Bluff Game

Course: **COMP.SEC.300 Secure Programming**

Tampere University

Authors:

Samith Binda Pantho [ID: 152702670]

Md Sakib Hasan [ID: 153061145]

Saurav Paul [ID: 153060573]

Date: May 2025

Table of Contents

Introduction

The **Bluff Game** is a strategic multiplayer game built around deception and psychological insight. Designed for two players competing against each other, the goal is to be the first to score the maximum points. Players take turns rolling a dice and verbally declaring the number they rolled—ranging from one to six. However, players are free to lie about the actual result. The opponent must then decide whether to call **"Bluff!"** or **"Believe"** the declared number.

If the opponent correctly identifies a bluff, they earn a point; otherwise, the player who rolled the dice gets the point. These dynamic tests each player's ability to bluff convincingly and to detect deception in others, making the gameplay engaging, tense, and often humorous.

Objectives

- **Ensure Secure User Authentication and Authorization**: Implement robust user registration, login, and session handling using ASP.NET Core to prevent unauthorized access and account misuse.
- **Protect Game State and Communication**: Use FastAPI with WebSockets securely to ensure that real-time gameplay data (e.g., dice rolls, decisions) is transmitted over encrypted channels and cannot be intercepted or tampered with.
- **Prevent Cheating and Game Logic Manipulation**: Secure the dice-rolling logic on the server side, ensuring that users cannot manipulate dice outcomes or falsify game states through client-side tampering.
- **Input Validation and Sanitization**: Validate all client inputs (e.g., username, game moves, bluff/believe choices) to prevent injection attacks, malformed data, or exploits.
- **Implement Role-Based Access Control (RBAC)**: Restrict access to APIs (e.g., match creation, match result submission) based on user roles to ensure only authorized actions can be performed.
- **Secure WebSocket Connections**: Authenticate and validate users before enrolling them into the lobby WebSocket connection, ensuring only authorized players can access gameplay sessions and reducing the risk of impersonation or unauthorized participation.
- **Data Privacy and Integrity**: Ensure all personal data, game history, and scores are securely stored and retrieved, using encryption.
- **Use Secure Tokens (e.g., JWTs)**: Manage session state securely with JSON Web Tokens (JWTs), add encryption to important data, like userid, sessionid.
- **Error Log and Monitoring**: Implement secure error logging and real-time monitoring to detect unusual behaviour, unauthorized access attempts, or application errors—while ensuring that logs do not expose sensitive user data or internal application logic.
- **Promote Secure Coding Practices Across the Stack**: Encourage best practices in both Unity (C#) and backend (FastAPI and ASP.NET Core) development and dependency management to avoid introducing vulnerabilities.

- **Prevent Replay Attacks**: Ensure that game actions (e.g., dice rolls or move declarations) are uniquely identified and prevent attackers from resending intercepted data packets.
- **Secure API Rate Limiting**: Apply rate limiting and throttling on game and authentication APIs to mitigate abuse, brute-force attempts, or denial-of-service (DoS) attacks.
- **Encrypt Sensitive Data in Transit and at Rest**: Use strong encryption for any sensitive player data, such as user id, password, or tokens, both in databases and during network transmission.
- **Implement Secure Logging Practices**: Log user activity and errors without exposing sensitive information such as passwords, tokens, or personally identifiable information (PII).
- **Mitigate Cross-Site Scripting (XSS) and ensure secure token-based communication:** Sanitize and validate all inputs in the Unity frontend to prevent XSS and use secure Bearer token authentication over HTTPS for all API and WebSocket communication to avoid CSRF risks.
- **Apply Dependency Security and Patch Management**: Regularly update and audit third-party libraries used in Unity, FastAPI, and ASP.NET Core to avoid known vulnerabilities.
- **Use Secure Random Number Generation**: Ensure the dice roll mechanism uses cryptographically secure random number generators to avoid predictable results or manipulation.
- **Design for Secure Matchmaking and Room Isolation**: Isolate game rooms to prevent users from joining or interacting with sessions they're not authorized to access.
- **Ensure Secure Error Handling**: Avoid exposing internal stack traces or error messages to clients. Instead, provide generic error responses while logging detailed ones securely on the server.

Working environment

1. Development Tools

- **IDE/Editor:**
  - **Unity**: Used for developing the game's UI and gameplay logic. Unity will handle rendering, animations, and real-time game updates.
  - **Rider**: Used for scripting for game development part.
  - **Pycharm**: For working with FastAPI (Python)
  - **Visual Studio 2019:** For working at authentication server in ASP.NET Core (C#).
  - **Postman**: For testing API endpoints during backend development.

- **Version Control:**
  - **Git**: Used for source code management. All code is versioned and managed through Git, with commits pushed to the central repository.
  - **GitHub**: Used for hosting the repository, managing issues, and continuous integration (CI).

2. Programming Languages

- **Unity Game Development:**
    - **C#**: Used for scripting within Unity. Handles game logic, player interactions, and the interface.
- **Backend Development:**
    - **FastAPI (Python)**: For creating the real-time WebSocket server with a Rest Api and managing game state and interactions.
    - **ASP.NET Core (C#)**: For user authentication, authorization, and any backend API interactions (e.g., login, registration, and match management like match request, match accept or declined and match result update).
- **Frontend Development (Unity UI):**
    - **C#**: Unity's main scripting language used to create interactive elements and logic in the game UI.
- **Database:**
    - **MySQL**: For storing user data, game session history, and leaderboards.

3. Game Engine and Frameworks

- **Unity**: Used as the game engine for developing the UI, animations, and gameplay mechanics. Unity's rich ecosystem provides tools for physics, animations, and player input.
- **FastAPI**: Handles the backend logic, real-time communication via WebSockets for gameplay updates, and API endpoints for game-related interactions (such as managing users turns, storing scores, next moves, decide on winners etc.).
- **ASP.NET Core**: Handles authentication, authorization, and interaction with the FastAPI backend.
- **WebSockets**: Facilitates real-time communication with the Unity frontend.

4. Authentication and Security

- **JWT**: Used for secure authentication of users. JWT (JSON Web Tokens) are passed as Bearer tokens in the Authorization header for secure API communication

5. Networking and Communication

- **WebSocket Protocol**: Used for establishing a real-time, full-duplex communication channel between the Unity frontend and the FastAPI backend. This allows players to interact with the game instantly (e.g., bluffing, calling, dice rolls).
- **RESTful APIs**: For game management actions (user registration, login, etc.), a set of secure RESTful APIs is created using ASP.NET Core.

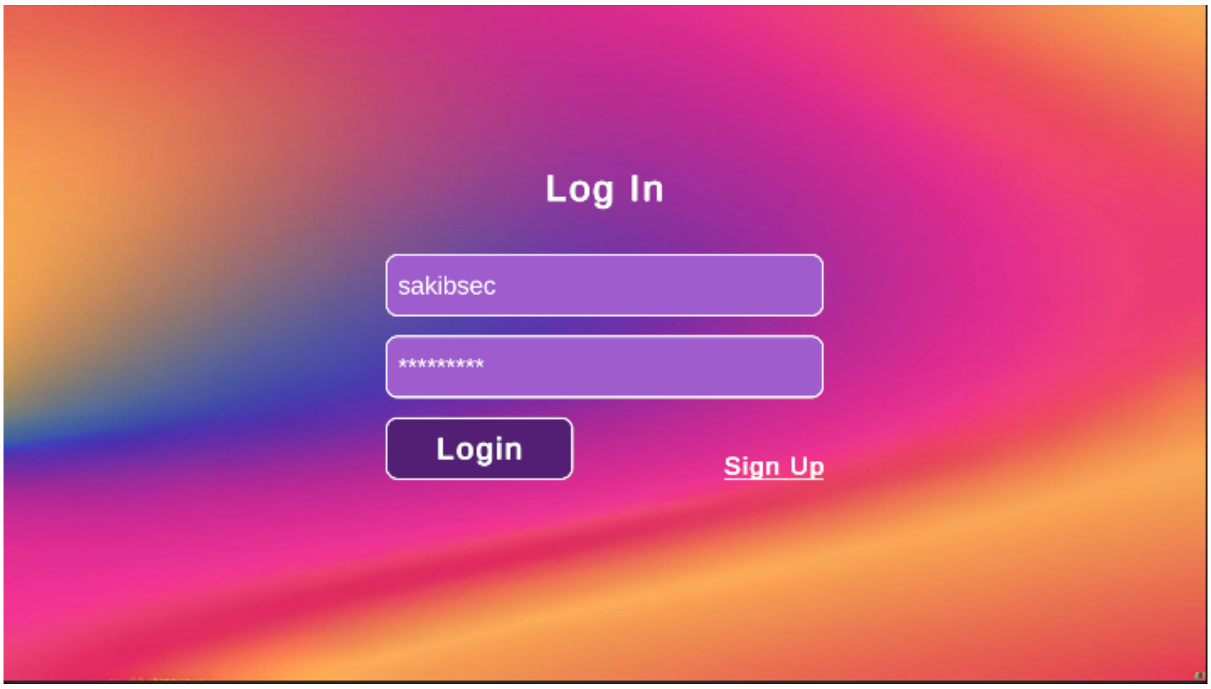
6. Deployment Environment

- **Docker**: Used for containerizing the FastAPI, ASP.NET Core, and game server. This ensures that the environment remains consistent across different machines.
- **Docker Compose**: Used for defining and running multi-container Docker applications (for example, having FastAPI, ASP.NET Core, and the database in separate containers).

