Alessia Vargiu 71/10/00049
Laura Di Palma 71/10/00044

PROJECT REPORT: DESIGN AND IMPLEMENTATION OF A SECURE END-TO-END ENCRYPTED MESSAGING ARCHITECTURE

1. Introduction

In the contemporary digital landscape, the confidentiality and integrity of communication channels are paramount. As network adversaries become more sophisticated, relying solely on transport layer security is no longer sufficient to guarantee the privacy of sensitive data. This project focuses on the design, implementation and verification of a secure client-server-client messaging system.

The objective was to create a "Zero-Knowledge" architecture where the central server functions exclusively as a relay for encrypted traffic, possessing no capability to decrypt the payloads it transports. To achieve this, the system implements a robust defense-in-depth strategy, layering Mutual TLS (mTLS) for network authentication with a custom End-to-End Encryption (E2EE) protocol using a hybrid cryptographic scheme (RSA-2048 for key exchange and AES-256-GCM for payload encryption). Furthermore, the architecture addresses advanced threat vectors, including Replay Attacks, Spoofing, Traffic Analysis, and Denial of Service (DoS) attempts.

2. The Foundation of Trust: Public Key Infrastructure (PKI)

The security of any distributed system is predicated on the strength of its identity management. Rather than relying on third-party authorities, this project establishes a private, self-contained Public Key Infrastructure (PKI) to govern the network's trust relationships.

The process begins with the generation of a high-entropy RSA key pair for the "NetSec Root CA," which is then used to self-sign a root certificate. This Certificate Authority serves as the ultimate arbiter of trust; only entities possessing certificates signed by this root are permitted to participate in the network. This closed ecosystem is critical for a high-security internal communication tool.

Following the establishment of the CA, the script generates specific identity certificates for the server and the clients (Alice and Bob). Unlike standard client certificates used solely for authentication, our certificates are generated with an enhanced extension: it explicitly permits

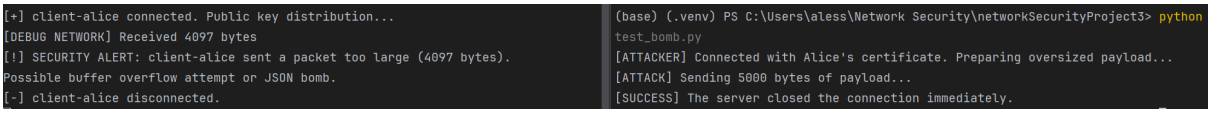
the use of the client's RSA public key to encrypt the symmetric session keys generated by other peers. Without the KeyEncipherment flag, strict SSL/TLS implementations or cryptographic libraries would reject the RSA encryption operation required for the hybrid E2EE scheme.

3. Server-Side Security and Transport Layer Defense

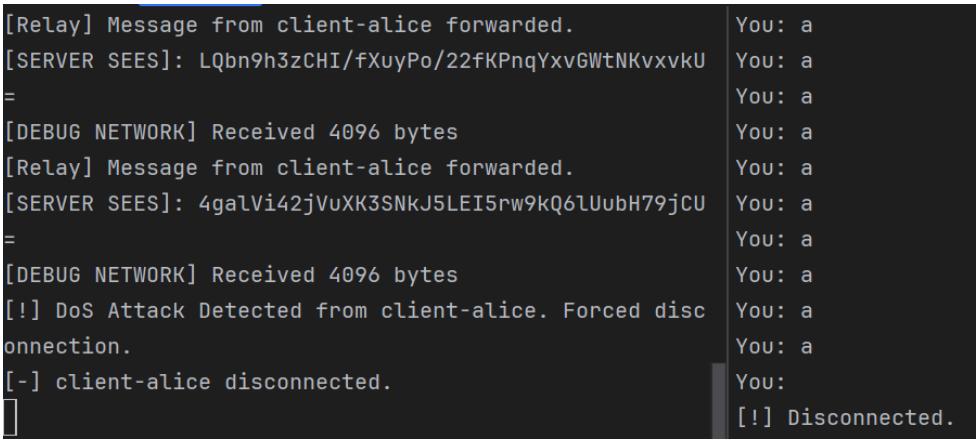
The server is implemented in `server.py`. Its primary responsibility is to maintain the availability of the network while enforcing strict authentication boundaries. The first line of defense is the implementation of Mutual TLS (mTLS). This server configures its SSL context with the `ssl.CERT_REQUIRED` verification mode. This mandates that any client attempting to initiate a TCP handshake must present a valid certificate signed by the local CA. If a scanner, bot, or unauthorized user attempts to connect without these credentials, the TLS handshake fails immediately, effectively rendering the server invisible to unauthorized scans and eliminating a vast class of opportunistic attacks.

