|  |
| --- |
| Home Page - Galala University - GU |

|  |
| --- |
| **CSE241:** Cryptography |
| **Instructor:** Dr/ Ibrahim Gomaa |

**EncrypTalk**



Where Your Words Stay Secret and Secure.

## **Team Members**

## 

|  |  |
| --- | --- |
| Marwan Tamer | 221100498 |
| Youssef Gamal | 221101013 |
| Yassin Waleed | 221100979 |

**TABLE OF CONTENTS**

Acknowledgement------------------------------------------------------ 3

Abstract ------------------------------------------------------------------ 4

Introduction ------------------------------------------------------------- 4

Theoretical Part

Project Scenario

Results and analysis ---------------------- 12

Conclusion ------------------------ 101

References --------------------------------------------------------------- 108

**Acknowledgement**

I would like to acknowledge the importance of addressing cybersecurity threats, including the risks associated with malvertising and dive-by downloads. This topic serves as a reminder of the critical need for digital vigilance and security measures in our increasingly interconnected world. By studying and understanding these malicious tactics, we strive to contribute to the ongoing efforts in raising awareness, implementing preventive measures, and fostering a safer digital environment for individuals and organizations alike.

**Report: Libraries Usage in Real-Time Chat Application**

**1. PyQt5**

PyQt5 serves as the primary framework for building the graphical user interface (GUI) of the real-time chat application. It provides a comprehensive set of tools and modules for designing interactive desktop applications with rich graphical elements and responsive user interfaces.

* **Modules Used**:
  + **PyQt5.QtCore**: Core functionalities such as event handling, timers, and signals/slots. This module enables the application to respond to user interactions and events, ensuring smooth and interactive behavior.
  + **PyQt5.QtGui**: Tools for graphical operations, including drawing shapes, handling colors, managing fonts, and processing input events like mouse clicks and keyboard inputs. This module enhances the visual appeal and interactivity of the application.
  + **PyQt5.QtWidgets**: Components for building GUI elements such as buttons, text fields, labels, layouts, and containers. These widgets provide the building blocks for creating intuitive user interfaces and organizing content effectively.

|  |  |
| --- | --- |
| Extensively used for GUI design, event handling, and managing user interactions. PyQt5 classes and modules are utilized to create various GUI components, handle user inputs, and organize the layout of the application's interface effectively. |  |

**2. Requests**

The requests library is employed for making HTTP requests to the server, enabling communication between the client-side application and the backend server. It facilitates tasks such as user authentication, data retrieval, and interaction with the server's API endpoints.

* **Modules Used**:
  + **requests**: The main module for sending HTTP requests with various methods like GET, POST, PUT, DELETE, etc. This module is used to interact with the server's RESTful API endpoints by sending JSON payloads and handling responses.
  + **requests.exceptions**: Contains exception classes for handling errors that may occur during network operations, such as connection timeouts, DNS resolution failures, and HTTP errors. Exception handling ensures robust error management and graceful recovery from network-related issues.
  + **requests.RequestException**: Base class for all exceptions in the requests module. It serves as a catch-all for network-related exceptions during HTTP requests, providing a unified interface for handling errors in the application.

|  |  |
| --- | --- |
| Employs HTTP requests to communicate with the server for tasks such as user authentication, data retrieval, and interaction with the server's API endpoints. Exception handling ensures robust error management and graceful recovery from network-related issues. |  |

**3. Cryptography**

The cryptography library is utilized for implementing encryption and decryption of messages, ensuring the security and confidentiality of communication between clients and the server. It provides cryptographic primitives and algorithms for secure data transmission.

* **Modules Used**:
  + **cryptography.hazmat.primitives.ciphers**: Contains implementations of cryptographic cipher algorithms such as AES (Advanced Encryption Standard). This module is used for encrypting and decrypting messages using symmetric-key encryption.
  + **cryptography.hazmat.primitives.kdf.pbkdf2**: Provides the PBKDF2-HMAC key derivation function for deriving cryptographic keys from passwords. Key derivation ensures secure handling of passwords and strengthens the security of encryption.
  + **cryptography.hazmat.primitives.hashes**: Implements cryptographic hash functions such as SHA-256 for hashing passwords and generating random salts. Hashing is used to securely store passwords and generate unique identifiers for cryptographic operations.
  + **cryptography.hazmat.backends.default\_backend**: Provides access to the default cryptographic backend for the system. It abstracts away the underlying cryptographic implementation details and ensures compatibility across different platforms and environments.

|  |  |
| --- | --- |
| Implements encryption and decryption of messages using AES encryption with CFB mode, ensuring the security and confidentiality of communication. Key derivation strengthens the security of encryption by deriving cryptographic keys from passwords securely. | Automated cryptocode generator is helping secure the web | MIT News |  Massachusetts Institute of Technology |

**4. Socketio**

socketio enables real-time bidirectional communication between clients and the server using WebSockets, facilitating instant messaging and updates in the chat application. It provides a reliable and efficient communication mechanism for delivering messages in real-time.

* **Modules Used**:
  + **socketio.Client**: The main class for creating WebSocket clients, allowing bidirectional communication with the server. This module establishes and manages WebSocket connections, sends and receives messages, and handles various events such as connection status changes and incoming messages.
  + **socketio.exceptions**: Contains exception classes for handling errors specific to the socketio library. Exception handling ensures robust error management during socket-related operations, such as connection failures or server-side errors.

|  |  |
| --- | --- |
| Facilitates real-time communication with the server using WebSockets, enabling instant messaging and updates in the chat application. Exception handling provides robust error management for socket-related operations, ensuring reliable communication with the server. | What's new in SocketIO 4? |

**5. Base64**

The base64 module is utilized for encoding and decoding binary data in a URL-safe format. It ensures the safe transmission of encrypted messages over network protocols by encoding binary data in a format that is compatible with URL encoding and decoding.

