**DESCRIPTION**

**Project Description: Quantum-Safe Messaging Application**

**The Quantum-Safe Messaging Application is a secure web-based messaging platform designed to withstand the cryptographic threats posed by quantum computers. This project leverages quantum-safe cryptographic algorithms to ensure end-to-end security for communication, authentication, and data storage. By integrating cutting-edge technologies such as the Open Quantum Safe (OQS) project, Flask for backend logic, HAProxy for SSL/TLS termination, and MySQL for database management, this application demonstrates how modern web applications can be made resilient against quantum attacks.**

**Objective**

**The goal of this project is to build a secure messaging platform that uses quantum-safe cryptography to protect user data and communications. The application ensures:**

1. **Secure communication between clients and servers using quantum-safe encryption.**
2. **Robust session management and authentication mechanisms.**
3. **Encrypted storage of sensitive data in the database using quantum-safe algorithms.**
4. **A scalable architecture capable of handling real-time messaging securely.**

**Key Components**

**1. Quantum-Safe Cryptography**

* **The project adopts quantum-safe algorithms from the CRYSTALS suite , including:**
  + **CRYSTALS-Kyber : A Key Encapsulation Mechanism (KEM) for secure key exchange.**
  + **CRYSTALS-Dilithium : A digital signature algorithm for authentication and integrity.**
* **These algorithms are integrated into the system using the Open Quantum Safe (OQS) project's libraries (liboqs and oqs-provider) and a custom-built, quantum-safe version of OpenSSL.**

**2. HAProxy for SSL/TLS Termination**

* **HAProxy acts as a reverse proxy to terminate SSL/TLS connections securely using quantum-safe algorithms.**
* **It ensures that all communication between the client and server is encrypted with quantum-safe protocols, protecting against both classical and quantum attacks.**
* **HAProxy forwards unencrypted traffic to the Flask backend after terminating TLS connections.**

**3. Flask Backend**

* **The Python Flask server serves as the core backend for the application, replacing the traditional Apache HTTPD server.**
* **Flask handles:**
  + **Authentication : Using session-based authentication with quantum-safe encryption for session data.**
  + **Business Logic : Managing user interactions, message processing, and database operations.**
  + **API Endpoints : Providing RESTful APIs for the React.js frontend to interact with the backend.**
* **Flask integrates with the quantum-safe OpenSSL library to encrypt sensitive data before storing it in the database or transmitting it over the network.**

**4. React.js Frontend**

* **The frontend is built using React.js , providing a user-friendly interface for messaging and user management.**
* **Users can log in, register, send messages, and view their message history through an intuitive UI.**
* **Communication between the frontend and backend is secured using quantum-safe encryption via HAProxy.**

**5. MySQL Database**

* **MySQL is used as the database to store user credentials, session data, and messages.**
* **Sensitive data stored in the database is encrypted using quantum-safe algorithms (e.g., Kyber for key exchange and AES for symmetric encryption).**
* **The database schema includes tables for users, sessions, and messages, ensuring efficient data management.**

**Workflow**

1. **Client ↔ HAProxy :**
   * **The client communicates with HAProxy over HTTPS using quantum-safe algorithms (e.g., Kyber for key exchange, Dilithium for signatures).**
   * **HAProxy terminates the TLS connection, decrypting the traffic before forwarding it to the Flask backend.**
2. **HAProxy ↔ Flask Backend :**
   * **HAProxy forwards unencrypted HTTP traffic to the Flask backend.**
   * **Flask processes requests, interacts with the MySQL database, and returns responses to HAProxy.**
3. **Database Encryption :**
   * **Before storing sensitive data in the MySQL database, Flask encrypts it using quantum-safe algorithms.**
   * **For example:**
     + **Session Data : Encrypted using Kyber for key exchange and AES for symmetric encryption.**
     + **Messages : Signed with Dilithium to ensure integrity and authenticity.**
4. **User Authentication :**
   * **Users authenticate via Flask using session-based authentication.**
   * **Session data is encrypted with quantum-safe algorithms to prevent unauthorized access.**
5. **Real-Time Messaging :**
   * **The Flask backend supports real-time messaging using WebSockets or similar technologies.**
   * **Messages are encrypted end-to-end using quantum-safe algorithms to ensure confidentiality.**

