8/7/2023

SECURE CHAT APP

RSA & AES Encryption

|  |
| --- |
| **Students Names / ID Number** |
| Douglas Odida - 668258 |
| Emmanuel Obinemelu - 661212 |
| Josphat Mbugua - 629752 |
| Kevin Mwangi - 667955 |
| Mathew Saha - 664505 |
| Sebastian Opiyo - 668072 |



ISC 6220A

USIU

**Lab Report: Secure Chat Application**

**Abstract**

This report breaks down a Secure Chat Application developed in Flask and uses WebSocket to facilitate real-time chat. RSA and AES encryption methods are employed to ensure the security and confidentiality of messages.

**Overview of Components:**

**Client Application:** This application is mainly built using Flask and is intended for end-users. It provides routes to connect, disconnect, send messages, and login. For real-time communication, the application uses WebSocket through Flask's SocketIO.

**Server Application:** This application also uses Flask and handles the back-end logic, including the management of WebSocket connections, encryption/decryption, and broadcasting messages to connected clients.

**System Data Flow Diagram**

A diagram of a chat

Description automatically generated

**System Sequence Diagram**

A screenshot of a computer screen

Description automatically generated

**Detailed Function Analysis:**

**Client Application:**

***Initialization:***

* Flask application and SocketIO are initialized.
* A secret key for Flask is either retrieved from an environment variable or a default is set.

***“/connect” Endpoint:***

* Checks if the username exists in the session (meaning a user has logged in).
* If not, it redirects to the login page. Otherwise, a success message is flashed.

***“/disconnect” Endpoint:***

* Disconnects the WebSocket by removing the username from the session.
* Notifies the user that they've been disconnected.

***“/” Endpoint:***

* Root endpoint which gives a prompt to login.

***“/login” Endpoint:***

* If POST request, the application takes the username and password from the form, makes a request to the server's login API, and if successful, sets the session's username and redirects to the chat.
* If unsuccessful or any other method, it renders the login page.

***“/chat” Endpoint:***

* For POST requests, it sends the chat message to the server. For GET requests, it retrieves chat messages.
* Renders the chat page with messages and user details.

***SocketIO Message Handling:***

* The handle\_message function deals with real-time messaging, getting the message from the client, and then sending it to other connected clients.

**Server Application:**

***Initialization:***

* Flask application is initialized with CSRF protection.
* Secret key for Flask is set.

***“/api/” Endpoint:***

* Returns a welcome message.

***“/api/login” Endpoint:***

* For each client trying to log in, it generates a pair of RSA keys for the client and the server.
* These keys are stored in the session and are also returned to the client. This ensures that the client and server can securely communicate using RSA.

***“/api/chat” Endpoint:***

* This endpoint handles the receipt and sending of messages.
* When a message is received, the text is encrypted using AES with the client's public key. The AES encrypted message is then signed using RSA with the server's private key to ensure integrity.
* The encrypted and signed message is then sent (broadcasted) to all connected clients.

***“/api/connect” and “/api/disconnect” Endpoints:***

* These handle the WebSocket connections, allowing clients to connect and disconnect from the server.

**Encryption Analysis**

***RSA Encryption:***

* RSA is an asymmetric encryption algorithm. This means there are two keys: a public key, which can be shared openly, and a private key, which must remain confidential. For demo purposes we are using 512-bit keys for speed, but in a production system it is advisable to use 2048-bit keys.
* # Generate new RSA keys for the client and the server
* client\_key = rsa.newkeys(512)
* server\_key = rsa.newkeys(512)
* In the server's */api/login* endpoint, RSA key pairs are generated for both the server and client. The public keys are shared with the client. We are then storing the keys and provided username/password in a communication session.
* # Store the keys and username in the session
* flask.session["client\_key"] = client\_key[0].save\_pkcs1().decode()
* flask.session["server\_key"] = server\_key[0].save\_pkcs1().decode()
* flask.session["username"] = username
* During the chat process, after AES encryption of the message, the server signs the encrypted message using its RSA private key. This signature allows clients to verify the message's authenticity and integrity: if the signature matches when verified with the server's public key, the message is authentic and hasn't been tampered with.

@app.route("/api/chat", methods=["POST"])

def chat():

"""

This endpoint handles the chat messages.

It gets the client and server keys from the session and loads them.

It then gets the JSON data from the request, and for each message in the data,

it encrypts the message text using AES with the client's public key and signs the ciphertext with the server's private key.

It then broadcasts the encrypted and signed message to all connected clients.