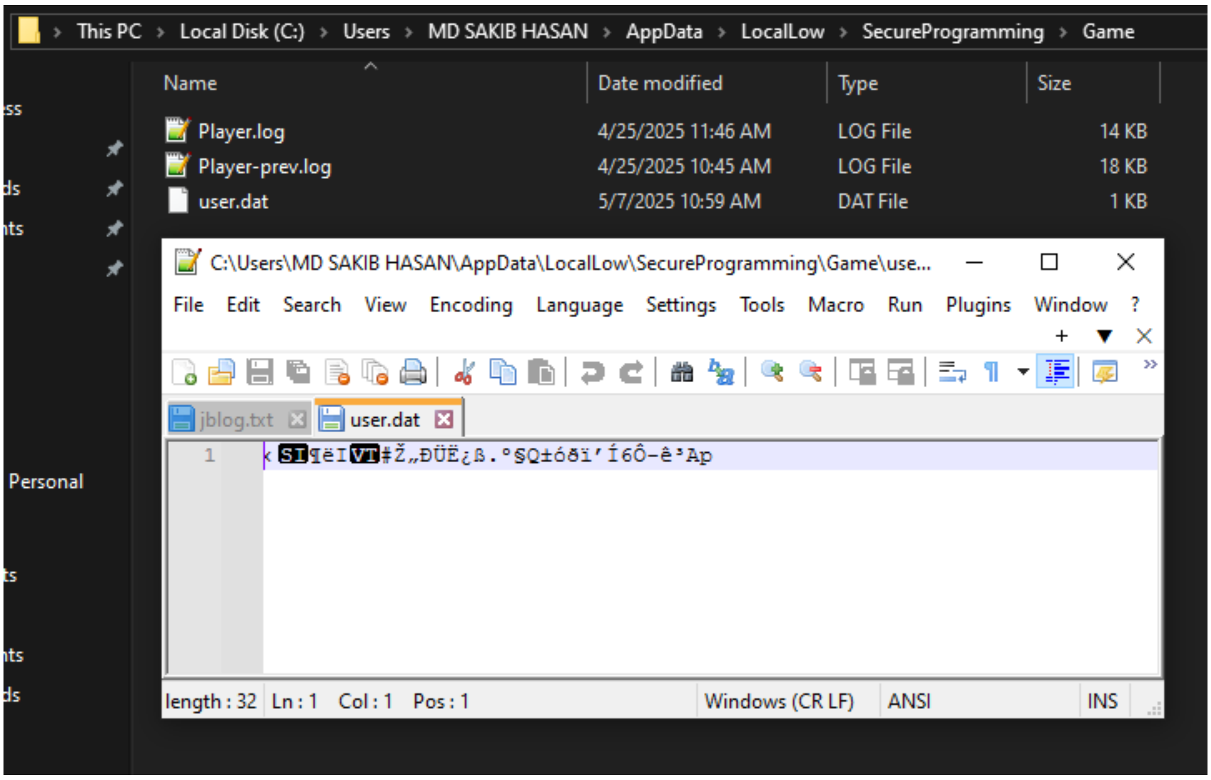
Security Aspects in Unity Game (Unity, C#)

Secure local store and Read

To make logging in simple we store the user credentials locally in a secure way. Next time user opens up the game the user sees the input fields already populated.



We store the credentials in user.dat file and that is encrypted.


This PC › Local Disk (C:) › Users › MD SAKIB HASAN › AppData › LocalLow › SecureProgramming › Game
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
jblog.txt
user.dat
Personal

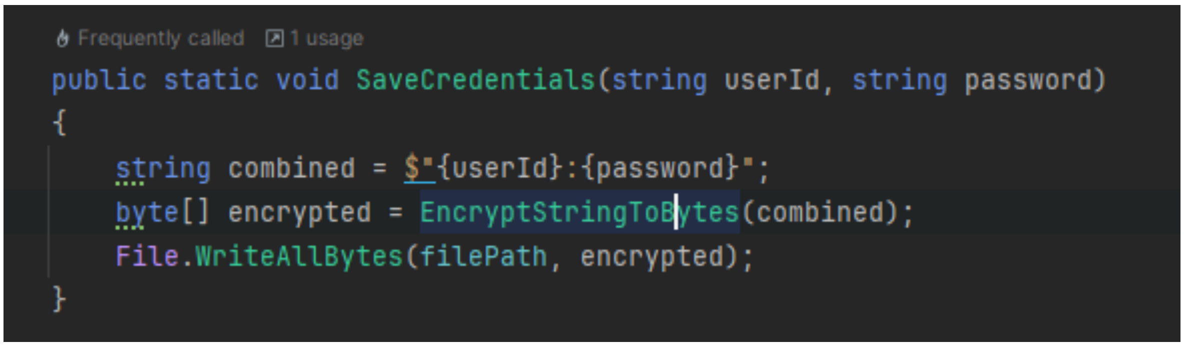
Name	Date modified	Type	Size
Player.log	4/25/2025 11:46 AM	LOG File	14 KB
Player-prev.log	4/25/2025 10:45 AM	LOG File	18 KB
user.dat	5/7/2025 10:59 AM	DAT File	1 KB

C:\Users\MD SAKIB HASAN\AppData\LocalLow\SecureProgramming\Game\use...

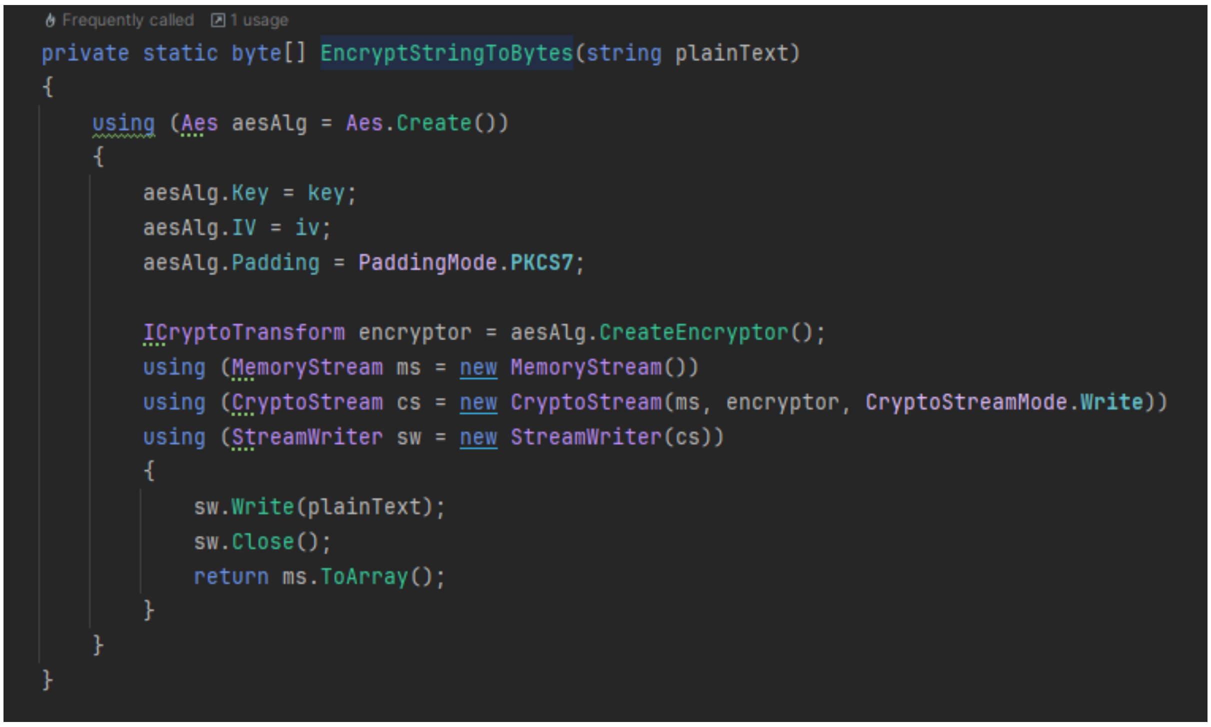
```
SI¶ëIVT‡Ž„ÐÜË¿ß.°§Q±óðï'Í6Ô–ê³Ap
```

length : 32 Ln : 1 Col : 1 Pos : 1 Windows (CR LF) ANSI INS

We used AES encryption here to securely encrypt and decrypt the user credentials.