* **Modules Used**:
  + **base64.urlsafe\_b64encode**, **base64.urlsafe\_b64decode**: Functions for encoding and decoding binary data in a URL-safe manner. These functions are used to encode encrypted messages before transmission and decode received messages for decryption, ensuring data integrity and compatibility with network protocols.

Ensures safe transmission of encrypted messages over network protocols by encoding binary data in a URL-safe format. Encoding and decoding functions ensure data integrity and compatibility with network protocols, preventing data corruption during transmission.

**Constants and Utility Functions**

base\_url = "https://real-time-chat-api-v1.onrender.com"

SECRET\_KEY = b"\xdf\xe8\xebD\xe8\x81\x00\xce\xf8C\x038i\xec\x1c{\xa8\xda}\xf1\x18\xd9)p\xc14E(\x1d\xa5v\xe9"  # 16 bytes for AES-128

def derive\_key(password: bytes, salt: bytes) -> bytes:

    kdf = PBKDF2HMAC(

        algorithm=hashes.SHA256(),

        length=32,

        salt=salt,

        iterations=100000,

        backend=default\_backend(),

    )

    return kdf.derive(password)

* **base\_url**: The base URL for the chat application's API.
* **SECRET\_KEY**: A 32-byte key used for encryption (AES-256).
* **derive\_key**: Derives a 256-bit encryption key from a password and salt using PBKDF2-HMAC-SHA256.

def encrypt\_message(message: str, key: bytes) -> str:

    salt = os.urandom(16)

    derived\_key = derive\_key(key, salt)

    iv = os.urandom(16)

    cipher = Cipher(

        algorithms.AES(derived\_key), modes.CFB(iv), backend=default\_backend()

    )

    encryptor = cipher.encryptor()

    encrypted\_message = encryptor.update(message.encode()) + encryptor.finalize()

    return urlsafe\_b64encode(salt + iv + encrypted\_message).decode()

**encrypt\_message**: Encrypts a message using AES-256-CFB mode.

* + **salt**: Random 16-byte salt for key derivation.
  + **derived\_key**: Derived key using PBKDF2.
  + **iv**: Random 16-byte initialization vector for AES.
  + **cipher**: AES cipher in CFB mode.
  + **encryptor**: Encryptor object to perform the encryption.
  + **encrypted\_message**: The encrypted message, base64 encoded with the salt and iv.

def decrypt\_message(encrypted\_message: str, key: bytes) -> str:

    encrypted\_message = urlsafe\_b64decode(encrypted\_message)

    salt = encrypted\_message[:16]

    iv = encrypted\_message[16:32]

    encrypted\_data = encrypted\_message[32:]

    derived\_key = derive\_key(key, salt)

    cipher = Cipher(

        algorithms.AES(derived\_key), modes.CFB(iv), backend=default\_backend()

    )

    decryptor = cipher.decryptor()

    decrypted\_message = decryptor.update(encrypted\_data) + decryptor.finalize()

    return decrypted\_message.decode()

**decrypt\_message**: Decrypts an encrypted message.

* + **encrypted\_message**: Base64 decoded message.
  + **salt**: Extracted salt from the encrypted message.
  + **iv**: Extracted initialization vector.
  + **encrypted\_data**: Extracted encrypted data.
  + **derived\_key**: Derived key using PBKDF2.
  + **cipher**: AES cipher in CFB mode.
  + **decryptor**: Decryptor object to perform the decryption.
  + **decrypted\_message**: The decrypted message.

**Main Application Class**

class MainApp(QMainWindow, ui):

    def \_\_init\_\_(self):

        QMainWindow.\_\_init\_\_(self)

        self.setupUi(self)

        self.UI\_Changes()

        self.Handle\_buttons()

        self.access\_token = ""

        self.username = ""

        self.sio = socketio.Client()

        self.sio.on("connect", self.on\_connect)

        self.sio.on("disconnect", self.on\_disconnect)

        self.sio.on("user\_online", self.on\_new\_message)

        self.sio.on("user\_ofline", self.on\_new\_message)

        self.sio.on("error", self.on\_error)

        self.sio.on("new\_message", self.on\_enc\_new\_message)

    def UI\_Changes(self):

        self.tabWidget.tabBar().setVisible(False)

        self.tabWidget.setCurrentIndex(0)

    def Handle\_buttons(self):

        self.toolButton\_11.clicked.connect(lambda: self.tabWidget.setCurrentIndex(1))

        self.toolButton\_12.clicked.connect(lambda: self.tabWidget.setCurrentIndex(0))

        self.toolButton\_13.clicked.connect(lambda: self.tabWidget.setCurrentIndex(3))

        self.toolButton\_10.clicked.connect(lambda: self.tabWidget.setCurrentIndex(2))

        self.login\_btn.clicked.connect(self.login)

        self.signup\_btn.clicked.connect(self.signup)

        self.login\_toolButton\_3.clicked.connect(self.update\_password)

        self.toolButton\_9.clicked.connect(self.send\_message)

**MainApp**: The main class for the application, inheriting from **QMainWindow** and **Ui\_MainWindow**.

* **init**: Initializes the application, sets up the UI, handles button clicks, and initializes the Socket.IO client.
  + **self.setupUi(self)**: Sets up the UI from the loaded .ui file.
  + **self.UI\_Changes()**: Customizes the initial state of the UI.
  + **self.Handle\_buttons()**: Connects buttons to their respective functions.
  + **self.sio**: Initializes the Socket.IO client and defines event handlers.

**UI\_Changes**: Customizes the UI by hiding the tab bar and setting the initial tab index.

**Handle\_buttons**: Connects button clicks to their respective functions using **lambda** to pass parameters where needed.