Once a secure tunnel is established, the server performs a critical "Identity Binding" check to prevent internal spoofing. In a networked environment, a valid user (Alice) might maliciously attempt to impersonate another valid user (Bob). To mitigate this, the server extracts the Common Name (CN) from the client's X.509 certificate and compares it against the `sender` field declared in the incoming JSON message structure. If a discrepancy is detected—for instance, if the certificate belongs to "client-alice" but the message claims to be from "client-bob"—the server identifies this as a spoofing attempt and discards the packet. This logic effectively neutralizes the attack vector demonstrated in the `test_spoofing.py` script, where a malicious payload with a falsified sender identity is rejected despite being sent over a valid SSL connection.

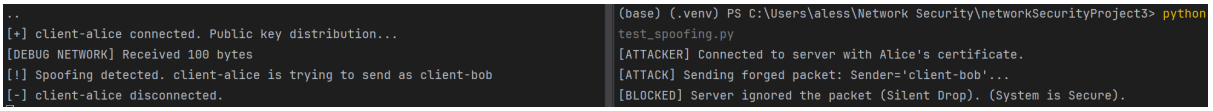
Beyond authentication, the server is engineered to withstand Denial of Service (DoS) attacks targeting the application layer. The system implements a rigorous Rate Limiting mechanism using a sliding window algorithm. The server tracks the timestamps of incoming messages for each client connection; if a client exceeds a defined threshold of ten messages within a five-second window, the server classifies the behavior as a DoS attempt and forcibly terminates the connection. This protects the server's computational resources from being overwhelmed by flooding attacks. Additionally, the server enforces strict input validation regarding message size. By pre-allocating a receive buffer of 4096 bytes (plus one byte for overflow detection), the server proactively defends against Buffer Overflow and Memory Exhaustion attacks. As validated by the `test_bomb.py` script, which attempts to send a 5000-byte payload, the server detects the violation of the `MAX_MSG_SIZE` constant and severs the connection before attempting to parse the potentially malicious data. This ensures that the server remains stable even under stress from malformed packets.



test_bomb.py



Attempt of DoS



test_spoofing.py

4. Client Implementation and End-to-End Encryption

The core of the project's confidentiality guarantee lies within the `client.py` module, which implements a hybrid encryption scheme. Upon joining the network, clients receive the public keys of other active users via a broadcast mechanism. When a user wishes to send a message, the client does not rely on static keys for the payload. Instead, it generates a fresh, ephemeral 32-byte AES session key using `os.urandom(32)` for every single message. This approach significantly limits the blast radius of a potential key compromise; if an attacker manages to recover a session key, they can decrypt only that specific message, not the entire conversation history. This session key is then encrypted using the recipient's RSA public key with OAEP padding, ensuring that only the intended recipient possesses the cryptographic material necessary to unlock the payload.

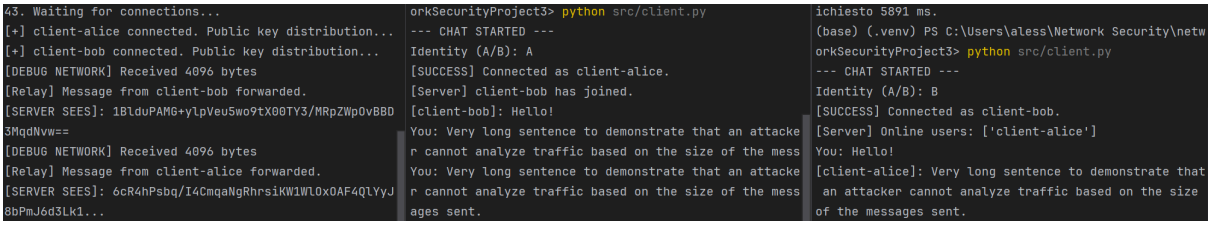
For the encryption of the message body, the system utilizes the Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM). GCM is an authenticated encryption algorithm that provides both confidentiality and data integrity. A critical aspect of this implementation is the management of the Initialization Vector (IV) or Nonce. The client generates a unique 12-byte nonce for every encryption operation, preventing the "nonce reuse" vulnerability that can lead to the recovery of the authentication key in GCM.