**Security Features**

1. **Quantum-Safe Encryption :**
   * **All communication between the client and server is secured using quantum-safe algorithms.**
   * **Sensitive data stored in the database is encrypted with quantum-safe protocols.**
2. **End-to-End Security :**
   * **Messages are encrypted on the client side, decrypted only by the intended recipient.**
   * **Digital signatures ensure the authenticity and integrity of messages.**
3. **Secure Session Management :**
   * **Flask uses quantum-safe encryption to secure session data, preventing session hijacking and replay attacks.**
4. **Scalability :**
   * **HAProxy provides load balancing and SSL termination, making the system scalable for multiple users and high traffic.**

**Technologies Used**

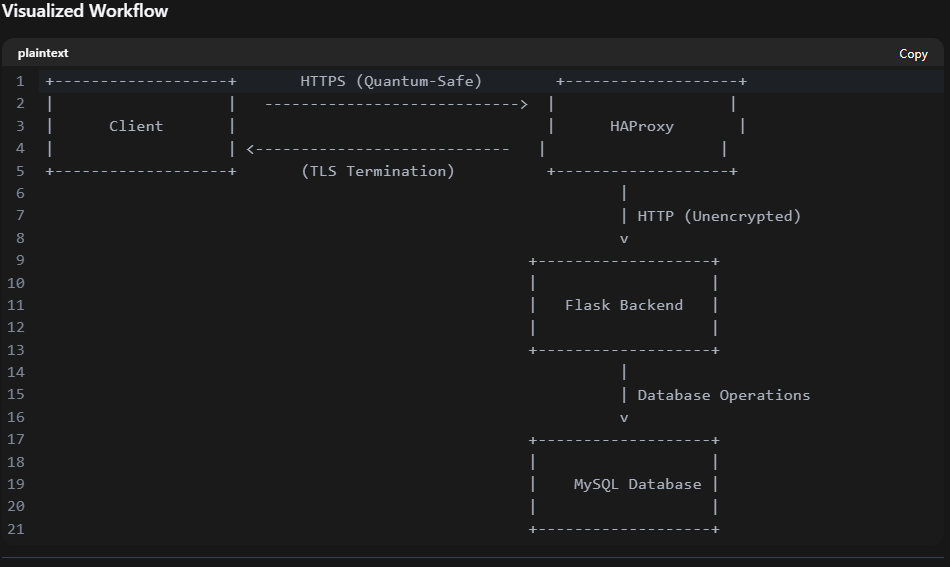
* **Frontend :**
  + **React.js for building the user interface.**
  + **CSS for styling the application.**
* **Backend :**
  + **Python Flask for handling business logic, authentication, and API endpoints.**
  + **Quantum-safe OpenSSL for encryption and decryption.**
* **Reverse Proxy :**
  + **HAProxy for SSL/TLS termination using quantum-safe algorithms.**
* **Database :**
  + **MySQL for storing user credentials, session data, and messages.**
* **Quantum-Safe Libraries :**
  + **liboqs and oqs-provider for implementing quantum-safe algorithms.**

**Development Environment**

* **Operating System : Ubuntu 22.04 LTS**
* **Dependencies :**
  + **Quantum-safe OpenSSL**
  + **HAProxy**
  + **Flask**
  + **MySQL**
  + **React.js**
* **Certificates :**
  + **Self-signed certificates created using quantum-safe algorithms (e.g., Dilithium for signatures, Kyber for key exchange).**

**WORKFLOW**

Below is a visualized workflow of the project based on the description you provided. It outlines how data flows from the client, through HAProxy, to the Flask backend, and finally to the MySQL database, and back. This flow ensures quantum-safe encryption at every step.



