

Post-Quantum Secure Messaging Web Application

Group Members

Jannat Butt 21I-0259
Misha Fakhar 21I-2718

GitHub Repository:

https://github.com/Jannat-Butt/Infosecurity_projects

PythonAnywhere Deployment:

<https://imjannatbutt.pythonanywhere.com/>

April 30, 2025

Contents

1	Introduction	2
2	System Architecture	2
3	Key Generation	2
4	Encryption Process	3
4.1	1. Key Encapsulation (Kyber768)	3
4.2	2. Symmetric Encryption (AES-GCM)	3
5	Decryption Process	4
5.1	1. Symmetric Decryption (AES-GCM)	4
5.2	2. Message Retrieval	4
6	Kyber768: The PQC Algorithm	5
6.1	Algorithm Overview	5
6.2	Security Parameters	5
7	AES-GCM: Symmetric Encryption	5
8	Technology Stack	6
9	Sample Code Snippets	6
9.1	AES Encryption Function	6
9.2	Kyber Key Generation	6
10	Conclusion	6

1 Introduction

With the rise of quantum computing, classical encryption algorithms are vulnerable to quantum attacks. This project demonstrates a secure web application that uses **post-quantum cryptography (PQC)** techniques to ensure encrypted communication remains secure even in the presence of quantum adversaries.

The web application uses **CRYSTALS-Kyber768**, a lattice-based Key Encapsulation Mechanism (KEM) standardized by NIST, in combination with **AES-GCM** for message confidentiality.

2 System Architecture

The secure messaging system is built using the Flask web framework and consists of the following components:

- Kyber768 KEM for key exchange.
- AES-GCM symmetric encryption for message confidentiality.
- HTML/CSS frontend for user interaction.
- Flask server logic to manage encryption and decryption.

3 Key Generation

When a user clicks **Generate Keys**, the system uses the Kyber768 algorithm to generate:

- A public key (used for encryption)
- A private key (used for decryption)

These keys are encoded in Base64 and displayed on the webpage.

Post-Quantum Secure Messaging

Generate Public & Private Keys

Your Keys

Public Key:

```
HqZ4xzWKEQPHCRc0A1IBsTH4ldStnIPrpagO/QttJiKMCWxkBaXDwtm0XoUxzG8kAMugRDGAfEqX9ypdjG5GKdiVLqjagpmXFSzu9EFSQo8e
KGL/IW4srsITfWeEEF/1ASWUWQx6Lkrr8YZcPhU/bjNouBpV/uXoSyF0wRjeCIG38Rcmxp1Ptq5RcUQgxS0IyewE4CV0b010n1k-jYe++ELLfb
YQ5kOK5RyuHYgwmCRiLMXh0P26aU2FFd/vE1GEepEjszu4ZqV5F+RIy6hyi4A1mqBoKWQgcGaQCY8yZoz+PDvIzCOPhB78CmXPJaHEYVPVx
rpEVTx+SYT2JbVDCkPbaSEIQKuympX5MrhxEPK1HbmrOy9WpdND0c1gAmwNNzy1KkMiFDW9aXdxS6hCC7eyVimc+XRkXE2GZhCFov1eL7PYT
q9RJKLbE01cUJeinBWFvTtQN8ZgBPEVCi2NbGLSfYt5QeyS1cthl1630hIdH3RZwZTAc9cwaT8m9HrK4bHDMszJ3rEsjX+qdcEERS8WUpdca4
/atZggBuusymSD0tXQnrXNtWGagjg++NXG1/Mm6Li00seYhEsoUFAxC1x5MjGQ145XrDvirXYF7IqnJdit98YcvAEvciVVo/ucQ/E3TJwIG6
cYREuUipWJDuu6R/havJjN3VxA/GpCHNi5ImQmS0GL4Lk63RBQrV5yCwU7YyJpDhxutTx9NhQ9jaQqBrSknFAFuhYj71qqgasF+EJ71pk1Sow
```

Private Key:

```
3WTi9rtPPuK694hR3rU320JLpeZFMRCN+epdeQFW2ypF0eV8QToo3aBhUvqW0E0GBsMwDOPcUvYBQmXesaQKq+aHLXBSy+zRxcM1uNnoVH10SOp
dQ0uWeMvewa1ZFWMMPCbcLBfwwLpUF/W5PHPvuuk/SVJTEKt0EnF71kyETNwkK23CDA0HQiL3y3b91dKSH0AbK21qc9pMajwqaBNGNvTmCXEA
zL9NePsYsM2PBjcdF1ZSGciqBaWFGcEKoQqJfU2YqZHXf8/TP0smYGCPDbTtnYc163FU60tFD8UOnIgdY+Qs1SKVEEOM8Ej1DHFp1b0M7WIo
rW7FZ4ZLICtxqE5FZfdgn7ouuUFCgkopAroK57yWmDEZ36pxJPuMIjBBnnOthvFCRERkkJJKOIfM4yHoRNSHT4Gm9cyd4t1Z1kUgrLpsoHRav
fGpzVTmM4ckkerCijJkOb7gmJfxeQzK1/CEQv30ftvGs0NxtwyxjwgyCeKWFZomNvEc8PUfN7ipT8QvOQ+UMwAPNg3UgyZSDRypL2FAzFKN8P
aU4wwITJJBkCGKgzBhs9QVoC7K1Yfi7y4Iz5Q522jbDx9yocdtUCHNxnogrwZz5eaiEhVNU3FwAVwG79dXf1Mrh8hSR1nB3MAUwFuYC/IhsR
mWigmxfgdgZeEuG2IdRF9YpFyqmCP5KUsFeFdNuasIir4vgvw/N7oxJQj4m6FGUYX0chv21jh4sB+ac721xb88ZQ3Wyc9WqBiQw1vfVL2Ic
```

