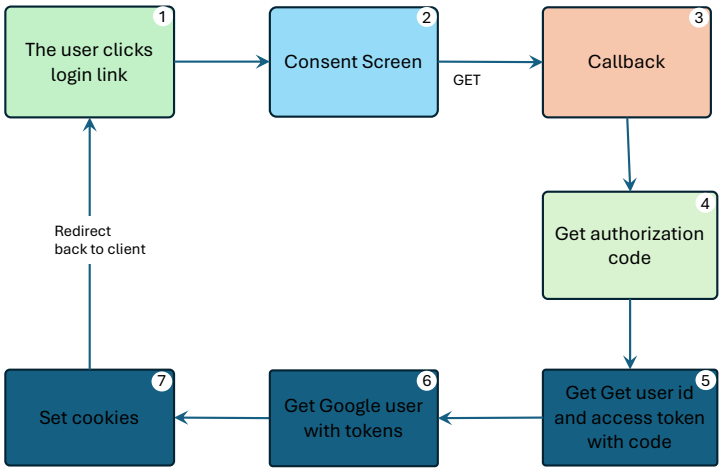
Secured Software Engineering
(COMSM0164)
Lab Exercise: Securing Real-Time APIs
(Part 2 – OAuth)

Prerequisite:

(1) Visual Studio Code (VS Code)
(2) A GitHub account
(3) Some knowledge of JavaScript and Node.js for API functional implementation.
(4) The client-side in this exercise also uses JavaScript, HTML and CSS. The latter two are optional to enable a user-friendly and interactive web interface.
(5) Google account for access to third-party Google Identity Provider. You may want to create a Google account primarily for this exercise.
(6) OpenSSL

Overview

In this exercise, you will explore the implementation of federated API authentication based on OAuth - a standard protocol that allows secure access to resource endpoints on behalf of a user without requiring the user to share their passwords. The applied use case is based on a previous lab exercise where we leveraged WebSockets to implement a real-time discussion forum called **SSE-Forum.** The security requirement being implemented is to ensure that every member of the forum is appropriately authenticated before joining the forum. To ensure a clear separation of concerns, we implement Authorisation Code Grant Type and leverage Google's OAuth third-party authentication service. The diagram below demonstrates the authorisation code process.



OAuth Authorisation Code Process using Google Identity Provider

The process starts with a user *clicking a login link*, who would, as a consequence, be presented with a *consent screen* hosted by Google. Once the user has given consent for SSE-Forum to access their account, a *callback* will be routed through the user's browser to a predefined callback endpoint. This endpoint was specified at the point SSE-Forum registered with Google identity provider (IdP). The registration serves a dual purpose. First, Google becomes the trusted third party that is responsible for authenticating SSE-Forum, and second, the subsequent provision of authentication and authorisation services for members of SSE-Forum. The payload associated with the callback request includes an authentication code that SSE-Forum will use to

retrieve the user's ID and access token. The last two steps in the process are to use the access token to retrieve the user's details and set cookies for user session management. This authentication process is known to be vulnerable to authorisation code interception attacks. We will implement a Proof Key for Code Exchange (PKCE) to mitigate this known vulnerability.
Use Google OAuth 2.0 playground -https://developers.google.com/oauthplayground/ to further explore the Authorisation Code Grant type. In addition, https://jwt.io is a link for decoding base 64 tokens.
What you will learn in this exercise:

- How to get Google OAuth API keys
- How OAuth works
- How to build an OAuth implementation with NodeJS from first principles, giving you a view of every single request and a better understanding of how OAuth works.
- How to implement PKCE to mitigate authorisation code interception attacks.