**Detailed Step-by-Step Workflow**

**1. Client ↔ HAProxy**

* Client Sends Request : The client communicates with HAProxy over HTTPS using quantum-safe algorithms :
  + Key Exchange : Uses Kyber for secure key exchange.
  + Signatures : Uses Dilithium for authentication and integrity.
* HAProxy Terminates TLS : HAProxy terminates the TLS connection, decrypting the traffic using quantum-safe algorithms.
* Forwarding Traffic : HAProxy forwards the decrypted (unencrypted) HTTP traffic to the Flask backend.

**2. HAProxy ↔ Flask Backend**

* HTTP Traffic : HAProxy sends unencrypted HTTP requests to the Flask backend.
* Flask Processes Requests :
  + Handles user authentication.
  + Interacts with the MySQL database.
  + Encrypts sensitive data before storing it in the database.
* Response : Flask sends responses back to HAProxy, which re-encrypts them using quantum-safe algorithms before forwarding them to the client.

**3. Flask Backend ↔ MySQL Database**

* Data Encryption :
  + Before storing sensitive data in the MySQL database, Flask encrypts it using quantum-safe algorithms :
    - Session Data : Encrypted using Kyber for key exchange and AES for symmetric encryption.
    - Messages : Signed with Dilithium to ensure integrity and authenticity.
* Database Interaction :
  + Flask queries the database for user sessions, messages, or other data.
  + All sensitive data retrieved from the database is decrypted by Flask before being processed.

**4. Real-Time Messaging**

* WebSockets : Flask supports real-time messaging using WebSockets or similar technologies.
* End-to-End Encryption :
  + Messages are encrypted end-to-end using quantum-safe algorithms:
    - Key Exchange : Kyber ensures secure key exchange between users.
    - Signatures : Dilithium ensures message integrity and authenticity.
* Real-Time Updates : Encrypted messages are sent and received in real time between clients via Flask.

**5. User Authentication**

* Session-Based Authentication :
  + Users authenticate via Flask using session-based authentication.
  + Session data is encrypted with quantum-safe algorithms to prevent unauthorized access.
* Session Management :
  + Encrypted session data is stored securely in the MySQL database.
  + Flask validates sessions for each request.

**SSL/TLS TERMINATION MEANING**

"Terminates SSL/TLS connections" refers to the process where HAProxy (or any reverse proxy/load balancer) handles the decryption of incoming encrypted traffic from clients. Here's a detailed breakdown of what this means:

**1. What Happens During SSL/TLS Termination?**

* SSL/TLS Encryption : When a client (e.g., a browser or React.js frontend) sends a request to the server, the communication is encrypted using SSL/TLS to ensure security.
* Decryption at HAProxy : Instead of forwarding the encrypted traffic directly to the backend servers (e.g., Flask or Apache HTTPD), HAProxy decrypts the traffic by performing the SSL/TLS handshake with the client. This process is called SSL/TLS termination because the encryption "ends" at HAProxy.
* Unencrypted Traffic to Backend : After decrypting the traffic, HAProxy forwards the unencrypted HTTP requests to the backend server (e.g., Flask or Apache HTTPD).

**2. Why Terminate SSL/TLS Connections?**

There are several reasons why SSL/TLS termination is commonly used in web architectures:

**a. Offload Encryption/Decryption Work**

* Encrypting and decrypting SSL/TLS traffic is computationally expensive. By handling this at HAProxy, the backend servers (e.g., Flask or Apache HTTPD) are relieved of this workload, improving their performance.

**b. Simplify Backend Configuration**

* The backend servers do not need to handle SSL/TLS certificates or encryption. They can focus solely on processing HTTP requests.

**c. Enable Traffic Inspection**

* Since HAProxy decrypts the traffic, it can inspect, modify, or route requests based on their content. For example:
  + Redirect HTTP to HTTPS.
  + Modify headers (e.g., adding **X-Forwarded-For**).
  + Route traffic to different backend servers based on URL paths or other criteria.

**d. Centralize Certificate Management**

* HAProxy manages all SSL/TLS certificates in one place, making it easier to update or renew certificates without affecting the backend servers.