"""

# Get the username from the session

username = flask.session.get("username")

if not username:

return flask.jsonify({"error": "User not logged in."})

# Get the client and server keys from the session and load them

client\_key\_data = flask.session["client\_key"]

server\_key\_data = flask.session["server\_key"]

client\_key = rsa.PublicKey.load\_pkcs1(client\_key\_data.encode())

server\_key = rsa.PublicKey.load\_pkcs1(server\_key\_data.encode())

***AES Encryption:***

* In symmetric encryption like AES, the same key is used for both encryption and decryption. It's faster and suitable for encrypting large data, such as the body of a chat message.
* In the code, the server uses the client's RSA public key to encrypt the chat message using AES. This means that only the client with the matching RSA private key can decrypt the AES key and, subsequently, the message.
* try:
* # Get the JSON data from the request
* data = flask.request.get\_json()
* if not isinstance(data, list):
* raise ValueError("Invalid JSON data. Expected a list.")
* messages = []
* for message in data:
* # Validate the message format
* if not isinstance(message, dict) or "text" not in message:
* raise ValueError("Invalid message format.")
* # Get the message text from the message
* message\_text = message["text"]
* # Encrypt the message text using AES with the client's public key
* message\_ciphertext = aes.encrypt(message\_text.encode(), client\_key)
* # Sign the ciphertext with the server's private key
* message\_signature = rsa.sign(message\_ciphertext, server\_key)
* # Add the encrypted and signed message to the messages list
* messages.append({
* "sender": username,
* "text": message\_text,
* "ciphertext": message\_ciphertext.decode(),
* "signature": message\_signature.decode(),
* "timestamp": datetime.now(timezone.utc).replace(microsecond=0).isoformat()
* })
* # Broadcast the message to all connected clients
* for client in connected\_clients:
* client.send(flask.jsonify({"data": messages}).data)
* return flask.jsonify({"success": True})
* except ValueError as e:
* return flask.jsonify({"error": str(e)})

**Deployment Process**

For ease of deployment and testing, the application has been configured to run via docker. This allows for consistent deployment and results due to the system agnostic nature of docker.

The Docker Compose file provided specifies a deployment involving two services: ***server*** and ***client***. Both services are built from their respective ***Dockerfiles*** and are deployed as individual containers: server and client.

**Network Setup**

A network named ***chatnetwork*** is created with Docker's default bridge network driver. This setup allows containers to communicate across the network while isolating them from outside networks.

**Server Service**

The server service is built from the ***Dockerfile*** located in the ***./server*** directory. The service's container is named server. The service exposes port 8000 on the host machine, mapped to port 8000 in the container. This service is a part of the ***chatnetwork*** network.

**Client Service**

The client service is built from the ***Dockerfile*** located in the ***./client*** directory. The service's container is named client. The service exposes port 5000 on the host machine, mapped to port 5000 in the container. The client service depends on the server service, which means Docker Compose will ensure the server is started before starting the client. This service is also part of the ***chatnetwork*** network.