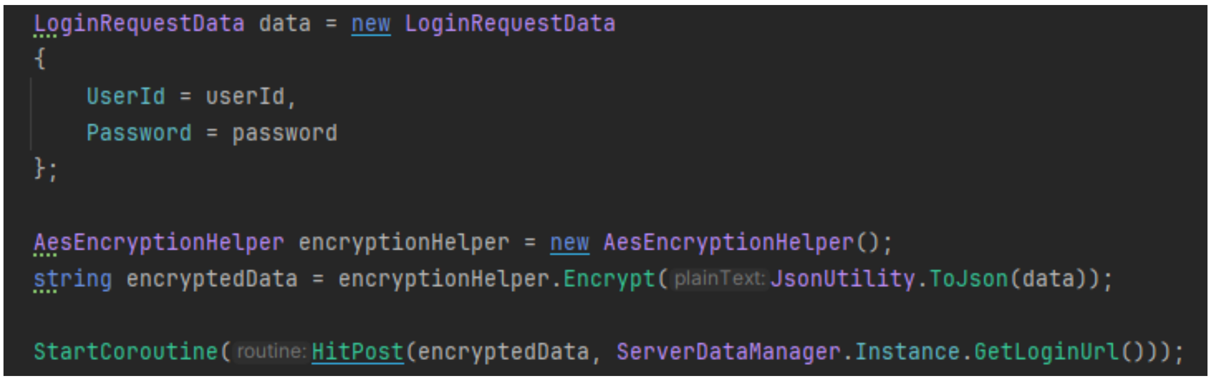
```
Frequently called   1 usage
public static void SaveCredentials(string userId, string password)
{
    string combined = $"{userId}:{password}";
    byte[] encrypted = EncryptStringToBytes(combined);
    File.WriteAllBytes(filePath, encrypted);
}
```



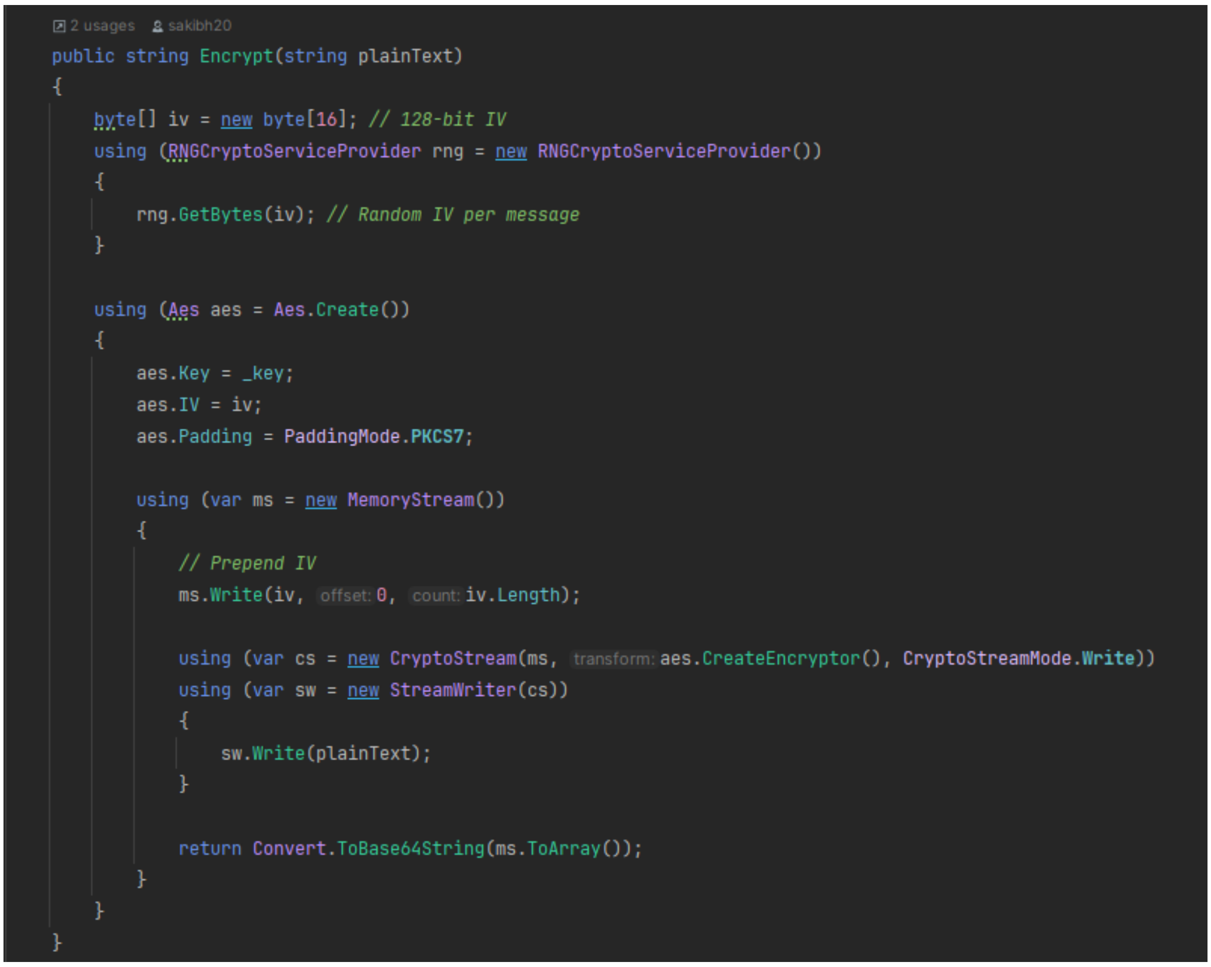
```
Frequently called   1 usage
private static byte[] EncryptStringToBytes(string plainText)
{
    using (Aes aesAlg = Aes.Create())
    {
        aesAlg.Key = key;
        aesAlg.IV = iv;
        aesAlg.Padding = PaddingMode.PKCS7;

        ICryptoTransform encryptor = aesAlg.CreateEncryptor();
        using (MemoryStream ms = new MemoryStream())
        using (CryptoStream cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write))
        using (StreamWriter sw = new StreamWriter(cs))
        {
            sw.Write(plainText);
            sw.Close();
            return ms.ToArray();
        }
    }
}
```

Secure Log In & Registration

As login and registration request contains sensitive credential information so instead of simple text, we are using encrypted request data as a form data that contains AES encrypted username, user Id, password.



```
LoginRequestData data = new LoginRequestData
{
    UserId = userId,
    Password = password
};

AesEncryptionHelper encryptionHelper = new AesEncryptionHelper();
string encryptedData = encryptionHelper.Encrypt( plainText: JsonUtility.ToJson(data));

StartCoroutine( routine: HitPost(encryptedData, ServerDataManager.Instance.GetLoginUrl()));
```

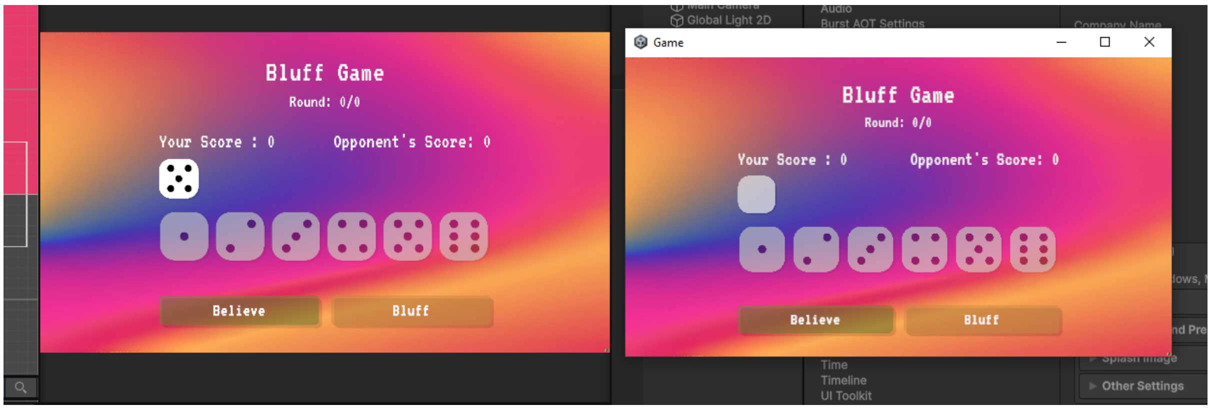


```
2 usages   sakibh20
public string Encrypt(string plainText)
{
    byte[] iv = new byte[16]; // 128-bit IV
    using (RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider())
    {
        rng.GetBytes(iv); // Random IV per message
    }

    using (Aes aes = Aes.Create())
    {
        aes.Key = _key;
        aes.IV = iv;
        aes.Padding = PaddingMode.PKCS7;

        using (var ms = new MemoryStream())
        {
            // Prepend IV
            ms.Write(iv, offset: 0, count: iv.Length);

            using (var cs = new CryptoStream(ms, transform: aes.CreateEncryptor(), CryptoStreamMode.Write))
            using (var sw = new StreamWriter(cs))
            {
                sw.Write(plainText);
            }

            return Convert.ToBase64String(ms.ToArray());
        }
    }
}
```