Figure 1: Key Generation

4 Encryption Process

The encryption process consists of two stages:

4.1 1. Key Encapsulation (Kyber768)

The sender uses the receiver's public key to perform the `encap()` operation, which yields:

- `ciphertext_kem`: Encapsulated shared secret
- `shared_secret`: A 32-byte secret key derived during encapsulation

4.2 2. Symmetric Encryption (AES-GCM)

The shared secret is used to derive a 256-bit AES key (first 32 bytes). Then, the plaintext message is encrypted using AES-GCM:

- A random 96-bit IV (initialization vector) is generated
- GCM mode ensures both confidentiality and integrity
- The resulting token includes IV, tag, and ciphertext

Encrypt a Message

i am jannat

Encrypt Message

Encrypted Output

Ciphertext (Kyber Encrypted Shared Key):

aB0VOZfhLuGZZVXW/WUkPAXUnng4aih2zQa1h4hpGPc=

Encrypted Message (AES Encrypted):

L8eQKAK1bkVHyBGTMQRSFvPKXRkimTTbnGfdeyRUoJlyY8duesbmA

Figure 2: Message Encryption

5 Decryption Process

The decryption also has two stages:

5.1 1. Symmetric Decryption (AES-GCM)

Instead of recalculating the shared secret, the server reuses the `shared_secret` from the previous encryption step. It extracts the AES key and decrypts the message using AES-GCM.

Note: In this implementation, the shared secret derived during encryption is reused during decryption for simplicity. In a production environment, the correct approach is to use the `decap()` function with the private key and `ciphertextkem` to re-derive the shared secret securely. This enhances security and aligns with best cryptographic practices.

5.2 2. Message Retrieval

The decrypted message is then displayed in a readable format.

Decrypted Message

A screenshot of a terminal window with a white background and a thin black border. The title bar at the top reads "Decrypted Message". Inside the terminal, the text "i am jannat" is displayed in a monospaced font. The text is positioned on the left side of the terminal, with a small cursor visible at the end of the line.

Figure 3: Message Decryption

6 Kyber768: The PQC Algorithm

Kyber768 is a quantum-secure algorithm based on the **Module-LWE** problem, which remains hard even for quantum computers.

6.1 Algorithm Overview

- Based on structured lattice problems
- Uses polynomial arithmetic over rings
- Offers strong security and high efficiency
- Selected by NIST for post-quantum standardization

6.2 Security Parameters

- Security level: **128-bit post-quantum security**
- Key size (public): 1.1KB
- Ciphertext size: 1KB
- Shared secret: 32 bytes

7 AES-GCM: Symmetric Encryption

AES-GCM (Advanced Encryption Standard in Galois/Counter Mode) is used for encrypting messages with the shared key.

- Combines encryption and authentication
- Fast and secure for real-time applications
- Requires IV (nonce), tag, and ciphertext

8 Technology Stack

- **Frontend:** HTML, CSS
- **Backend:** Python, Flask
- **Crypto Libraries:**
 - pypqc for Kyber768
 - cryptography for AES-GCM
- **Security Middleware:** Flask-Talisman
- **Environment Management:** python-dotenv

9 Sample Code Snippets

9.1 AES Encryption Function

```
def aes_encrypt(key, plaintext):  
    iv = secrets.token_bytes(12)  
    encryptor = Cipher(  
        algorithms.AES(key),  
        modes.GCM(iv),  
        backend=default_backend()  
    ).encryptor()  
    ciphertext = encryptor.update(plaintext.encode()) + encryptor.  
        finalize()  
    return base64.b64encode(iv + encryptor.tag + ciphertext).decode('utf-8')
```

Listing 1: AES-GCM Encryption

9.2 Kyber Key Generation

```
public_key, private_key = kyber.keypair()  
public_key_b64 = base64.b64encode(public_key).decode('utf-8')  
private_key_b64 = base64.b64encode(private_key).decode('utf-8')
```

Listing 2: Generate Kyber Keys

10 Conclusion

This project effectively demonstrates the integration of post-quantum cryptography into a real-world web application. By combining Kyber768 KEM with AES-GCM, we provide both forward secrecy and authenticated encryption, ensuring robust communication security even in a post-quantum world.