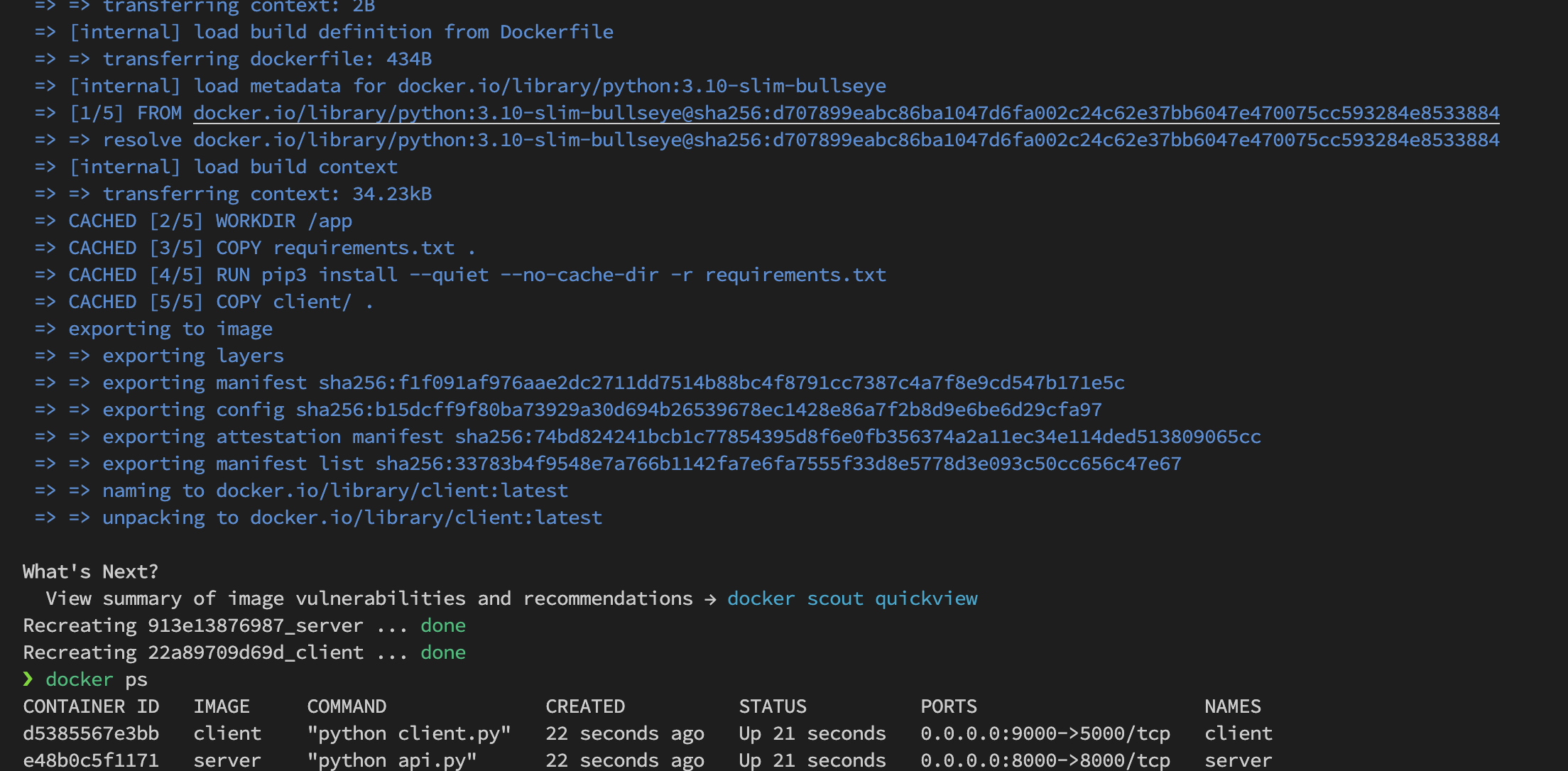
***Deployment Process***

1. Docker Compose will initiate the network ***chatnetwork*** using the bridge driver.
2. The ***server*** service will be built using the context from the current directory and the ***Dockerfile*** from ***./server/Dockerfile***. The service is deployed and named as the server container.
3. The ***client*** service will be built using the context from the current directory and the ***Dockerfile*** from ***./client/Dockerfile***. As the client service depends on the server service, Docker Compose will ensure that server is up and running before starting the client service. The service is deployed and named as the client container.

**Sample outputs**

***Start the deployment:***

❯ docker-compose up --build -d



***Root Endpoint***

A screenshot of a computer

Description automatically generated

***Login Endpoint***

A screenshot of a computer

Description automatically generated

***Chat Endpoint***

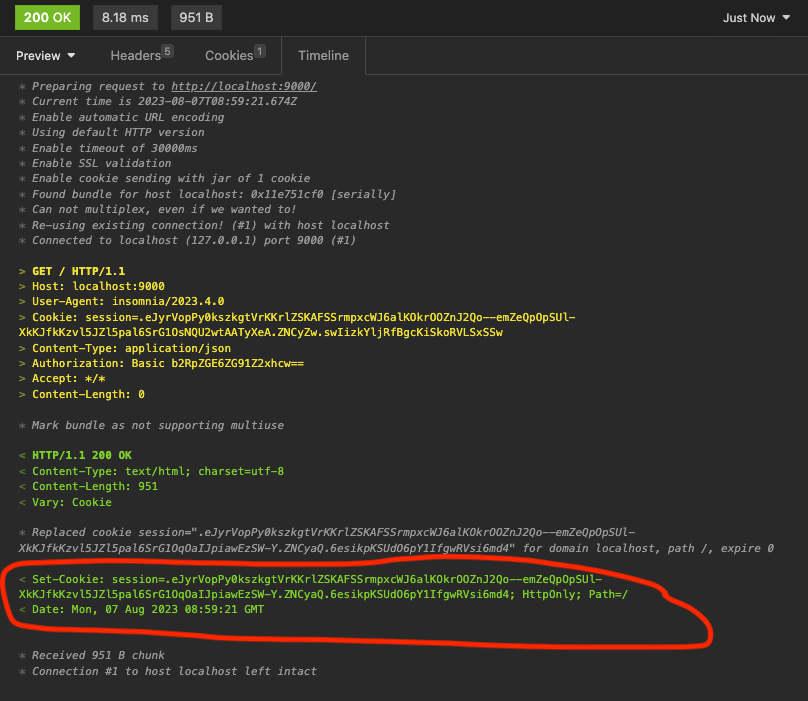
A screenshot of a chat

Description automatically generated

***Chat Demo***

|  |  |
| --- | --- |
| A screenshot of a chat  Description automatically generated | A screenshot of a chat  Description automatically generated |

***Flask session cookie generation***



***RSA Client and server keys generation***

A screenshot of a computer

Description automatically generated

**Conclusion**

The secure chat application leverages the benefits of both AES and RSA encryption to ensure confidential, integral, and authentic communication between users. While AES ensures the message remains confidential, RSA provides the means for users to verify the authenticity and integrity of messages. This dual-layer encryption approach makes the chat system robust against eavesdropping and tampering.