Protect Game State

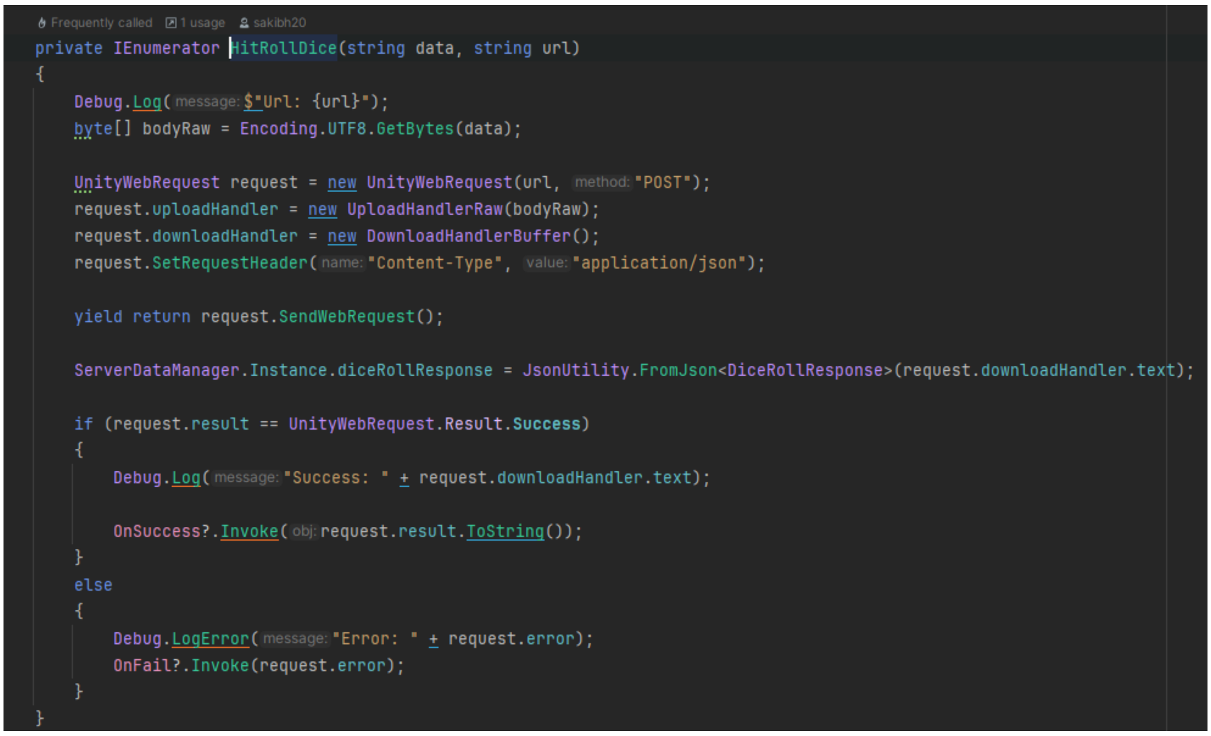
To protect game state, we are controlling availability of interactions in the UI based on the current state. Even if this front-end interaction is tempered, we are always validating the requested actions in the backend to ensure a secure state management.

The below image shows that, all the un wanted interactions are disabled based on current game situation for both players.



Prevent Cheating

We secured the dice-rolling logic on the server side, ensuring that users cannot manipulate dice outcomes or falsify game states through client-side tampering. When it is time for a players roll, front-end asks the server to do roll and give it back to the player. And the front end displays that information.



```
Frequently called   1 usage   sakibh20
private IEnumerator HitRollDice(string data, string url)
{
    Debug.Log( message: $"Url: {url}");
    byte[] bodyRaw = Encoding.UTF8.GetBytes(data);

    UnityWebRequest request = new UnityWebRequest(url,  method: "POST");
    request.uploadHandler = new UploadHandlerRaw(bodyRaw);
    request.downloadHandler = new DownloadHandlerBuffer();
    request.SetRequestHeader( name: "Content-Type",  value: "application/json");

    yield return request.SendWebRequest();

    ServerDataManager.Instance.diceRollResponse = JsonUtility.FromJson<DiceRollResponse>(request.downloadHandler.text);

    if (request.result == UnityWebRequest.Result.Success)
    {
        Debug.Log( message: "Success: " + request.downloadHandler.text);

        OnSuccess?.Invoke( obj: request.result.ToString());
    }
    else
    {
        Debug.LogError( message: "Error: " + request.error);
        OnFail?.Invoke(request.error);
    }
}
```

Prevent Statistics Data Manipulation

To prevent game stats data manipulation the whole logic to update game statistics we kept on the server end. When a game comes to a conclusion, the game service detects that and update/stores last match’s data from there. That automatically prevents any statistics related data manipulation.

Game Service (FastAPI, Python)

Overview

The game_service project is a backend service designed to manage a multiplayer game system. It provides APIs and WebSocket endpoints for real-time communication, game state management, and user interactions. The service is built using Python and the FastAPI framework, ensuring high performance, scalability, and security.

Key Features

WebSocket Communication:

- Real-time communication is enabled through WebSocket endpoints for lobby, user, and game interactions.
- WebSocket routes:
    a. **/ws/lobby:**
        i. Handles WebSocket connections for the game lobby.
        ii. Uses the handle_lobby_socket function to manage lobby-related interactions.
    b. **/ws/user/{user_id}:**
        i. Manages WebSocket connections for specific users.
        ii. Uses the handle_user_socket function to handle user-specific WebSocket interactions.
    c. **/ws/game/{user_id}/{match_id}:**
        i. Handles WebSocket connections for a specific game session.
        ii. Uses the handle_match_join function to manage player connections to a game and synchronize game state.

RESTful APIs:

Provides endpoints for game actions such as rolling dice, claiming dice, making decisions, and managing matches.

Key API routes:

- **/roll-dice (POST):**
    - Rolls a dice for the current player in a match.
    - Calls handle_game_roll_dice to update the game state with the dice roll.
- **/claim (POST):**
    - Processes a player's claim during the game.
    - Calls handle_game_claim_dice to update the game state with the claim.
- **/decide (POST):**
    - Handles a player's decision (e.g., accepting or rejecting a claim).
    - Calls handle_game_round_decide to update the game state with the decision.
- **/match/request (POST):**

        - Sends a match request to another player.
        - Calls match_request_notifier to notify the opponent.
    - **/match/accept (POST):**
        - Accepts a match request and initializes the game.
        - Calls create_game to set up the game and notify players.

Game State Management:

- The game state is managed using the GameManager class, which tracks the current turn, game process, and player actions.
- Game data is stored in memory using dictionaries (game_data and match_players).

Communication with Authentication Server:

The service integrates with an external API to update match results using the update_match_result function in services/external_requests.py.

Architecture

Framework: FastAPI

Modules:

- routers/websocket_routes.py: Defines WebSocket routes for real-time communication.
- routers/api_routes.py: Defines RESTful API routes for game actions.
- services/game.py: Contains core game logic, including game state management and player actions.
- services/external_requests.py: Handles external API requests for updating match results.

Data Models:

- services/models.py: Defines data models for requests and game states.

Game Manager:

- Manages the game lifecycle, including turns, processes, and state transitions.

Security Features

Token-Based Authentication:

- All critical API endpoints and WebSocket connections require a valid token for authentication.
- Tokens are verified using the verify_user_permission function in services/game.py, ensuring only authorized users can perform actions.

Communication From Authentication Server:

- The /match/request and /match/accept APIs are exclusively accessible by the authentication server.
- These APIs require a websecret key for verification to ensure that only authorized systems can interact with them.The websecret key is sent by the authentication server as part of the request headers.
- The server validates the websecret key against a securely stored value (e.g., environment variable or configuration file).
- If the websecret key is invalid or missing, the API responds with a 403 Forbidden error.
- This mechanism prevents unauthorized systems or users from creating or accepting matches.
- It ensures that only the authentication server can initiate or approve match-related actions, adding an additional layer of security to the game service.

Role-Based Access Control (RBAC):

- The game enforces strict turn-based access control. For example:
  - Only the current player can roll the dice.
  - Only the opponent can make a claim.
- Unauthorized actions result in 403 Forbidden errors.
- Users are verified with their temporary match-based token.

Data Validation:

- All incoming requests are validated using data models defined in services/models.py.
- Invalid or malformed requests are rejected with appropriate error messages.

WebSocket Security:

- WebSocket connections are validated to ensure only authorized users can join game sessions.
- Disconnection events are logged, and future improvements will include handling reconnections securely.

External API Security:

- The update_match_result function uses secure headers, including a Bearer token, to authenticate requests to the external API.
- Payloads are sent in JSON format with proper content-type headers.

Error Handling:

- Comprehensive error handling is implemented to prevent information leakage.
- Sensitive errors (e.g., invalid tokens) return generic error messages to avoid exposing internal logic.

Input Sanitization:

- All user inputs, such as match IDs, user IDs, and tokens, are sanitized to prevent injection attacks.

Logging and Monitoring:

- All critical actions, such as WebSocket connections, API calls, and game state changes, are logged for auditing and debugging purposes.
- Logs include timestamps, user IDs, and action details.

Challenges Addressed

Real-Time Communication:

- Implemented WebSocket endpoints to handle real-time interactions between players.

Game State Synchronization:

- Used in-memory data structures to synchronize game states across multiple players.

Security:

- Verified user permissions using tokens to ensure only authorized players can perform actions.
- Enforced strict turn-based access control to prevent unauthorized actions.

GameManager Class Overview

The `**GameManager**` class is a core component of the game service, responsible for managing the lifecycle, state transitions, and player interactions during a game. It ensures that the game progresses according to the rules and handles communication with players through WebSocket connections.

Key Responsibilities

1. Game Initialization:

- The `GameManager` is initialized with a `Game` object containing details such as players, match ID, and initial game state.

2. Game State Management:

- Tracks the current state of the game, including the current turn, process (e.g., rolling, claiming, deciding), and scores.

- Updates the game state as players perform actions.

3. Turn Management:

- The `next_turn` method advances the game to the next phase or turn, ensuring the correct sequence of actions (rolling, claiming, deciding, etc.).

- Handles transitions between game processes and determines when the game is over.

4. Player Interaction:

- Sends commands to players (e.g., "roll_dice", "claim_dice") using WebSocket connections.

- Notifies players of game events such as match start, round over, and game over.

5. Game Completion:

- Determines the winner based on scores and updates the game status to `FINISHED`.

- Calls the `update_match_result` function to report the match outcome to an external API.