* + **self.toolButton\_11**: Switches to tab 1.
  + **self.toolButton\_12**: Switches to tab 0.
  + **self.toolButton\_13**: Switches to tab 3.
  + **self.toolButton\_10**: Switches to tab 2.
  + **self.login\_btn**: Calls the **login** function.
  + **self.signup\_btn**: Calls the **signup** function.
  + **self.login\_toolButton\_3**: Calls the **update\_password** function.
  + **self.toolButton\_9**: Calls the **send\_message** function.

**Socket.IO Event Handlers**

 def on\_connect(self):

        print("Connected to Socket.IO server.")

    def on\_disconnect(self):

        print("Disconnected from Socket.IO server.")

    def on\_error(self, error):

        QMessageBox.information(self, "Error", error)

    def on\_new\_message(self, message):

        self.textEdit\_5.insertPlainText(message + "\n")

    def on\_enc\_new\_message(self, message):

        decrypted\_message = decrypt\_message(message, SECRET\_KEY)

        self.textEdit\_5.insertPlainText(decrypted\_message + "\n")

    def connect\_to\_server(self, token):

        server\_url = base\_url

        self.sio.connect(

            server\_url,

            auth={"token": token},

            transports=["websocket"],

        )

        self.textEdit\_5.insertPlainText(self.username + " join chat" + "\n")

**on\_connect**: Prints a message when connected to the Socket.IO server.

**on\_disconnect**: Prints a message when disconnected from the Socket.IO server.

**on\_error**: Displays an error message using a QMessageBox.

**on\_new\_message**: Appends a new message to the text editor.

**on\_enc\_new\_message**: Decrypts an encrypted message and appends it to the text editor.

**connect\_to\_server**: Connects to the server using the provided token.

* + **auth**: Sends the token for authentication.
  + **transports**: Specifies the use of WebSocket for transport.
  + **self.textEdit\_5.insertPlainText**: Displays a join message in the chat.

**Authentication Functions**

 def login(self):

        print("Login")

        username = self.lineEdit\_3.text()

        password = self.lineEdit\_4.text()

        json\_body = {

            "username": username,

            "password": password,

        }

        api\_url = f"{base\_url}/api/v1/auth/login"

        try:

            response = requests.post(api\_url, json=json\_body)

            if response.status\_code == 200:

                response\_json = response.json()

                user\_data = response\_json.get("data", {})

                access\_token = user\_data.get("accessToken")

                username = user\_data.get("username")

                self.access\_token = access\_token

                self.username = username

                self.connect\_to\_server(access\_token)

                self.tabWidget.setCurrentIndex(3)

                QMessageBox.information(

                    self, "Login Successful", "Login was successful."

                )

            else:

                error\_message = (

                    response.json().get("error", {}).get("message", "Unknown error")

                )

                QMessageBox.information(self, "Login Error", error\_message)

        except requests.RequestException as e:

            QMessageBox.information(self, "Network Error", str(e))

**login**: Handles user login.

* + **json\_body**: Contains the username and password.
  + **api\_url**: URL for the login endpoint.
  + **response**: Sends a POST request to the API.
  + **response\_json**: Parses the response JSON.
  + **user\_data**: Extracts user data from the response.
  + **access\_token**: Extracts the access token.
  + **self.connect\_to\_server**: Connects to the server with the token.
  + **QMessageBox.information**: Displays success or error messages.

 def signup(self):

        print("signup")

        username = self.lineEdit\_5.text()

        password = self.lineEdit\_6.text()

        json\_body = {

            "username": username,

            "password": password,

        }

        api\_url = f"{base\_url}/api/v1/auth/signup"

        try:

            response = requests.post(api\_url, json=json\_body)

            if response.status\_code == 200:

                response\_json = response.json()

                user\_data = response\_json.get("data", {})

                access\_token = user\_data.get("accessToken")

                username = user\_data.get("username")

                self.access\_token = access\_token

                self.username = username

                self.connect\_to\_server(access\_token)

                self.tabWidget.setCurrentIndex(3)

                QMessageBox.information(

                    self, "Signup Successful", "Signup was successful."

                )

            else:

                error\_message = (

                    response.json().get("error", {}).get("message", "Unknown error")

                )

                QMessageBox.information(self, "Signup Error", error\_message)

        except requests.RequestException as e:

            QMessageBox.information(self, "Network Error", str(e))

**signup**: Handles user signup, similar to the login function but for registration.

def update\_password(self):

        print("Update Password")

        oldPass = self.lineEdit\_7.text()

        newPassword = self.lineEdit\_9.text()

        confNewPass = self.lineEdit\_8.text()

        json\_body = {

            "oldPassword": oldPass,

            "newPassword": newPassword,

            "confNewPassword": confNewPass,

        }

        headers = {

            "Authorization": f"Bearer {self.access\_token}",

            "Content-Type": "application/json",

        }

        api\_url = f"{base\_url}/api/v1/auth/update-password"

        try:

            response = requests.post(api\_url, json=json\_body, headers=headers)

            if response.status\_code == 200:

                response\_json = response.json()

                QMessageBox.information(

                    self, "Updated Successful", "Password was updated successful."

                )

            else:

                error\_message = (

                    response.json().get("error", {}).get("message", "Unknown error")

                )

                QMessageBox.information(self, "Login Error", error\_message)

        except requests.RequestException as e:

            QMessageBox.information(self, "Network Error", str(e))

* **update\_password**: Handles password updates.
  + **headers**: Includes the authorization token.
  + **api\_url**: URL for the update-password endpoint.
  + **response**: Sends a POST request with the new password details.
  + **QMessageBox.information**: Displays success or error messages.