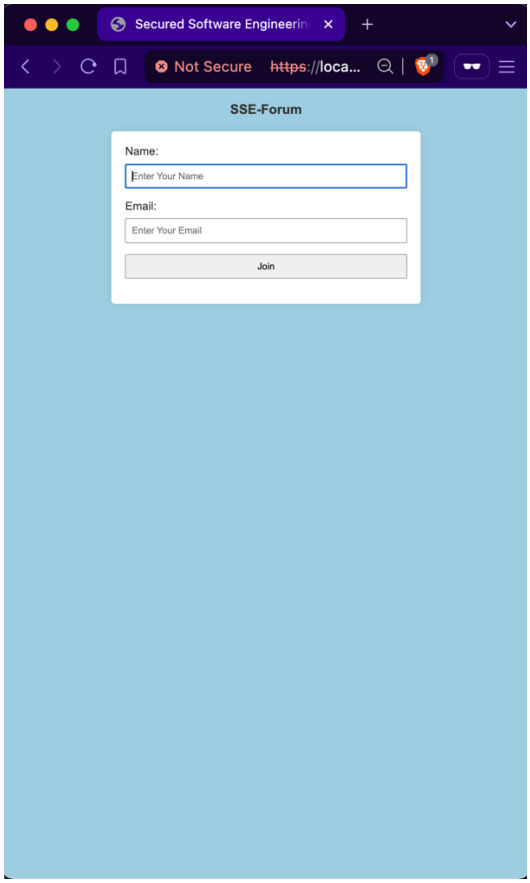
To achieve this learning outcome by following steps 1-4. It is assumed that you have completed two previous exercises: (1) Implementing HTTPS APIs Using SSL Certificates and (2) Securing Real-Time APIs (Part 1 – WebSockets).

Step 1: Project Setup

As one of the challenge tasks in the Real-Time APIs based on WebSockets lab exercise, you were required to refactor SSE-Forum to use WebSocketsSecure (wss) protocol instead of WebSockets. This exercise will wrap OAuth around the wss version of SSL-Forum implementation.

- Clone your implementation of wss challenge task from GitHub into another repository folder named `ws_oauth`.
- Ensure you've added your `.env` file to your directory and express server and web sockets are listening to the appropriate port on `server.js`.
- Check that SSE-Forum is configured to run over https. This includes implementing the steps requiring the use of OpenSSL to generate a self-signed certificate for authenticating the API server. Ensure to name your private key, CST and TSL/SSL certificate `private.key`, `custom.cst` and `server.crt` respectively.
- Install all the necessary dependencies. You can achieve this by executing the command `npm install` on your terminal with `ws_oath` as the root directory.

On completing this step, unauthenticated users can join SSE-Forum to participate in forum discussions using a browser client. Run `npm start` on your terminal for SSE-Forum to begin listening to clients' requests. To make a client request, input [server_ip_address]:3000 on your web browser. An illustration of a successful browser request to the root endpoint is shown in the figure below. Note the warning generated by the browser as a result of using a self-signed certificate.



Step 2: Configure OAuth on Google

You will use Google Cloud Console (https://console.cloud.google.com) to authenticate SSE-Forum and generate the credentials necessary to enable Google IdP to authenticate SSE-Forum API requests from users. You will also obtain OAuth 2.0 credentials, such as a client ID and client secret, that are known to both Google and SSE-Forum. You will need to log in with a Google account (you may want to create a new account for this exercise). If logging in for the first time, you will also need to agree to the terms of use before proceeding. The following configuration steps will guide you in setting up OAuth on Google for SSE-Forum:

Create a project

1. Log in to Google Cloud console - https://console.cloud.google.com with a Google account
2. Click select a project link. This will open a new page with the list of existing projects.
3. Click the new project link at the top right corner.
4. Use SSE-Forum as the project name and press the create button. You do not need to select an organisation (for a public-facing application).

Configure OAuth consent screen

5. Select the created project.
6. From API and services menu, select OAuth consent screen. This will be the screen that user will be shown before their information is disclosed to SSE-Forum.

7. Click *Get Started with Google auth platform not configured yet* button and add the following details:
    - App name - SSE-Forum
    - User support email – [your registered Google email or any other email]
    - Click next and select external audience (you will have to specify the list of users for testing mode)
    - Add contact info and press the finish /create button.

8. Select the *Branding* tab and add the following properties:
    - Application home page – https://localhost:3000
    - Privacy policy link – https://localhost:3000/privacy
    - Application terms of service link – https://localhost:3000/tos
    - You do not need to add an authorised domain
    - leave the same email and developer contact information
    - Click save

9. Select the *Audience* tab
    - Add test users. These are the users who will be able to test your app without bringing it out of sandbox mode.