Key Methods

1. is_game_over:

- Checks if the game has reached its conclusion based on the number of rounds or scores.

2. next_turn:

- Advances the game to the next turn or phase.
- Handles different game processes (rolling, claiming, deciding, round over) and updates the game state accordingly.

3. handle_rolling, handle_claiming, handle_decide:

- Manages specific game processes by sending appropriate commands to the current player.

4. send_command:

- Sends a JSON command to a player's WebSocket connection, enabling real-time communication.

5. get_winner:

- Determines the winner of the game based on scores.

6. notify_match_start, notify_round_over, notify_game_over:

- Sends notifications to players about key game events.

7. update_game_state:

- Updates the game state based on the current process and transitions to the next phase.

8. get_opponent:

- Retrieves the opponent of the current player.

Game Flow

1. Match Start:

- The game begins with the `notify_match_start` method, informing players of the match details.

2. Turn-Based Actions:

- Players take turns performing actions (rolling, claiming, deciding) as directed by the `next_turn` method.

3. Round Completion:

- At the end of each round, scores are updated, and the `notify_round_over` method informs players of the round results.

4. Game Completion:

- When the game ends, the `notify_game_over` method announces the winner, and the match result is reported to an external API.

Integration with External Systems

- The `update_match_result` function is called upon game completion to send match details (e.g., scores, winner) to an external API for record-keeping.

Error Handling

- Ensures that invalid actions (e.g., playing out of turn) are rejected with appropriate error messages.
- Handles game-over scenarios gracefully by preventing further actions and notifying players.

Security Aspects in Authentication Server (Asp .Net Core 8.0)

This part is developed and managed by Samith Binda Pantho.

I. Security Aspects in Architecture

1. Layered Architecture with Responsibility Separation

- **SecureService.API:** Entry point, enforces authentication, request validation, rate limiting, IP filtering, and CORS policies.
- **SecureService.BLL:** Business logic, handles permissions, input sanitization, and enforces security policies.
- **SecureService.DAL/Context:** Deals with data access, providing protection against SQL Injection via ORM.
- **SecureService.Entity:** Defines view models and DTOs for safe data transfer (preventing over-posting and mass assignment).
- **SecureService.Logging:** Provides logging, useful for auditing and detecting intrusions or abuse.

Benefit: Isolation of responsibilities limits the blast radius of vulnerabilities.

2. JWT-Based Authentication

- Uses JSON Web Tokens (JWT) to authorize requests.
- Includes:
  - Token expiration
  - Encrypted User ID
  - Encrypted Session ID

Benefit: Prevents session hijacking, ensures stateless and scalable auth.

3. IP Filtering Middleware

- Restricts access to specific IP addresses or subnets (e.g., internal APIs).

Benefit: Limits attack surface by blocking unauthorized network locations.

4. Strong Typing & DTO Usage

- Use of models like LoginViewModel.cs, RegistrationViewModel.cs helps sanitize and validate user input.
- Prevents exposing internal DB models directly to the API.

Benefit: Reduces risks of over-posting attacks and leaking sensitive internal schema.

5. Logging

- Helps in tracking user activity, errors, and potential malicious behaviour.
- Ensure:
  - No sensitive data (e.g., passwords, tokens) are logged.
  - Logs are securely stored and access controlled.

Benefit: Critical for audit trails and forensic analysis post-incident.

6. Dependency Injection

- Proper use helps prevent issues like service misconfiguration and allows for central enforcement of policies (e.g., using secure HTTP clients, database context with retry policies, etc.).

Benefit: Centralizes security-critical service configuration.

II. Security Aspects in Different Rest APIs

- *Registration*
  - **Encrypted Input Handling:** The client is expected to send encrypted data, which protects sensitive data like passwords in transit.
  - **Strong Input Validation**: Validating inputs using regular expressions:
    - UserId Validation: The UserId is validated using a regular expression: @"^[A-Za-z0-9]{1,10}$". This ensures that the user ID is alphanumeric and no longer than 10 characters. This prevents malicious inputs that could exploit the system with unexpected characters or overly long strings.
    - Email Validation: Regex is used to perform a basic email format validation in the registration logic. This regular expression ensures that the input contains one or more characters before and after the @ symbol, and includes a period (.) to separate the domain and top-level domain.
    - Password validation:
      - Length check: Ensures the password is at least 8 characters long.
      - Uppercase letter check: Ensures the password contains at least one uppercase letter ([A-Z]).
      - Lowercase letter check: Ensures the password contains at least one lowercase letter ([a-z]).
      - Numeric check: Ensures the password contains at least one digit (\d).
      - Special character check: Ensures the password contains at least one special character (e.g., @, #, $, etc.).
  - **Password Hashing:** Used hashing before saving passwords into database.
  - **Cross-Site Scripting (XSS) Protection:** The code uses **parameterized queries** to interact with the database (e.g., Where(it => it.UserId.Equals(registrationModel.UserId))), which helps mitigate risks related to SQL injection attacks.
  - **Prevention of Duplicate Accounts:** The code checks for existing users by both UserId and Email. This helps to prevent account duplication and also reduces the risk of an attacker trying to use a duplicate email or user ID to impersonate another user:
    - UserId is checked to ensure that no duplicate user IDs exist in the database.
    - Email is checked to ensure no duplicate email addresses are used.

- **Error Handling and Logging:** The code wraps the registration logic inside a try-catch block. If an error occurs during registration (e.g., validation failure, database issues), the exception is caught and logged.
- **Database Integrity:** The code ensures that the user registration process is only successful if all validation checks pass, and if the user is successfully inserted into the database (_context.SaveChanges() > 0).
- **Use of Secure Services:** The service is configured via dependency injection.
- **Endpoint Details**:

Endpoint	../api/Register
Method	POST
Header	
Request	encryptData (as form-data) EX: cgFQ4bSXq+8qzB1AvGQBBIte4QIl1PeSa8m+4Gc....df

Response Body	{
	"Status": "OK",
	"Message": "Successfully Registered.",
	"Result": null
	}

Error Body	{
	"Status": "FAILED",
	"Message": " sourav is already existed.",
	"Result": null
	}

Handled Errors	• Invalid User ID. User ID should be within 10 digits and contain only alphanumeric characters. • Invalid Password. Confirm password doesn't match with provided password. • Password must be at least 8 characters long. • Password must contain at least one uppercase letter. • Password must contain at least one lowercase letter. • Password must contain at least one digit. • Password must contain at least one special character (e.g., @, #, $, etc.). • Invalid Email. • Xyz is already existed. • Xyzmail.com is already existed. • XYZ is already existed.

- *Login*
  - **Password Hash Comparison Using FixedTimeEquals:** Prevents **timing attacks** by ensuring comparisons take constant time.

- **Account Lockout After 3 Failed Attempts:** Protects against brute-force password guessing.
- **Encrypted Credentials from Client:** Prevents exposure of sensitive login data during transmission.
- **JWT and Refresh Token Usage:** Adheres to modern token-based authentication practices. Refresh tokens enable longer sessions securely.
- **Session Management:** Prevents multiple simultaneous active sessions, reducing the attack surface.
- **Error Logging:** Provides traceability and accountability for failed login attempts and errors.
- **Session ID Entropy:** Using cryptographically secure random session id generation.
- **Endpoint Details:**

Endpoint	../ api/Login
Method	POST
Header	
Request	encryptData (as form-data) EX: cgFQ4bSXq+8qzB1AvGQBBIte4QIl1PeSa8m+4Gc....df
Response Body	{ "Status": "OK", "Message": "Login Successfull.", "Result": { "useID": "sourav", "email": "saurav.paul@hotmail.com", "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....", "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....", "accessTokenExpiry": "2025-04-20T02:26:26.1598726+03:00", "refreshTokenExpiry": "2025-04-27T02:11:26.5606252+03:00" } }
Error Body	{ "Status": "FAILED", "Message": "Wrong Password.", "Result": null }
Handled Errors	• Account has been Deactivated. Please contact with Admiistrator. • Account has been Deactivated, because of 3 times wrong password attempt. • Wrong Password. • Login Failed.

- *Login using Refresh Token*
    - **Date Handling:** Using DateTime.UtcNow to avoid time zone-related inconsistencies and potential abuse if servers are in different time zones.
    - **Claim Validation and Decryption:** Validates presence of claims (UserId, TokenID, ExpiryDate) before proceeding. Prevents null reference exceptions and ensures the identity has necessary information. Uses a secure algorithm for critical information (UserId, TokenID) decryption.
    - **Refresh Token Expiry Check:** The method explicitly checks if the refresh token has expired using ExpiryDate. Prevents reuse of expired tokens and enforces token lifecycle management.
    - **Validation of Active Refresh Token in DB:** Verifies that the refresh token is active in the database before allowing session creation. Prevents unauthorized reuse of revoked or inactive refresh tokens.
    - **Revocation of Previous Sessions:** Iterates through previous sessions and sets IsActiveSessionFlag to false before starting a new one. Helps prevent session fixation or multiple simultaneous active sessions for the same user.
    - **Creation of New Session with UUID:** Generates a new session with a GUID as SessionID. Unique session IDs reduce the risk of session prediction or replay attacks.
    - **Strict Token and User Validity Checks:** Defense-in-depth, helps reduce attack surface even if one layer is bypassed. Multiple layers of checks to ensure:
        - User ID is valid.
        - Token is active.
        - Token and user match.
    - **Error Handling:** Uses clear exception paths to stop processing on validation failure
    - **Endpoint Details:**

Endpoint	../ api/GenerateAccessTokenByRefreshToken
Method	GET
Header	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....
Request	

Response Body	{
	"Status": "OK",
	"Message": "Generated Access Token",
	"Result": {
	"useID": "sourav",
	"email": "saurav.paul@hotmail.com",
	"accessToken":
	"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....",
	"refreshToken":
	"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.... ",
	"accessTokenExpiry": "2025-04-
	20T02:31:35.45289+03:00",

	"refreshTokenExpiry": "2025-04-27T02:11:27"
	  }
	}

Error Body	{
	  "Status": " UNAUTH",
	  "Message": "Token Deactivated.",
	  "Result": null
	}

Handled Errors	• Valid Refresh Token Required. • Refresh Token Expired. • Token Deactivated. • Invalid Token.