**Message Sending Function**

def send\_message(self):

        message = f"{self.username}: {str(self.textEdit\_4.toPlainText())}"

        encrypted\_message = encrypt\_message(message, SECRET\_KEY)

        self.sio.emit("new\_message", encrypted\_message)

        self.textEdit\_5.insertPlainText(

            f"me: {str(self.textEdit\_4.toPlainText())}" + "\n"

        )

        self.textEdit\_4.setPlainText("")

**send\_message**: Sends a message.

* + **message**: Formats the message with the username.
  + **encrypted\_message**: Encrypts the message.
  + **self.sio.emit**: Sends the encrypted message via Socket.IO.
  + **self.textEdit\_5.insertPlainText**: Displays the sent message.
  + **self.textEdit\_4.setPlainText**: Clears the input field.

**Main Function**

def main():

    app = QApplication(sys.argv)

    window = MainApp()

    window.show()

    app.exec()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**main**: Entry point of the application.

* + **app**: Creates the application.
  + **window**: Initializes the main window.
  + **window.show()**: Displays the main window.
  + **app.exec()**: Starts the application's event loop.

The code demonstrates a clear and structured approach to developing a secure real-time chat application, handling user authentication, message encryption, and real-time communication effectively.

Server

Methodology

**Overview of the Code**

The provided code uses AES (Advanced Encryption Standard) in CFB (Cipher Feedback) mode to encrypt and decrypt messages. It also uses PBKDF2 (Password-Based Key Derivation Function 2) with HMAC-SHA256 to derive encryption keys from a password.

1. **SECRET\_KEY**: A byte string that appears to be intended for use as a key or part of the key derivation process.
2. **Key Derivation Function (KDF)**:
   * **PBKDF2HMAC**: This is used to derive a secure key from a password and a salt. It uses HMAC (Hash-based Message Authentication Code) with SHA-256 as the hash function.
   * **Salt**: A random value added to the password before hashing to ensure that the same password does not always produce the same derived key, enhancing security against dictionary attacks.
3. **Encryption**:
   * **Salt**: Randomly generated for each encryption session.
   * **Initialization Vector (IV)**: Also randomly generated for each encryption session. It ensures that the same plaintext encrypted with the same key will produce different ciphertexts.
   * **AES in CFB Mode**: Encrypts the plaintext in a way that ensures even small changes in the plaintext will result in significant changes in the ciphertext.
4. **Decryption**:
   * **Extract Salt and IV**: The salt and IV are extracted from the beginning of the encrypted message.
   * **Derived Key**: The same PBKDF2 function is used to regenerate the key from the password and extracted salt.
   * **AES in CFB Mode**: Used to decrypt the ciphertext back to the original plaintext.

**Step-by-Step Explanation**

**Key Derivation**

The key derivation function transforms a password into a cryptographic key:

python

Copy code

def derive\_key(password: bytes, salt: bytes) -> bytes: kdf = PBKDF2HMAC( algorithm=hashes.SHA256(), length=32, salt=salt, iterations=100000, backend=default\_backend(), ) return kdf.derive(password)

* **PBKDF2HMAC**: Uses HMAC with SHA-256 to create a 256-bit key (32 bytes) from the password and salt.
* **Iterations**: Repeatedly applies the hashing process to slow down brute-force attacks.

**Encryption**

The encryption process includes generating random values for salt and IV, deriving the key, and encrypting the message:

python

Copy code

def encrypt\_message(message: str, key: bytes) -> str: salt = os.urandom(16) derived\_key = derive\_key(key, salt) iv = os.urandom(16) cipher = Cipher( algorithms.AES(derived\_key), modes.CFB(iv), backend=default\_backend() ) encryptor = cipher.encryptor() encrypted\_message = encryptor.update(message.encode()) + encryptor.finalize() return urlsafe\_b64encode(salt + iv + encrypted\_message).decode()

* **Salt and IV**: Both are 16 bytes long, generated using **os.urandom**.
* **Cipher**: Uses AES algorithm with the derived key and CFB mode.
* **Encryptor**: Handles the encryption, combining the encoded message with the final encrypted data.
* **Base64 Encoding**: Encodes the salt, IV, and encrypted message into a string for safe transmission.

**Decryption**

The decryption process involves reversing the steps of encryption:

python

Copy code

def decrypt\_message(encrypted\_message: str, key: bytes) -> str: encrypted\_message = urlsafe\_b64decode(encrypted\_message) salt = encrypted\_message[:16] iv = encrypted\_message[16:32] encrypted\_data = encrypted\_message[32:] derived\_key = derive\_key(key, salt) cipher = Cipher( algorithms.AES(derived\_key), modes.CFB(iv), backend=default\_backend() ) decryptor = cipher.decryptor() decrypted\_message = decryptor.update(encrypted\_data) + decryptor.finalize() return decrypted\_message.decode()

* **Base64 Decoding**: Converts the encoded string back into bytes.
* **Extract Salt and IV**: The first 16 bytes are the salt, and the next 16 bytes are the IV.
* **Cipher**: Recreated with the same key derivation and CFB mode for decryption.
* **Decryptor**: Reverses the encryption to retrieve the original plaintext message.

**Cryptographic Concepts**

1. **AES (Advanced Encryption Standard)**:
   * A symmetric encryption algorithm widely used for securing data.
   * In this code, it operates in **CFB (Cipher Feedback) mode**, which turns a block cipher into a stream cipher, allowing for secure encryption of data of arbitrary length.
2. **PBKDF2 (Password-Based Key Derivation Function 2)**:
   * Used to derive a secure cryptographic key from a password.
   * Incorporates a salt and multiple iterations to thwart brute-force and dictionary attacks.
3. **Salt**:
   * A random value added to the password before key derivation to ensure that the same password yields different keys across different instances.
4. **IV (Initialization Vector)**:
   * Ensures that identical plaintexts encrypt to different ciphertexts even with the same key, adding an additional layer of security.