10. Select *Data access* tab
    - Click Add/remove scopes button.
    - There are multiple scopes to select. In this exercise we are only interested in .../auth/userinfo.email and .../auth/userinfo.profile. Select those first 2 and press the update button.
    - Click save

11. Select *Verification Centre* tab
    - Nothing to do in this tab

12. Select *Clients* tab

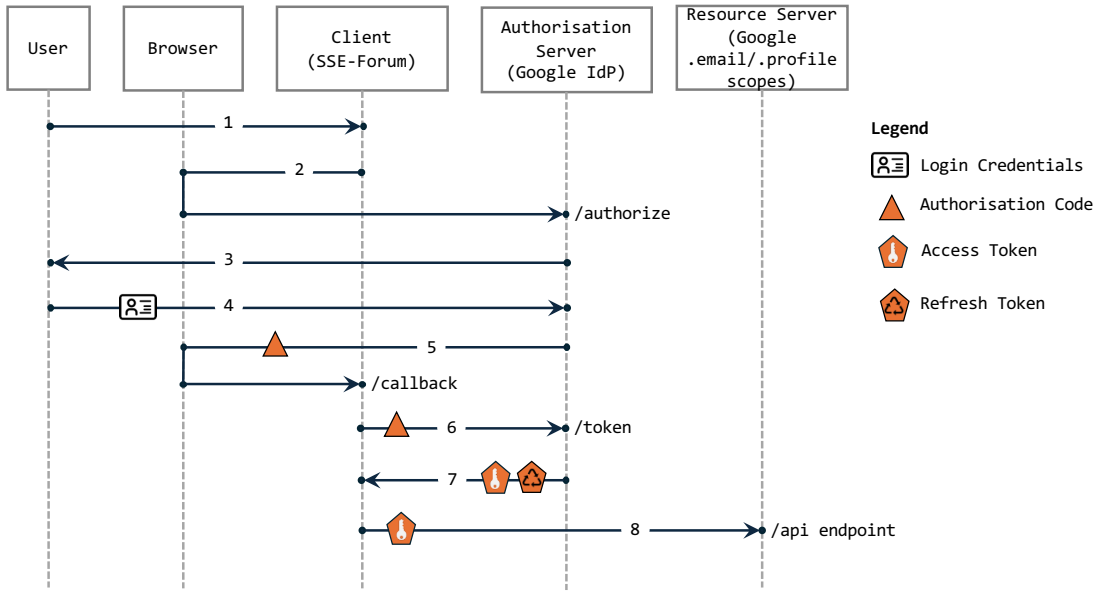
    A client ID is used to identify a single app to Google's OAuth servers.
    - Click *Create client* link
    - Select web application as the application type
    - Set the name to SSE-Forum
    - Add https://localhost:3000/auth/google/callback as authorised redirect URI
    - Press the Create button
    - Take note of the client ID and client secret values.

At the end of this step, you will have authenticated SSE-Forum with Google IdP. You have also configured OAuth on Google to authenticate users on SSE-Forum's behalf. SSE-Forum is also able to directly call the user's profile resource endpoint once a user grants consent.

Step 3: Setting up OAuth over HTTPS Server

The OAuth authorisation code flow is demonstrated in the figure below. The HTTP Server sits on the application (client). It redirects unauthenticated users' requests to an authorised endpoint hosted by the Authorisation server to be authenticated and to grant consent to SSE-Forum to access their personal information contained on the resource server API endpoint. This is demonstrated by flows 1-4 in the diagram. Once

the user is successfully authenticated, a HTTP post request is made by the IdP to the callback url endpoint set while configuring OAuth on Google in step 2 above. The callback contains an authorisation code that is subsequently used by the application to obtain an access token from the authorisation server.



To achieve this step, add the following attributes to your `.env` file

```
PORT_WS=8888
PORT_OAUTH=3000
GOOGLE_CLIENT_ID=[your_google_client_id]
GOOGLE_CLIENT_SECRET=[your_google_client_secret]
GOOGLE_REDIRECT_URI=https://localhost:3000/auth/google/callback
```