- *Access Token Validation for Further Endpoints*
  - **Claims Decryption:** Decrypting claims (like UserId and SessionID) before use helps ensure that token contents were not tampered with. Ensures data integrity and confidentiality of token claims**.**
  - **Token Expiry Check:** Explicit check for token expiration against nowDate. Prevents the use of expired tokens, enforcing session time limits.
  - **Session Validation in Database:** Ensures that tokens cannot be reused after logout or session invalidation. Access token is valid only if:
    - The session ID exists in DB.
    - It is marked active (IsActiveSessionFlag == true).
  - **Multi-layer Validation:** Implements defense in depth against forged tokens or manipulated claims. Method validates:
    - Identity presence.
    - Decrypted claim values.
    - Expiration time.
    - Session and user presence in DB.
  - **Fails Fast on Invalid Inputs:** Limits attack surface and resource usage. Multiple throw new Exception("...") statements ensure:
    - Early exit on failure.
    - Minimal processing on invalid/expired/deactivated tokens.
  - **Token Details:**

    JWT Access Token:

    eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJVc2VySWQiOiJ0UjIzRiszZ3RFVUYrcXcvYTZoQlphbmJlMVUzSUZkOVRveFQvY25oZXVrPSIsIlNlc3Npb25JRCI6InRSMjNGKzNndEVVRitxdy9hNmhCWld4L0tCeDljc1kzK213cFAveTlOaHNVTGFFOExObjUvTVdlcU5ZL1hDbmNNVGJJR1VrUlQxdWRaeHdHZS9pYzNnPT0iLCJTdGFydERhdGUiOiIwNS1NYXktMjAyNSAyMzozOTo1MSIsIkV4cGlyeURhdGUiOiIwNS1NYXktMjAyNSAyMzo1NDo1MSIsImV4cCI6MTc0NjQ4OTI5MSwiaXNzIjoiU2VjdXJlU2VydmljZSIsImF1ZCI6IlNlY3VyZVNlcnZpY2UifQ.GIDzJYl1g0L0W9JfFW6H6ocgqLTgLr5BDsClr6Detqs

Decoded JWT Token:

```
{
  "UserId": "tR23F+3gtEUF+qw/a6hBZanbe1U3IFd9ToxT/cnheuk=",
  "SessionID":
"tR23F+3gtEUF+qw/a6hBZWx/KBx9csY3+mwpP/y9NhsULaE8LNn5/MWHqNY/XC
ncMTbIGUkRT1udZxwGe/ic3g==",
  "StartDate": "05-May-2025 23:39:51",
  "ExpiryDate": "05-May-2025 23:54:51",
  "exp": 1746489291,
  "iss": "SecureService",
  "aud": "SecureService"
}
```

- *Logout after Access Token Validation*
    - **Session Invalidation:** Disables all active sessions (IsActiveSessionFlag = false) for the user. Prevents further use of the access token tied to those sessions — essential for proper logout security.
    - **Refresh Token Revocation:** Deactivates all active refresh tokens (IsActive = false) for the user. Prevents long-lived session hijacking or token reuse after logout.
    - **Database Update for Both Sessions and Tokens:** Updates records explicitly in the database with .Update() and .SaveChanges(). Ensures logout changes are persisted immediately and atomically.
    - **Graceful Handling of Null User:** Handles the edge case where logged in user is null without crashing. Avoids null reference exceptions and ensures graceful behaviour under misuse.
    - **Exception Logging:** Catches exceptions and logs detailed error information including stack trace.
    - **Endpoint Details:**

Endpoint	../api/Logout
Method	GET
Header	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9......
Request	
Response Body	{ "Status": "OK", "Message": "Logged Out Successfully.", "Result": null }
Error Body	{ "Status": " UNAUTH", "Message": "Token Deactivated.", "Result": null }

Handled Errors	• Valid Token Required. • Refresh Token Expired. • Token Deactivated. • Invalid Token. • Logged Out Failed.

- *Initialize Match Request after Access Token Validation*
  - **Data Integrity (Match ID Generation):** Guid.NewGuid().ToString() ensures a unique, hard-to-guess match ID. Helps prevent ID spoofing or collisions.
  - **Encrypted Session ID:** The SessionID is hashed using _cls.EncryptSha256Hash(...) before being transmitted, by avoiding the use of plaintext session identifiers, this approach helps mitigate the risk of session ID leakage and potential session hijacking. This hashed value is used for communication between the application and the game engine via WebSocket for Player 1.
  - **Exception Logging:** Exceptions are logged with full stack trace.
  - **Endpoint Details:**

Endpoint	../api/InitializeMatchRequest
Method	GET
Header	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....
Request	?playerID=samith

Response Body	{
	  "Status": "OK",
	  "Message": "sourav has requested samith for a Match.",
	  "Result": null
	}

Error Body	{
	  "Status": " UNAUTH",
	  "Message": "Token Deactivated.",
	  "Result": null
	}

Handled Errors	• Valid Token Required. • Refresh Token Expired. • Token Deactivated. • Invalid Token. • Invalid Player ID.

- *Accept or Decline Match Request after Access Token Validation*
  - **Ownership Check:** Ensures only Player2 can respond to the match request. Prevents unauthorized users from accepting or rejecting matches.
  - **Match Existence and Status Check:** Validates that the match exists and is still "PENDING". Prevents actions on stale or already processed match records.

- **Match Token Generation:** Uses a JWT for match authorization. Tokens provide a secure and verifiable way to confirm a match's legitimacy.
- **Encrypted Session ID:** The SessionID is hashed using _cls.EncryptSha256Hash(...) before being transmitted, by avoiding the use of plaintext session identifiers, this approach helps mitigate the risk of session ID leakage and potential session hijacking. This hashed value is used for communication between the application and the game engine via WebSocket for Player 2.
- **Exception Logging:** Exceptions are logged with full stack trace.
- **Endpoint Details:**

Endpoint	../api/ResponseMatchRequest
Method	POST
Header	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....
Request	{ "MatchId": "0f915ef7-8feb-4ba4-95de-976b41e67b6f", "Player1": "sourav", "Player2": "samith", "MatchStatus": "ACCEPTED" } OR { "MatchId": "0f915ef7-8feb-4ba4-95de-976b41e67b6f", "Player1": "sourav", "Player2": "samith", "MatchStatus": "DECLINED" }
Response Body	{ "Status": "OK", "Message": "samith has accepted sourav's Match Request.", "Result": null } OR { "Status": "OK", "Message": "samith has declined sourav's Match Request.", "Result": null }
Error Body	{ "Status": "FAILED", "Message": "Match is already accepted", "Result": null }

Handled Errors	• Valid Token Required. • Refresh Token Expired. • Token Deactivated. • Invalid Token. • Invalid match Information. • Match is already accepted. • Match is already declined. • Match is already finished. • Invalid Match Status. • Invalid Player 2.

- *Update Match Result (Call from Game Engine (Fast API))*
    - **JWT Validation with Expiry**: Uses ExpiryDate to prevent token reuse after expiration.
    - **Decryption & Claim Extraction**: Claims like UserId and SessionID are encrypted and then decrypted.
    - **User-Match Consistency Check**: Verifies the decrypted UserId (e.g., player1#player2) and the match ID with DB state.
    - **Ensures Match Is Ongoing**: Checks EndTime == null to ensure it's not reused.
    - **Match Status Check**: Prevents update unless match is in "ACCEPTED" state.
    - **Match ID Cross-Verification**: Ensures that the update is happening only for the expected match.
    - **Transactional Update**: Updates DB only after checks and uses EF tracking.
    - **Exception Logging:** Exceptions are logged with full stack trace.
    - **Endpoint Details:**

Endpoint	../api/UpdateMatchResult
Method	POST
Header	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9....

Request	{
	"MatchId": "0f915ef7-8feb-4ba4-95de-976b41e67b6f",
	"Player1Moves": 5,
	"Player2Moves": 6,
	"Winner": "samith"
	}

Response Body	{
	"Status": "OK",
	"Message": "Match Info Updated.",
	"Result": null
	}

Error Body	{ "Status": " UNAUTH",

	"Message": "Token Deactivated.", "Result": null }
Handled Errors	• Valid Token Required. • Refresh Token Expired. • Token Deactivated. • Invalid Token. • Invalid Match ID. • Match status is pending. • Match status is declined. • Match status is finished.