**Encryption Process Diagram**

**Steps:**

1. **Generate Salt and IV**: Create random values for salt and IV.
2. **Key Derivation**: Use PBKDF2 to derive a key from the password and salt.
3. **Encryption**: Encrypt the message using AES in CFB mode with the derived key and IV.
4. **Concatenate and Encode**: Combine the salt, IV, and encrypted message, then encode in Base64 for safe transmission.

**Diagram:**

plaintext

Copy code

+------------+ | Password | +------------+ | v +----------------+ | Generate Salt |<---+ | (16 bytes) | | +----------------+ | | | v | +----------------+ | | PBKDF2 Key | | | Derivation | | +----------------+ | | Derived Key | | +----------------+ | | | v | +----------------+ | | Generate IV | | | (16 bytes) | | +----------------+ | | | v | +----------------+ | | AES-CFB | | | Encryption | | +----------------+ | | | v | +-------------------------------+ | Encrypted Message (Ciphertext)| +-------------------------------+ | v +----------------+ | Concatenate | | Salt + IV + | | Ciphertext | +----------------+ | v +----------------+ | Base64 Encode | +----------------+ | v +----------------+ | Encrypted Data | +----------------+

**Decryption Process Diagram**

**Steps:**

1. **Decode and Extract**: Decode the Base64 string and extract the salt, IV, and encrypted message.
2. **Key Derivation**: Use PBKDF2 to derive the key from the password and extracted salt.
3. **Decryption**: Decrypt the encrypted message using AES in CFB mode with the derived key and IV.
4. **Obtain Original Message**: The result is the original plaintext message.

**Diagram:**

plaintext

Copy code

+----------------+ | Encrypted Data | +----------------+ | v +----------------+ | Base64 Decode | +----------------+ | v +----------------+ | Extract Salt | | (16 bytes) | +----------------+ | v +----------------+ | Extract IV | | (16 bytes) | +----------------+ | v +----------------+ | Extract | | Ciphertext | +----------------+ | v +------------+ | Password | +------------+ | v +----------------+ | PBKDF2 Key | | Derivation | +----------------+ | Derived Key | +----------------+ | v +----------------+ | AES-CFB | | Decryption | +----------------+ | v +-------------------------------+ | Original Message (Plaintext) | +-------------------------------+

**Detailed Concepts**

1. **Salt**:
   * Randomly generated for each encryption to ensure uniqueness.
   * Mixed with the password in the key derivation process to produce a unique derived key.
2. **IV (Initialization Vector)**:
   * Randomly generated for each encryption.
   * Ensures that even if the same plaintext is encrypted multiple times with the same key, the ciphertext will be different each time.
3. **PBKDF2 with HMAC-SHA256**:
   * Strengthens password-based key derivation by applying a hash function (SHA-256) iteratively.
   * Makes it computationally expensive to perform brute-force attacks.
4. **AES in CFB Mode**:
   * A block cipher mode that turns a block cipher into a stream cipher.
   * Encrypts data of arbitrary length and ensures that each block of plaintext affects subsequent blocks.

These diagrams illustrate the encryption and decryption process steps in a clear, step-by-step manner, making the cryptographic principles and operations involved more understandable.

**WebSocket Protocol and Traffic Analysis**

The WebSocket protocol facilitates full-duplex communication channels over a single TCP connection. This enables real-time data exchange between a client and server, making it ideal for applications like chat systems, live updates, and interactive web applications. Here's an in-depth explanation of WebSocket communication, the handshake process, and the analysis of the provided traffic captures.

|  |
| --- |
|  |

**WebSocket Handshake**

The WebSocket handshake is a crucial part of establishing a WebSocket connection. It involves an initial HTTP request from the client to upgrade the connection to WebSocket, followed by an HTTP response from the server confirming the upgrade.

**Client Handshake Request**

The client's handshake request looks like this:

GET /socket.io/?transport=websocket&EIO=4&t=1715884623.6839314 HTTP/1.1

**Upgrade:** websocket

**Host:** real-time-chat-api-v1.onrender.com

**Origin:** https://real-time-chat-api-v1.onrender.com

**Sec-WebSocket-Key:** uQspeeKqLT50QvmtRt0tPA==

**Sec-WebSocket-Version:** 13

**Connection:** UpgradeKey

headers in this request:

* **Upgrade**: Indicates the desired protocol upgrade (to WebSocket).
* **Sec-WebSocket-Key**: A base64-encoded random value for security.
* **Sec-WebSocket-Version**: Specifies the WebSocket protocol version.
* **Connection**: Specifies that the connection should be upgraded.

**Server Handshake Response**

The server responds with:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: upgrade

Sec-WebSocket-Accept: I8pB0p3CExA/tzbMmyHPmCy5PeU=

Key headers in this response:

* **101 Switching Protocols**: Confirms the protocol switch.
* **Sec-WebSocket-Accept**: A base64-encoded string derived from the **Sec-WebSocket-Key** and a GUID, ensuring the handshake is valid.