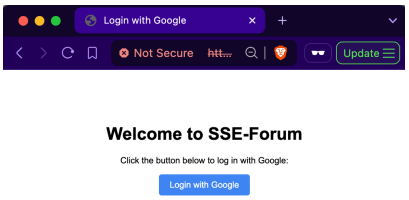
Also, install the necessary dependencies with the resulting package.json file containing the following dependencies:

```
{
      "scripts": {
          "start": "node oserver-ws.js"
        },

      "dependencies": {
          "axios": "^1.8.4",
          "cookie-parser": "^1.4.7",
          "cors": "^2.8.5",
          "crypto": "^1.0.1",
          "dotenv": "^16.4.7",
          "express": "^4.21.2",
          "express-session": "^1.18.1",
          "googleapis": "^146.0.0",
          "ip": "^2.0.1",
          "jsonwebtoken": "^9.0.2",
          "ws": "^8.18.1"
      }
}
```

Next copy `oserver-ws.js`, `helper-oauth-handler.js` `helper-oauth-url.js`, `eindex-oauth.html`, and `login.html` files from the exercise folder into the `ws_oauth` repository folder. Also update the `package.json` scripts start property value to point to `oserver-ws.js`. These files contain the necessary refactoring to implement OAuth authorisation code flow in SSE-Forum. **Take some time to study and understand the additional functionality implemented in this component.** The key points are as follows:

1. Starting with `oserver-ws.js`, observe the different resource endpoint implementations. The authentication process commences when the user makes a request to the root resource endpoint on the client, with a response that sends a login page (`login.html`) to the user's browser.

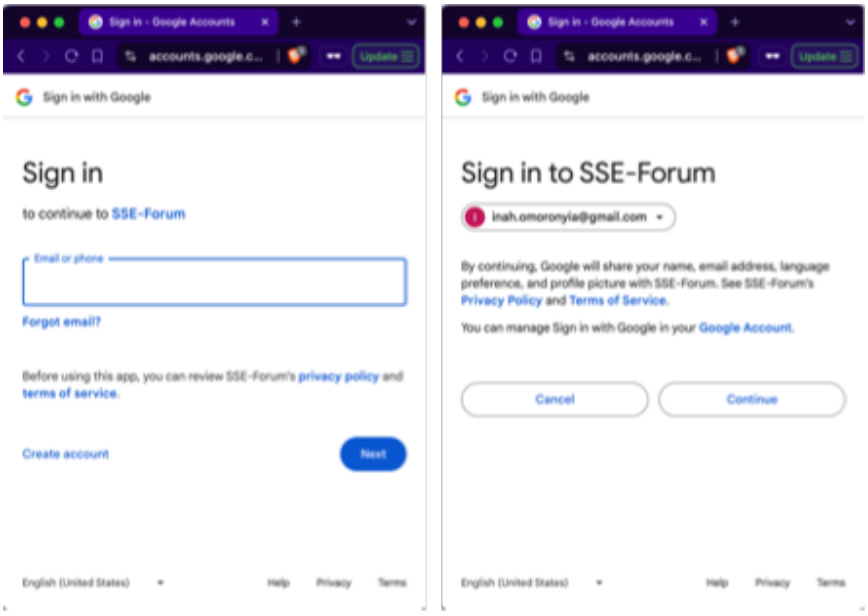


2. The login page then makes a request to the `/session` resource endpoint. The response contains a session identifier (`msession-id`), which is used to manage the OAuth state between the user browser and the SSE-Forum client.

3. A successful session initialisation results in a further request to the `/auth` resource endpoint on the client. This triggers the OAuth authentication process with a call to `authenticateToken` function on `oserver-ws.js`.

4. The `authenticateToken` function calls the `getGoogleOAuthURL` function on helper-oauth-url.js to generate a URL that redirects the user's browser to the authorisation endpoint hosted by the authorisation server. Observe how `authenticateToken` adds a state query parameter to the redirect URL and sets its value to `msession-id`.
   The state parameter is a standard part of the OAuth 2.0 specification. It is used to maintain state between the request and callback. By including the session identifier in the state parameter, we've securely associated the OAuth flow with the user's session. We can now prevent Cross-Site Request Forgery (CSRF) attacks by validating the state parameter. Observe how the `/auth/google/callback` endpoint validates a session's state parameter.

5. On receipt of a response, the login page redirects the user's browser to Google's OAuth authorisation endpoint, where the user is presented with an authentication screen. When the user successfully authenticates, they are then presented with a consent screen as shown in the diagram below.