- *Fetch Leaderboard after Access Token Validation*
  - **No External Input Used in Query:** The method doesn't use any user-provided query parameters, so it's not vulnerable to injection via LINQ or SQL at this level.
  - **Handles Missing User Gracefully:** If the user is not in the leaderboard (i.e., hasn't won any games), the method creates a default response instead of throwing an error**.**
  - **No Sensitive Information Returned:** Only public game stats (username and wins) are returned. No email, session info, or personal data is exposed.
  - **Endpoint Details:**

Endpoint	../api/FetchLeaderBoard
Method	GET
Header	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Request	

Response Body	{
	  "Status": "OK",
	  "Message": "Leaderboard fetched successfully.",
	  "Result": {
	    "topUsers": [
	      {
	        "position": "1",
	        "player": "samith",
	        "wins": 1
	      },
	      {
	        "position": "2",
	        "player": "sourav",
	        "wins": 1
	      }
	    ],
	    "user": {
	      "position": "-",
	      "player": "samith3",
	      "wins": 0
	    }
	  }

	}

Error Body	{
	"Status": " UNAUTH",
	"Message": "Token Deactivated.",
	"Result": null
	}

Handled Errors	• Valid Token Required. • Refresh Token Expired. • Token Deactivated. • Invalid Token.

- *Fetch Match History after Access Token Validation*
  - **Restricted by Authenticated User:** Restricting the data to only that user's matches.
  - **Server-side Filtering:** Matches are filtered server-side using UserId and EndTime != null, minimizing unnecessary data exposure.
  - **Endpoint Details:**

Endpoint	../api/FetchMatchHistory
Method	GET
Header	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Request	

Response Body	{
	"Status": "SUCCESS",
	"Message": "Matches History fetched successfully.",
	"Result": [
	{
	"matchId": "2a1e2e25-32cf-49cd-a681-6c7c592d0bd5",
	"player1": "samith",
	"player2": "sourav",
	"startTime": "2025-04-06T23:23:58",
	"endTime": "2025-04-06T23:27:40",
	"player1Moves": 5,
	"player2Moves": 6,
	"winner": "sourav"
	}
	]
	}

Error Body	{
	"Status": "UNAUTH",
	"Message": "Token Deactivated.",
	"Result": null
	}

Handled Errors	

- *Global Concerns*
    - **AES Encryption Used:** Using AES for symmetric encryption, which is a strong standard.
        - **Key Hashing with SHA-256:** Instead of using the raw key from the environment, hashed it with SHA-256 to ensure it's 256-bit.
        - **Environment-based Key Management:** Sensitive information (like encryption keys) is not hardcoded. This improves security and allows for environment-specific configuration.
    - **Dependency Injection:** By injecting dependencies through the constructor, the design promotes **testability** and **flexibility**. This approach decouples the class from its dependencies, making it easier to substitute real implementations with mocks or stubs during testing. It also allows for greater flexibility, as the class can work with different implementations of its dependencies without needing to be modified, adhering to the Open/Closed Principle in object-oriented design. This practice enhances maintainability and enables easier unit testing of components in isolation.
    - **Memory Caching Readiness:** Use of IMemoryCache for performance optimization intentions.
    - **Supports Multiple Date Formats:** Considered diverse input formats in ConvertToDateTime(), a user-centric decision, particularly useful in multi-regional applications.
    - **SHA-256 Hashing:** A simple, fast, and secure for generating encrypted password storage or token signing.
    - **Proper Error Logging:** Consistently in logging errors, which is critical for production diagnostics and security auditing.
    - **CORS Configuration:** CORS is properly configured with the flexibility to allow any origin, method, and header. This makes the API more versatile while keeping security controls in place.
    - **MySQL Database Integration:** Database connection is being configured using a custom extension method, which allows the flexibility of changing the database configurations without cluttering the Startup class.
    - **Swagger API Documentation:** Enabled Swagger for API documentation and added JWT authentication to Swagger, which is useful for developers consuming your API.
    - **Rate Limiting:** Using rate limiting with AspNetCoreRateLimit to prevent abuse of your API, which adds an important layer of security and resource management. Configured this rate limiting on a per-endpoint basis.

III. Security Aspects in Docker (backend , mysql)

- *Backend*
    - **Use of Docker Volumes:** The backend service is configured to persist error logs (./backend/Error:/app/Error). Keeping logs separate from the container and ensuring that they are available after container restarts helps in auditing and troubleshooting. However, it's important to ensure that sensitive information is not being logged.
    - **Network Isolation in Docker:** The backend and MySQL services are part of the secure-network, which isolates them from other containers on the host. This restricts unwanted access from external containers, ensuring that only trusted services on this network can interact with the backend.
    - **Minimal Service Exposure:** The service exposes only the necessary ports (port 8080) and does not expose other ports, ensuring that services are not exposed to the public unnecessarily.

- *Mysql*
    - **Root Password Protection:** The MySQL root password is set in the environment (MYSQL_ROOT_PASSWORD), which ensures that unauthorized users cannot access the MySQL instance unless they have the root password.
    - **Custom Authentication Plugin:** The use of --default-authentication-plugin=mysql_native_password ensures compatibility with older MySQL clients and can improve security when dealing with known vulnerabilities in newer authentication mechanisms (caching_sha2_password). It also allows greater control over authentication methods used.
    - **Database Initialization Scripts:** The ./Database/mysql/init volume allows for custom database initialization. This can include creating necessary users, setting proper access controls, and enforcing a secure schema on initial start-up.
    - **Database Data Persistence:** The use of a dedicated volume (mysql_data:/var/lib/mysql) ensures that MySQL data is persisted and survives container restarts. This is critical to prevent data loss and ensure that sensitive data is stored securely.
    - **Container Restart Policy:** The restart: always option ensures that the MySQL service restarts automatically in case of failures, which is important for ensuring availability and preventing downtime.

IV. Testing

To ensure the SecureService application functions as expected, I have used Postman as our primary API testing tool. Postman allowed me to interact with the RESTful endpoints of the ASP.NET Core backend in a controlled and repeatable environment. Below are the key functionalities we tested:

- **Authentication & Authorization**
    - **User Login**: Sent POST requests with valid and invalid credentials to the /api/Login endpoint to verify JWT issuance and error handling.

  - **Token Validation**: Tested access to secured endpoints by including or omitting the Bearer token in headers.
  - **Expired Token Handling**: Simulated expired tokens to check for proper 401 Unauthorized responses and custom headers like Token has Expired.

- **CRUD Operations**
  - **Register**: Used POST requests to test the /api/Register, sending valid JSON payloads into the endpoint, validating response codes (e.g., 200).
  - **Initialize Match Request**: Used GET requests to test the /api/ InitializeMatchRequest, also validating response codes (e.g., 200).
  - **Response Match Request**: Used POST requests to test the /api/ResponseMatchRequest, sending valid JSON payloads into the endpoint validating response codes (e.g., 200).
  - **Retrieve Match Details**: Sent GET requests to endpoints like /api/FetchUserInfo, /api/FetchLeaderBoard and /api/FetchMatchHistory to verify correct retrieval of individual and multiple records.
  - **Update Match Result**: Verified POST requests to modify existing records, and ensured validation rules were enforced for /api/UpdateMatchResult.
  - **Logout**: Sent GET requests to remove existing sessions and refresh tokens and confirmed proper status codes and database updates for /api/Logout.

- Rate Limiting Behavior
  - Using a script to repeatedly hit a specific endpoint, we confirmed the API enforced the IP rate limiting rule (e.g., 5 requests per minute) and responded with HTTP 429 after the limit was exceeded.

- **Error Handling & Edge Cases**
  - **Invalid JSON**: Tested malformed request bodies to ensure the API returns appropriate validation messages.
  - **Missing Parameters**: Verified that missing required fields produce clear error messages.
  - **Unauthorized Access**: Ensured restricted endpoints return proper 403 status when accessed without authentication.

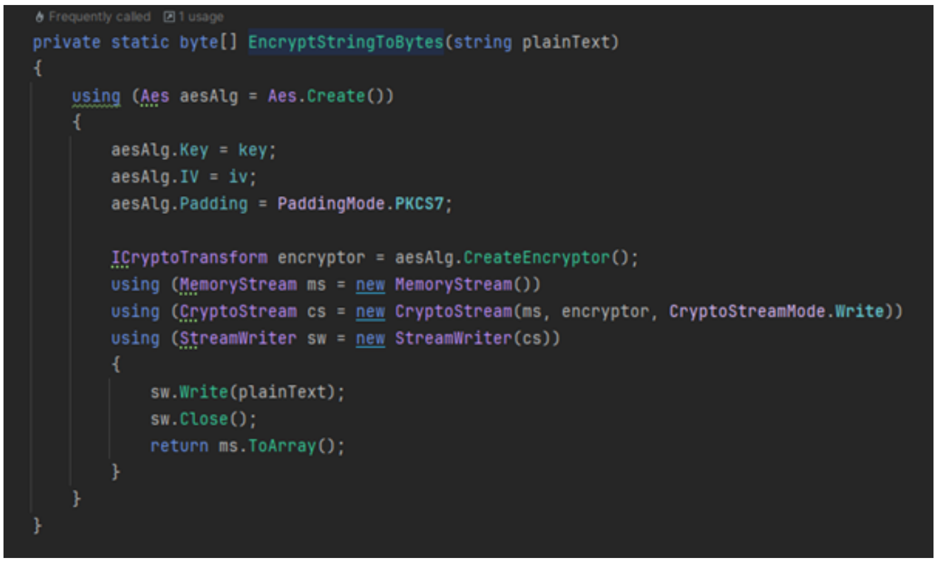
- **Security Headers**
  - Validated HTTP responses to ensure security headers were present (e.g., Authorization, custom headers on token failure).