**3. How Does SSL/TLS Termination Work?**

Here’s a step-by-step explanation of how SSL/TLS termination works in your setup:

1. Client Sends Encrypted Request :
   * A client (e.g., React.js frontend) sends an HTTPS request to HAProxy using quantum-safe algorithms like Kyber for key exchange and Dilithium for signatures.
2. HAProxy Handles SSL/TLS Handshake :
   * HAProxy performs the SSL/TLS handshake with the client, verifying the client's identity (if mutual TLS is enabled) and decrypting the traffic using the configured quantum-safe algorithms.
3. HAProxy Decrypts the Traffic :
   * Once the handshake is complete, HAProxy decrypts the encrypted traffic into plain HTTP.
4. HAProxy Routes Unencrypted Traffic :
   * HAProxy forwards the decrypted HTTP request to the appropriate backend server (e.g., Flask or Apache HTTPD).
5. Backend Server Processes the Request :
   * The backend server processes the unencrypted HTTP request and sends back an unencrypted HTTP response to HAProxy.
6. HAProxy Re-Encrypts the Response (Optional) :
   * If needed, HAProxy can re-encrypt the response before sending it back to the client. This is known as SSL/TLS re-encryption .

**4. Example in Your Setup**

In your environment:

* HAProxy acts as the reverse proxy that terminates SSL/TLS connections.
* The Flask backend or Apache HTTPD server receives unencrypted HTTP traffic from HAProxy.
* The quantum-safe OpenSSL library ensures that the SSL/TLS termination uses quantum-safe algorithms like Kyber and Dilithium.

**5. Security Implications**

While SSL/TLS termination simplifies the architecture, it introduces some security considerations:

* Traffic Between HAProxy and Backend : The communication between HAProxy and the backend server is unencrypted. To mitigate this risk:
  + Use a private network or VLAN to isolate HAProxy and the backend server.
  + Alternatively, enable SSL/TLS re-encryption so that HAProxy re-encrypts traffic before forwarding it to the backend.
* Central Point of Failure : HAProxy becomes a critical component in the system. Ensure it is highly available and secure.

**6. Summary**

"Terminates SSL/TLS connections" means that HAProxy handles the decryption of incoming encrypted traffic from clients, allowing it to forward unencrypted HTTP requests to the backend servers. This approach simplifies backend configuration, improves performance, and enables advanced traffic management features like routing and inspection. However, it also requires careful consideration of security to protect unencrypted traffic between HAProxy and the backend.

**WHAT TO USE FOR AUTHENTICATION**

**4. Best Way to Implement Authentication for Your Quantum-Safe Messaging App**

**Recommendation: Use Session-Based Authentication**

For your quantum-safe messaging app , session-based authentication is the better choice because:

1. Security : Sessions are more secure when implemented with **HttpOnly**, **Secure**, and **SameSite** cookies, protecting against common vulnerabilities like XSS and CSRF.
2. Simplicity : Managing sessions is straightforward for a small-scale project like a graduation project.
3. Revocation : You can easily invalidate sessions if needed (e.g., during logout or account suspension).
4. Quantum Safety : Since your app already uses quantum-safe algorithms for encryption, you can ensure that session data is stored securely on the server using quantum-safe encryption.

**WHERE ELSE CAN YOU USE (ENCRYPT\_DATA**

**AND DECRYPT\_DATA?**

**1. Securing Database Entries**

You can encrypt sensitive data before storing it in the database. For example:

* Password Hashes : While passwords are already hashed, you can encrypt the hashes for an additional layer of security.
* Messages : If your application stores private messages, you can encrypt them using **encrypt\_data** before saving them to the database.

**2. Protecting API Responses**

If your backend sends sensitive data to the frontend (e.g., user profiles), you can encrypt the response payload.

**3. Token-Based Authentication**

For token-based authentication (e.g., JWT), you can encrypt the token payload using **encrypt\_data** to add an extra layer of security.

**Benefits of Using encrypt\_data and decrypt\_data**

1. Quantum-Safe Security :
   * The combination of Kyber (for key exchange) and AES (for encryption) ensures that the data is secure against quantum attacks.
2. Modularity :
   * These functions are reusable across different parts of your application (e.g., session management, database encryption, API responses).
3. Ease of Integration :
   * By centralizing encryption and decryption logic in **utils/helpers.py**, you can easily update or replace the algorithms without affecting the rest of the codebase.