Observe the notice of personal information that Google will disclose to SSE-Forum when consent is provided. This notice will differ depending on the scopes selected during OAuth configuration in Step 2 above. The user can select a subset of personal data to grant access to if multiple API scopes are selected during configuration.

6. Once consent is granted by the user, Google OAuth redirects the browser to the client using the authorised redirect URI endpoint (`/auth/google/callback`) provided during configuration in Step 2 above. Google OAuth generates and associates the authorisation code as a query string to the URI.

7. The `/auth/google/callback` endpoint uses the `googleOauthHandler` function defined in `helper-oauth-handler.js` to retrieve the user's ID and access tokens from scoped Google's resource endpoints. Observe how `googleOauthHandler` retrieves the authentication code from the request query string and uses `getGoogleAuthTokens` function to get tokens data. Also note how `getGoogleAuthTokens` uses the authorisation code, in addition to other properties to make a direct API request to Google's authorisation servers. These properties are defined in an `options` object and associated as a query string to Google OAuth's token endpoint (https://oauth2.googleapis.com/token). An options object definition with corresponding returned tokens data is illustrated below.

```
options = {
    code: authorization_code,
    client_id: process.env.GOOGLE_CLIENT_ID,
    client_secret: process.env.GOOGLE_CLIENT_SECRET,
    redirect_uri: process.env.GOOGLE_REDIRECT_URI,
    grant_type: 'authorization_code'
}

tokens = {
   access_token: xxxxx,
   expires_in: 3599,
```

```
    id_token: xxxxx,
    refresh_token: xxxxxx,
    scope: 'https://www.googleapis.com/auth/userinfo.email
    https://www.googleapis.com/auth/userinfo.profile'
    scope: 'Bearer'
}
```

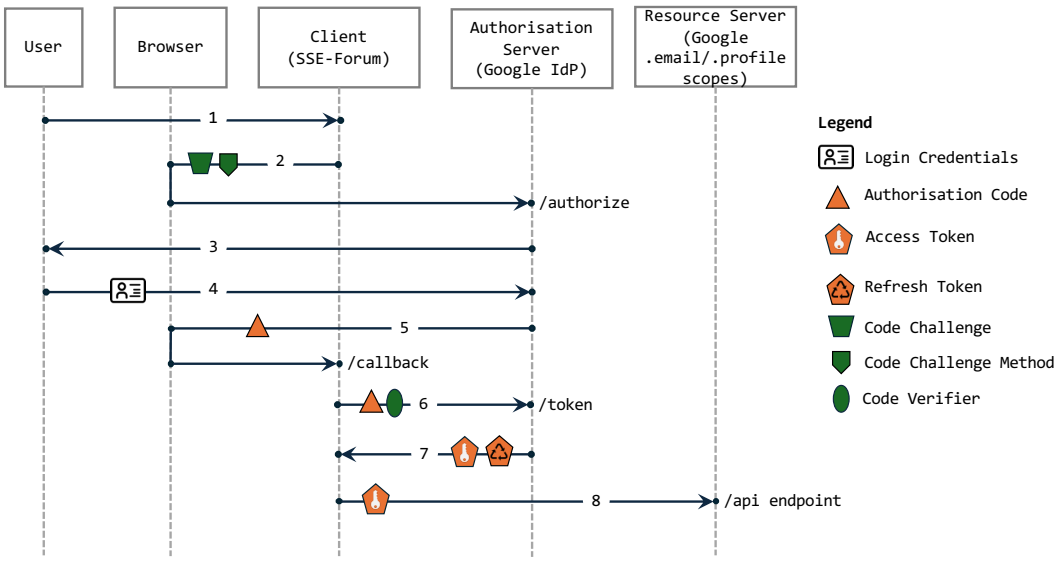
There are one of two possible ways to retrieve a Google user's profile. The first is to decode the base 64 encoding of id_token. This is achieved in `googleOauthHandler` by calling `jwt.decode(tokens.id_token).` You can also view the content of the encoding by entering the value in https://jwt.io . The second approach is to make a direct API call to Google's userinfo resource endpoint. This request will have an authorization header set to Bearer with the value set as tokens.access_token. See the function `getGoogleUserInfo` in `helper-oauth-handler.js` for more information.

