** user, completing the OAuth 2.0 flow to obtain an **Admin access token**.
   * The access token issued for the Admin user allows access to resources that require Admin privileges, such as sensitive system settings or administrative data.

*Fig: Access token of admin*

1. **Student Login:**
   * Then log in again, but this time as a **Student** user, obtaining the **Student’s access token**.
   * Normally, this token should only allow access to student-specific data, such as grades and personal information.

***Fig: Access token of student***

1. **Reuse:**
   * Rather than using the **Student's access token** to make requests for student data, the attacker **replaces the Student token with the Admin token**.
   * Using Burp Suite or manual manipulation of the HTTP request, the attacker crafts a request to access **Student** resources (such as user information) while using the **Admin access token** in the Authorization header.



*Fig: Reusing the tokens*

**Accessing Protected Resources:**

* + Since the Admin token has broader permissions, it can be used to access resources not intended for the Student user. The system fails to properly differentiate between the user roles when verifying the token, allowing the attacker to access student data or perform unauthorized actions.
  1. **Vulnerabilities Exploited**:

1. **Lack of Role-Based Validation:**
   * The system does not enforce role-based token validation. Although the access token is intended for a specific user role (Admin or Student), the system only checks if the token is valid and does not check whether it is used for the correct role.
2. **Lack of Role-Based Validation:**
   * The system does not enforce role-based token validation. Although the access token is intended for a specific user role (Admin or Student), the system only checks if the token is valid and does not check whether it is used for the correct role.
3. **Token Reuse Across Different Roles:**
   * The access token issued for an Admin user is not restricted to Admin-level resources. It can be reused by a Student user, allowing them to bypass security controls and access sensitive data.
4. **No Token Expiration or Revocation Mechanism:**
   * The system does not invalidate tokens after use or provide a mechanism to revoke tokens. Once an attacker obtains the Admin token, they can reuse it indefinitely, further extending the window of opportunity for exploitation.
   1. **Impact of Attack**
5. **Unauthorized Access:**
   * By reusing the Admin token, the attacker gains access to **Student-specific resources** such as grades, profiles, and other protected data that should not be accessible to them.
6. **Privilege Escalation:**
   * The attacker could potentially escalate their privileges by using the Admin token to access administrative functions, modify data, or manipulate the system in ways that could harm the application's integrity or confidentiality.
7. **Compromise of Sensitive Data:**
   * The attacker’s unauthorized access to student data could result in data exposure, leading to privacy violations or even the theft of sensitive information.
8. **Loss of Trust:**
   * Such attacks undermine trust in the system’s security, especially if the attacker manages to manipulate data or gain access to highly sensitive resources like admin settings or personal student information
   1. **Mitigation Strategies**
9. **Role-Based Token Validation:**
   * Implement role-based validation for tokens. Each token should be validated not only for its authenticity but also for the specific user role or scope it is intended for. The system should check that the user associated with the token is authorized to access the requested resources.
10. **Scope-Based Access Control:**
    * Use OAuth scopes to limit the actions that can be performed by the token. For example, the Admin token should have an admin:read scope, while the Student token should have a student:read scope. The system should verify that the token’s scope matches the requested resource type.
11. **Token Expiry and Revocation:**
    * Ensure that access tokens are **short-lived** and **expire** after a predefined period, forcing users to re-authenticate regularly. Implement a token revocation mechanism to invalidate tokens when necessary, especially if they are suspected of being compromised.
12. **Session Management:**
    * Bind tokens to specific user sessions and enforce secure session management practices. This will prevent tokens from being reused across different user roles or sessions.
13. **Implement Anti-Replay Measures:**
    * Add mechanisms such as **nonce values** or **timestamp validation** to prevent replay attacks. This ensures that tokens are not reusable or re-accepted by the system after they have been used.

**Conclusion**

The **Token Reuse Attack** demonstrated the dangers of improper OAuth token management, particularly in the context of user role validation. By reusing an **Admin access token** for accessing **Student data**, the attacker bypassed the intended access controls, leading to unauthorized access and potential data compromise. To prevent such attacks, OAuth implementations must rigorously enforce role-based token validation, use token scopes effectively, and implement robust expiration and revocation mechanisms. This will ensure that access tokens cannot be misused across different roles, thus safeguarding user data and system integrity.

**References:**

<https://portswigger.net/web-security/oauth/lab-oauth-authentication-bypass-via-oauth-implicit-flow>

<https://anmolsinghthakur.medium.com/oauth-2-0-authentication-vulnerabilities-9e1bc273852a>

https://portswigger.net/web-security/oauth