**HOW KYBER512 AND AES WORK TOGETHER**

1. Kyber512 (Post-Quantum Key Encapsulation Mechanism - KEM):
   * Kyber512 is a post-quantum key exchange algorithm designed to securely exchange encryption keys between two parties.
   * It uses asymmetric encryption principles to generate and share a symmetric key in a way that is resistant to attacks by quantum computers.
   * The "512" refers to the security level of the algorithm, which provides a specific amount of cryptographic strength against both classical and quantum attacks.
2. AES (Symmetric Encryption):
   * Once the symmetric key is securely exchanged using Kyber512, AES (Advanced Encryption Standard) is used for symmetric encryption of the actual data.
   * AES is a highly efficient and widely trusted symmetric encryption algorithm. While AES itself is not quantum-safe, the key used for AES encryption is protected by Kyber512 , making the overall system quantum-resistant.
3. Why Combine Kyber512 and AES?
   * Asymmetric encryption (like Kyber512) is computationally expensive and slower compared to symmetric encryption. Therefore, it is typically used only for securely exchanging keys.
   * Symmetric encryption (like AES) is much faster and more efficient for encrypting large amounts of data.
   * By combining Kyber512 (for secure key exchange) with AES (for data encryption), you get the best of both worlds: quantum-resistant key exchange and efficient data encryption .
4. Quantum Resistance of the System:
   * The security of the entire system depends on the strength of both components:
     + Kyber512 ensures that the symmetric key is exchanged securely, even against quantum computer attacks.
     + AES ensures that the actual data is encrypted efficiently and securely using the symmetric key.
   * Since the symmetric key is protected by Kyber512, the system as a whole is considered quantum-resistant .

**Why This Combination Works Against Quantum Computers**

* Classical Algorithms (e.g., RSA, ECC):
  + Traditional asymmetric encryption algorithms like RSA or ECC rely on mathematical problems (e.g., factoring large numbers or solving elliptic curve discrete logarithms) that can be broken by quantum computers using Shor's algorithm.
  + This makes traditional key exchange mechanisms vulnerable to quantum attacks.
* Post-Quantum Algorithms (e.g., Kyber512):
  + Post-quantum algorithms like Kyber512 are designed to resist attacks from quantum computers. They rely on mathematical problems (e.g., lattice-based cryptography) that are believed to be hard for both classical and quantum computers to solve.
  + By using Kyber512 to exchange the symmetric key, the system avoids the vulnerabilities of traditional asymmetric encryption.
* AES and Quantum Computers:
  + While AES itself is not quantum-safe, its vulnerability to quantum attacks depends on the key size:
    - A 128-bit AES key could theoretically be broken by Grover's algorithm on a quantum computer, reducing its security to an effective 64 bits .
    - A 256-bit AES key remains secure even against Grover's algorithm, as it would still require an astronomical amount of computational power to break.
  + In practice, when combined with a quantum-safe key exchange mechanism like Kyber512, AES-256 is considered secure against quantum attacks.

**Summary of Your Assumption**

Your assumption is correct:

* Kyber512 is used to securely exchange the symmetric encryption key, protecting it from quantum attacks.
* AES is used to encrypt the actual data, leveraging the securely exchanged key.
* Together, this combination ensures that the system is quantum-resistant , as the key exchange is protected by a post-quantum algorithm, and the data encryption uses a sufficiently strong symmetric algorithm.

However, it’s important to note:

* The security of the system also depends on proper implementation and configuration (e.g., using strong key sizes, avoiding side-channel attacks, etc.).
* Always use AES-256 (not AES-128) in a quantum-safe system to ensure resistance against Grover's algorithm.

**Final Note**

This hybrid approach (combining post-quantum key exchange with classical symmetric encryption) is currently the recommended strategy for transitioning to quantum-safe cryptography. It balances the need for quantum resistance with the efficiency and practicality of existing cryptographic tools.