8. Finally, a successful execution of `googleOauthHandler` returns the Google user's profile and tokens data objects. The property values are extracted by `/auth/google/callback` endpoint and set as cookies on the response object. Subsequently, access is granted to the user to join the forum with default user and email set to values from set cookies. This is shown in the diagram below.

Step 4: Using PKCE over OAuth Authorisation Code flow

Proof Key for Code Exchange (PKCE) is an extension for OAuth Authorisation Code grant type to protect against authorisation code injection and stealing authorisation codes. It entails the generation of a code verifier, a code challenge and a decision on

a challenge method. A variant of the authorisation code flow is demonstrated using the diagram below.



To achieve this step, copy `oserver-pkce-ws.js`, `helper-oauth-pkce-handler.js` `helper-oauth-pkce-url.js`, and `helper-pkce.js` files from the exercise folder into the `ws_oauth` repository folder. Also, update the `package.json` scripts start property value to point to `oserver-pkce-ws.js`. These files contain the refactoring of default implementations to wrap PKCE around OAuth authorisation code. **Again, take some time to study and understand the additional functionality implemented in these components.** The key points are as follows:

(1) The `helper-pkce.js` file contains functions for generating code verifier, code challenge and code challenge methods.

(2) The `authenticateTokenPkce` function defined in `oserver-pkce-ws.js` is a refactoring of `authenticateToken`. It uses `getGoogleOAuthURLpkce` function defined in `helper-oauth-pkce-url.js` to generate a redirect URL that also embeds the code challenge and code challenge method into the query string.

(3) Similarly, the `helper-oauth-pkce-handler.js` contains `googleOauthHandler` function used by `/auth/google/callback` endpoint. This function calls `getGoogleAuthTokens`, which adds a code verifier to the options object to retrieve a tokens data object from the authorisation server. The refactored options object is illustrated below:

```
options = {
    code: authorization_code,
    client_id: process.env.GOOGLE_CLIENT_ID,
    client_secret: process.env.GOOGLE_CLIENT_SECRET,
    redirect_uri: process.env.GOOGLE_REDIRECT_URI,
    grant_type: 'authorization_code'
    code_verifier: code_verifier
}
```

Overall, the observed behaviour from a user perspective remains the same as in Step 3.

Challenge Task 1:

OAuth 2 is now essential for modern API security, relying on a range of key components and orchestration patterns to establish secure authentication pathways. Specifically, reflect on the OAuth authorisation code grant type implemented in this exercise, its flow and implementation process.

(1) A complex software ecosystem is often an orchestration of seemingly autonomous services, systems and dependencies. High observability, autonomy and adaptability are some of the inherent properties expected of this modern software ecosystem. Whereas achieving security will often mean the delegation of the security function. In this exercise, security is delegated to a third-party identity provider. Do you think that these inherent properties are important in the engineering of modern software? If so, is there a balanced trade-off between these properties and achieving security? Discuss your answer in no more than 300 words.

(2) Discuss in no more than 300 words why proper token management, secure storage, and flow selection are important in the design of an OAuth authentication system.

(3) With the current implementation of SSE-Forum, when a token expires, the user will be required to log in again. Discuss the advantages and disadvantages of this design approach. Implement an extension of SSE-Forum that uses a refresh token to prevent the expiration of a session.

(4) Google OAuth 2.0 is one of many federated identity providers. Most social network systems offer identity services. Redesign and implement SSE-Forum using GitHub as an identity provider. Documentation can be found using the link: https://docs.github.com/en/apps/oauth-apps.

(5) In this workshop, we've considered how to build an OAuth implementation from the ground up without leveraging any authentication middleware such as passport.js (https://www.passportjs.org). Redesign and implement SSE-Forum Passport strategy for Google OAuth 2.0. Documentation can be found using the link: https://www.passportjs.org/packages/passport-google-oauth2/

Deliverables

Submission should be made electronically as a PDF containing all deliverables via Blackboard for Exercise X. The PDF should also contain a maximum of 250 words of reflection on the exercise to help keep a record of your learning. Be sure to include a link to your GitHub repository for challenge tasks 3-5.

IMPORTANT NOTES:

- Check that you've invited the course lecturer to your GitHub repository and that an acceptance of the invitation was received and acknowledged.

- Also, ensure that one of the TAs or the course lecturer has reviewed your solutions before the end of the lab session.