**WebSocket Frames**

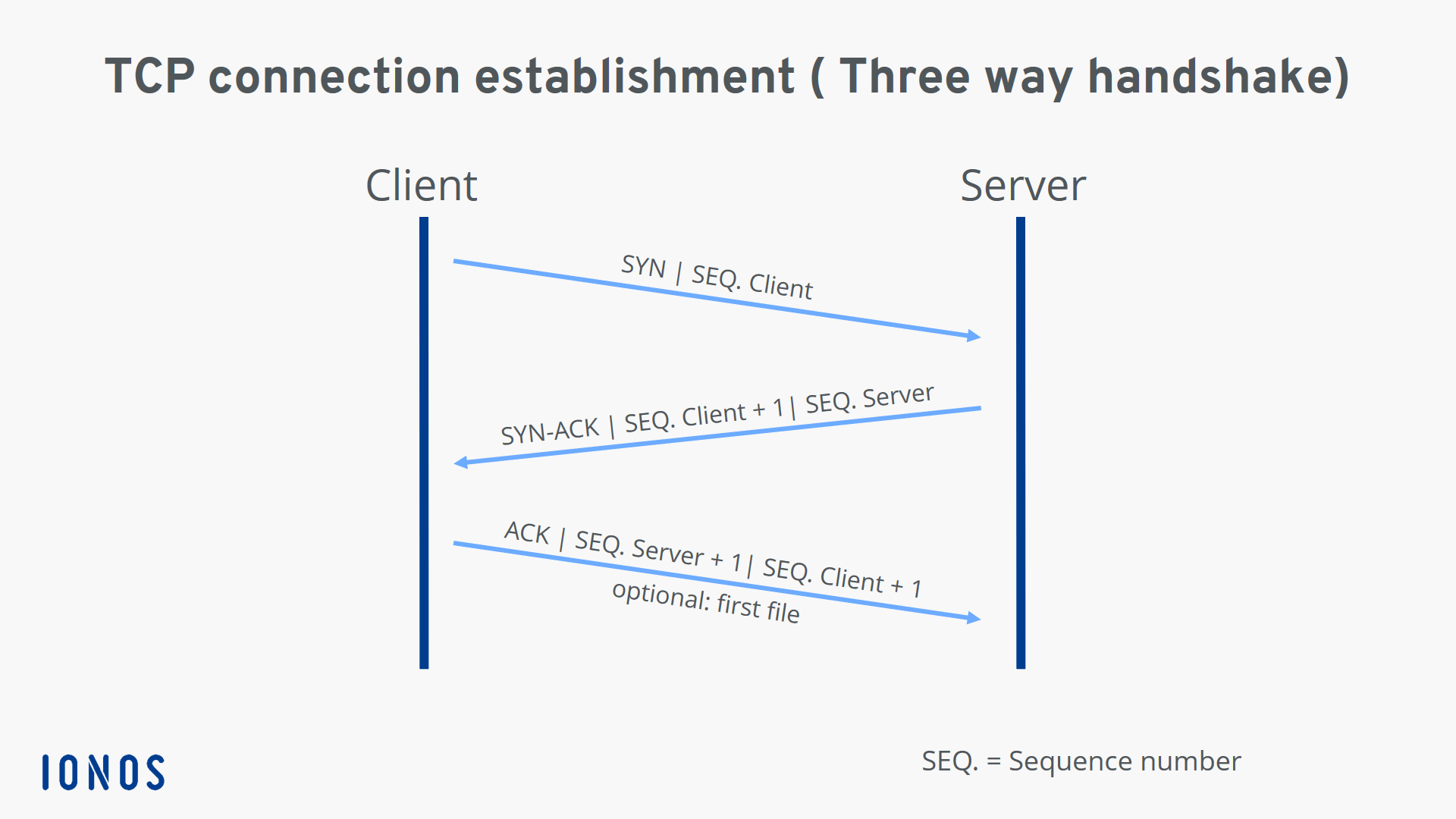
After the handshake, communication between the client and server happens through WebSocket frames. These frames can contain text, binary data, or control messages. Here’s an example of a WebSocket frame containing text data:

.k0{"sid":"q3muN2cszRjpIOr3AAAE","upgrades":[],"pingInterval":25000,"pingTimeout":20000,"maxPayload":1000000}

**TCP and TLS Handshake Overview**

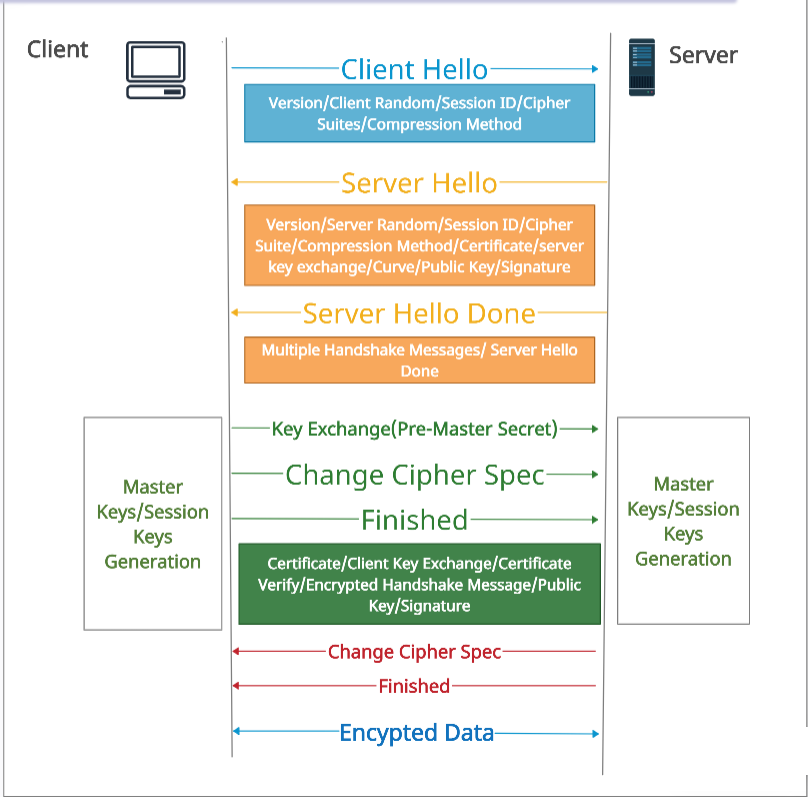
Before WebSocket communication, a TCP and TLS handshake ensures a secure connection. Here's a breakdown of the handshake process captured in your network traffic:

**TCP Handshake**



1. **SYN**: The client initiates a connection with a SYN packet.
2. **SYN-ACK**: The server responds with a SYN-ACK packet.
3. **ACK**: The client sends an ACK packet, completing the three-way handshake.