Use of AI

I. Game

- **AES Encryption Implementation**

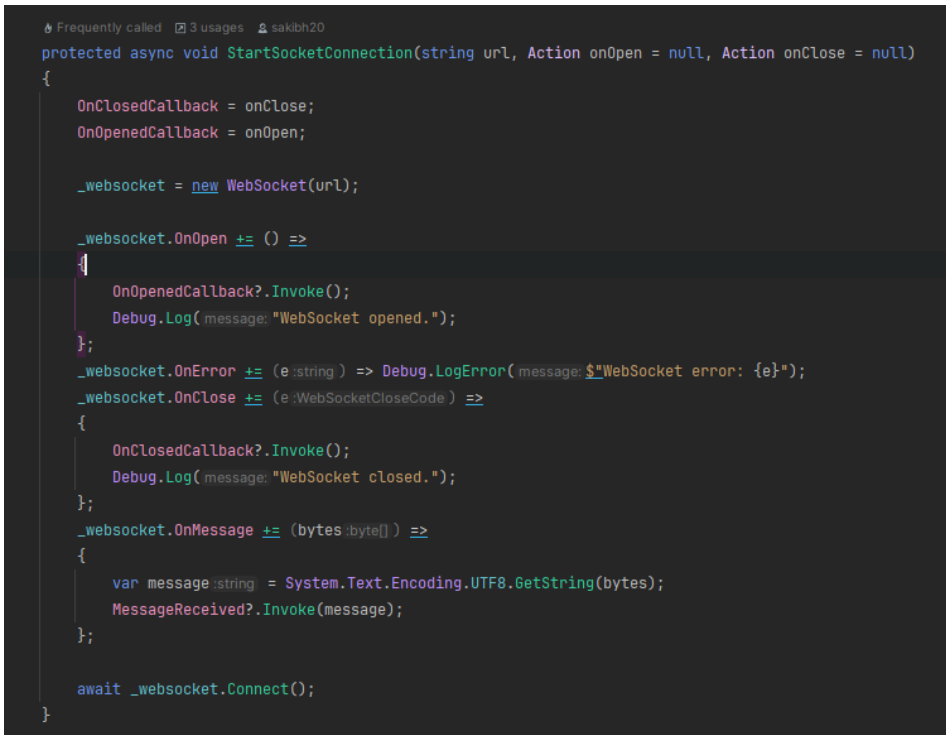
I used ChatGPT assistance to implement AES-256 encryption in C# for securely storing local user credentials. The AI helped adapt backend cryptographic logic to Unity, including key/IV handling and secure file storage practices.



```
private static byte[] EncryptStringToBytes(string plainText)
{
    using (Aes aesAlg = Aes.Create())
    {
        aesAlg.Key = key;
        aesAlg.IV = iv;
        aesAlg.Padding = PaddingMode.PKCS7;

        ICryptoTransform encryptor = aesAlg.CreateEncryptor();
        using (MemoryStream ms = new MemoryStream())
        using (CryptoStream cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write))
        using (StreamWriter sw = new StreamWriter(cs))
        {
            sw.Write(plainText);
            sw.Close();
            return ms.ToArray();
        }
    }
}
```

- **WebSocket Connection Setup**

AI support was used to set up and debug the WebSocket client in Unity using the NativeWebSocket library.



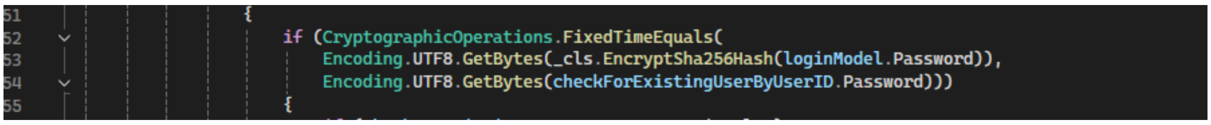
```
protected async void StartSocketConnection(string url, Action onOpen = null, Action onClose = null)
{
    OnClosedCallback = onClose;
    OnOpenedCallback = onOpen;

    _websocket = new WebSocket(url);

    _websocket.OnOpen += () =>
    {
        OnOpenedCallback?.Invoke();
        Debug.Log( message: "WebSocket opened.");
    };
    _websocket.OnError += (e :string ) => Debug.LogError( message: $"WebSocket error: {e}");
    _websocket.OnClose += (e :WebSocketCloseCode ) =>
    {
        OnClosedCallback?.Invoke();
        Debug.Log( message: "WebSocket closed.");
    };
    _websocket.OnMessage += (bytes :byte[] ) =>
    {
        var message :string = System.Text.Encoding.UTF8.GetString(bytes);
        MessageReceived?.Invoke(message);
    };

    await _websocket.Connect();
}
```

II. Game Engine

- Used to do research about the websocket connection
- Debugging websocket connection issue
- Setting up docker
- Help taken to write the report

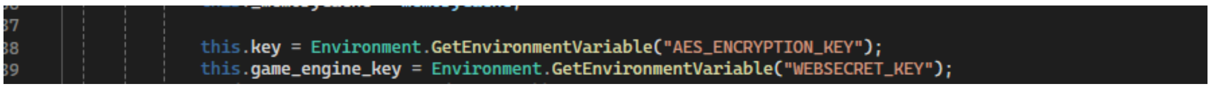
III. Authentication Server (ASP .NET Core 8.0)

- **Password Hash Comparison Using FixedTimeEquals:** I have previously checked password with the given password after hashing it, but Chat GPT 4.o suggested that hash comparison should be done using FixedTimeEquals. So that, a **constant-time comparison** function such as CryptographicOperations.FixedTimeEquals() in .NET, compares byte arrays without leaking timing information.



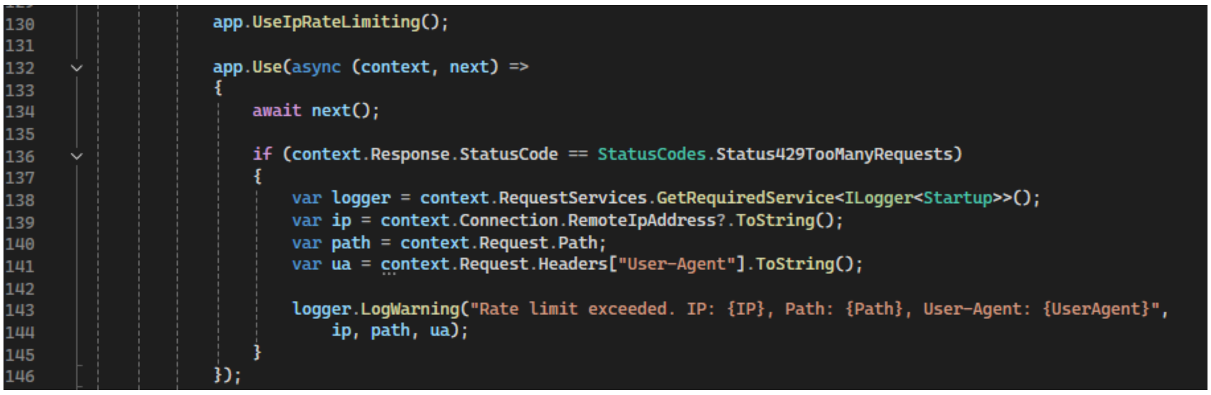
```
{
    if (CryptographicOperations.FixedTimeEquals(
        Encoding.UTF8.GetBytes(_cls.EncryptSha256Hash(loginModel.Password)),
        Encoding.UTF8.GetBytes(checkForExistingUserByUserID.Password)))
    {
```

- **Environment-based Key Management:** Previously I have hard-coded the following keys into the code, but Chat GPT 4.o suggested that managing API keys, secrets, and other sensitive configurations via environment variables is a best practice for security, portability, and maintainability.



```
this.key = Environment.GetEnvironmentVariable("AES_ENCRYPTION_KEY");
this.game_engine_key = Environment.GetEnvironmentVariable("WEBSECRET_KEY");
```

- **Rate Limiting:** From the suggestion from Chat GPT 4.o, I have added rate limiting on a per-endpoint basis in this solution.



```
app.UseIpRateLimiting();

app.Use(async (context, next) =>
{
    await next();

    if (context.Response.StatusCode == StatusCodes.Status429TooManyRequests)
    {
        var logger = context.RequestServices.GetRequiredService<ILogger<Startup>>();
        var ip = context.Connection.RemoteIpAddress?.ToString();
        var path = context.Request.Path;
        var ua = context.Request.Headers["User-Agent"].ToString();

        logger.LogWarning("Rate limit exceeded. IP: {IP}, Path: {Path}, User-Agent: {UserAgent}",
            ip, path, ua);
    }
});
```

IV. Report

We used ChatGPT-4o to review our draft content, fix grammatical errors, simplify complex sentences, and ensure that our writing was clear and professional. Moreover, ChatGPT-4o helped restructure disorganized sections of the report by providing recommendations on logical flow, heading hierarchy. For certain longer sections, we used ChatGPT-4o to generate summaries, which helped us create concise executive summaries or abstract sections. In reviewing our security-related configurations (e.g., JWT implementation, environment variable usage, password hashing), ChatGPT-4o highlighted best practices and recommended changes (e.g., using FixedTimeEquals for hash comparison or moving secrets to environment variables), which we reflected both in our implementation and our documentation.