**TLS Handshake**

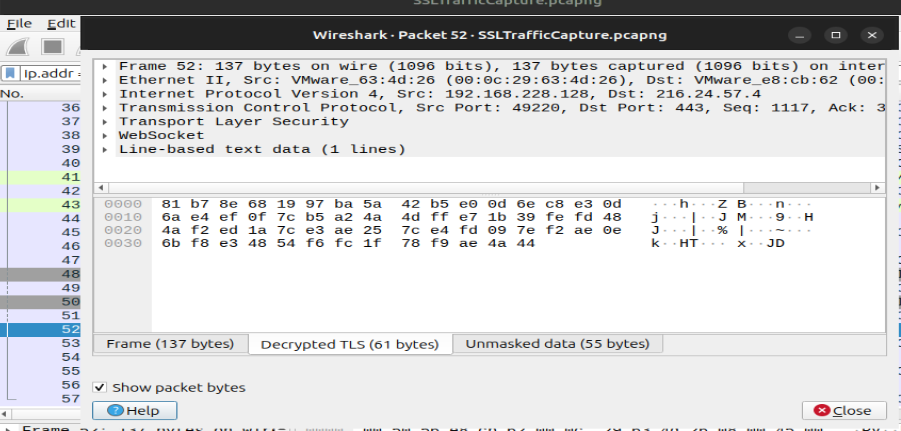


1. **Client Hello**: The client sends a **Client Hello** message to start the TLS handshake.
2. **Server Hello**: The server responds with a **Server Hello** message.
3. **Certificate, Change Cipher Spec, Finished**: The server sends its certificate, changes cipher specifications, and sends a **Finished** message.
4. **Client Change Cipher Spec, Finished**: The client sends its **Change Cipher Spec** and **Finished** messages.

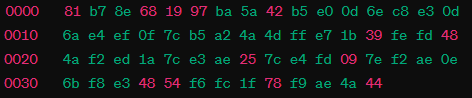
**WebSocket Frame Analysis**

**Masked Payload**

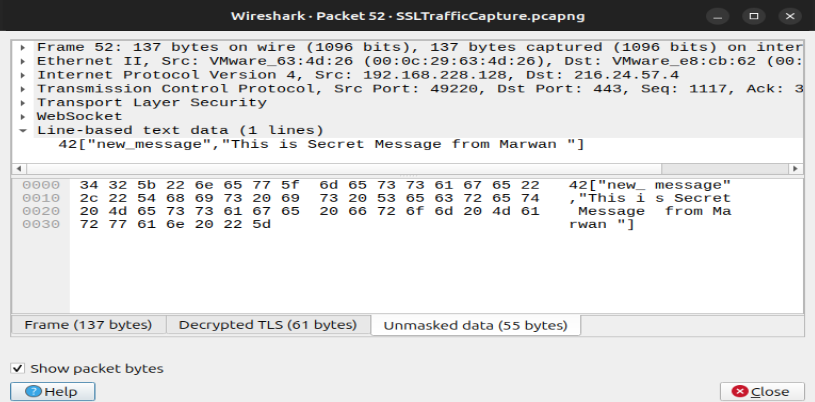
Client-to-server WebSocket frames must be masked to prevent certain types of attacks.



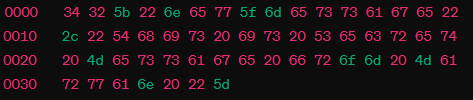
**Masked Payload:**



**Unmasked Payload**

****

Server-to-client WebSocket frames typically are not masked:



**Converting Masked to Unmasked Payload**

To convert a masked payload to its original form, XOR each byte of the masked payload with the corresponding byte of the masking key.

* **Masked Payload**: **81 b7 8e 68 19 97 ba 5a 42 b5 e0 0d 6e c8 e3 0d**
* **Masking Key**: **8e 68 19 97**

We will XOR each byte of the payload with the masking key, repeating the key as necessary.

1. Masked byte `8e` XOR Masking key `8e` = 0x00

2. Masked byte `68` XOR Masking key `68` = 0x00

3. Masked byte `19` XOR Masking key `19` = 0x00

4. Masked byte `97` XOR Masking key `97` = 0x00

5. Masked byte `ba` XOR Masking key `8e` = 0x34

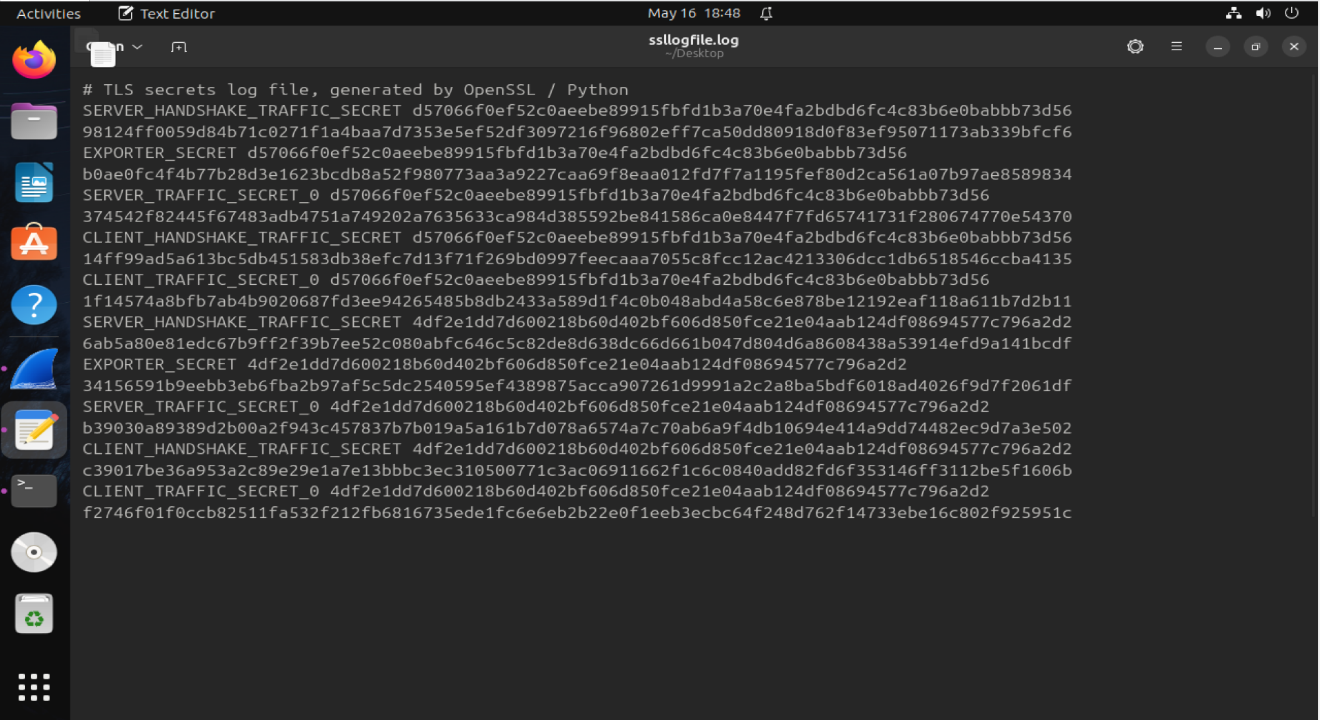
6. Masked byte `5a` XOR Masking key `68` = 0x32

7. Masked byte `42` XOR Masking key `19` = 0x5b

8. Masked byte `b5` XOR Masking key `97` = 0x22

XOR each byte of the payload with the masking key to get the original message.

**Traffic Analysis and TLS Secrets**



1. **Client Handshake Traffic Secret (CLIENT\_HANDSHAKE\_TRAFFIC\_SECRET)**
   * This key is used by the client to encrypt the handshake messages it sends to the server.
2. **Server Handshake Traffic Secret (SERVER\_HANDSHAKE\_TRAFFIC\_SECRET)**
   * This key is used by the server to encrypt the handshake messages it sends to the client.
3. **Client Traffic Secret 0 (CLIENT\_TRAFFIC\_SECRET\_0)**
   * This key is used by the client to encrypt application data after the handshake is complete and during the initial stages of the connection (also known as "early data" or "0-RTT data").
4. **Server Traffic Secret 0 (SERVER\_TRAFFIC\_SECRET\_0)**
   * This key is used by the server to encrypt application data after the handshake is complete.
5. **Exporter Secret (EXPORTER\_SECRET)**
   * This key is used to derive additional keys for specific purposes such as keying material exporters, which can be used for purposes like channel binding or exporting keys for other cryptographic operations.

**Explanation of Each Key Block**

Each key block in the log file consists of two parts:

* The first part is a session identifier or a unique value to differentiate different sets of keys.
* The second part is the actual secret key used for encryption or decryption.

**Breakdown**

1. **SERVER\_HANDSHAKE\_TRAFFIC\_SECRET**
   * Session Identifier: **d57066f0ef52c0aeebe89915fbfd1b3a70e4fa2bdbd6fc4c83b6e0babbb73d56**
   * Secret Key: **98124ff0059d84b71c0271f1a4baa7d7353e5ef52df3097216f96802eff7ca50dd80918d0f83ef95071173ab339bfcf6**
   * Usage: Used by the server to encrypt handshake messages.
2. **EXPORTER\_SECRET**
   * Session Identifier: **d57066f0ef52c0aeebe89915fbfd1b3a70e4fa2bdbd6fc4c83b6e0babbb73d56**
   * Secret Key: **b0ae0fc4f4b77b28d3e1623bcdb8a52f980773aa3a9227caa69f8eaa012fd7f7a1195fef80d2ca561a07b97ae8589834**
   * Usage: Used to derive additional keys for specific cryptographic operations.
3. **SERVER\_TRAFFIC\_SECRET\_0**
   * Session Identifier: **d57066f0ef52c0aeebe89915fbfd1b3a70e4fa2bdbd6fc4c83b6e0babbb73d56**
   * Secret Key: **374542f82445f67483adb4751a749202a7635633ca984d385592be841586ca0e8447f7fd65741731f280674770e54370**
   * Usage: Used by the server to encrypt application data after the handshake.
4. **CLIENT\_HANDSHAKE\_TRAFFIC\_SECRET**
   * Session Identifier: **d57066f0ef52c0aeebe89915fbfd1b3a70e4fa2bdbd6fc4c83b6e0babbb73d56**
   * Secret Key: **14ff99ad5a613bc5db451583db38efc7d13f71f269bd0997feecaaa7055c8fcc12ac4213306dcc1db6518546ccba4135**
   * Usage: Used by the client to encrypt handshake messages.
5. **CLIENT\_TRAFFIC\_SECRET\_0**
   * Session Identifier: **d57066f0ef52c0aeebe89915fbfd1b3a70e4fa2bdbd6fc4c83b6e0babbb73d56**
   * Secret Key: **1f14574a8bfb7ab4b9020687fd3ee94265485b8db2433a589d1f4c0b048abd4a58c6e878be12192eaf118a611b7d2b11**
   * Usage: Used by the client to encrypt application data after the handshake.

These keys are used during various stages of the TLS handshake and session to encrypt and decrypt traffic, ensuring secure communication.

WebSocket protocol, layered on top of TCP and TLS, provides a robust mechanism for real-time communication. Understanding the handshake process, frame structure, and traffic analysis helps in diagnosing and securing WebSocket communications. The conversion of masked payloads and the use of TLS secrets are critical for decrypting and analyzing the traffic in secure WebSocket sessions.

Results & Analysis