A distinguishing feature of this implementation is the use of Additional Authenticated Data (AAD) to bind the metadata to the ciphertext. The client constructs an AAD string containing the sender's identity and the timestamp and passes this to the encryption engine. This cryptographic binding ensures that the metadata visible in the JSON header, which the server uses for routing, cannot be tampered with by a Man-in-the-Middle without invalidating the decryption tag. If an attacker intercepts a packet and modifies the timestamp to alter the message order, or changes the sender field, the recipient's decryption process will fail with an InvalidTag error, alerting the user to the integrity violation. This effectively extends the integrity guarantee of the encrypted payload to the plaintext routing information.

5. Advanced Defensive Mechanisms: Anti-Replay and Traffic Analysis

Beyond standard encryption, the system addresses more subtle network threats such as Replay Attacks and Traffic Analysis. A Replay Attack occurs when an adversary intercepts a valid encrypted packet and re-transmits it later to duplicate an action or confuse the recipient. To counter this, the client implements a dual-layer defense. First, it enforces a strict temporal window: any message with a timestamp older than 60 seconds is immediately discarded. Second, to prevent replay within that 60-second window, the client maintains a cache of message signatures. It calculates a SHA-256 hash of the encrypted payload combined with the timestamp and stores it in a `seen_signatures` set. If a received message generates a hash that already exists in this set, it is identified as a duplicate and silently ignored. This ensures that every processed message is cryptographically unique and timely.

Furthermore, the system mitigates Traffic Analysis, a technique where an eavesdropper infers the content of a conversation by observing the size of the encrypted packets (e.g., distinguishing a short "yes" from a long paragraph). The client application implements a padding strategy that standardizes the size of all outgoing packets. Before transmission, the system calculates the length of the JSON object and appends a variable amount of empty space to a padding field, ensuring that every packet sent over the wire is exactly 4096 bytes in length. This creates a uniform traffic profile, rendering the encrypted messages indistinguishable from one another in terms of size and obscuring the nature of the communication from network observers.



Traffic analysis

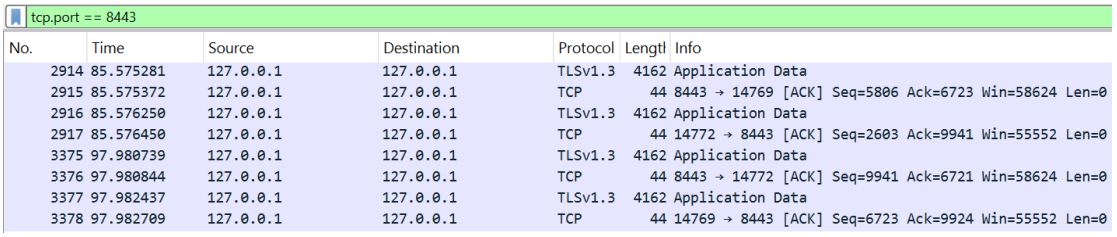
6. Network traffic analysis: Wireshark validation

To strictly validate the security controls, a packet-level analysis was conducted using Wireshark monitoring the loopback interface (`tcp.port == 8443`). This analysis confirms that the theoretical protections implemented in Python translate into effective network-level defenses.

6.1 Confidentiality and Traffic Analysis Mitigation

To validate the system's confidentiality and resistance to traffic analysis, a test was conducted by capturing network traffic during a chat session where messages of varying lengths, ranging from short greetings like "Hello" to extensive paragraphs, were exchanged. The resulting Wireshark analysis confirms the efficacy of the TLS tunnel, as only `TLSv1.3` protocol headers and encrypted `Application Data` are visible on the wire, with no exposure of plaintext JSON or readable message content.

Crucially, an examination of the packet details highlights the success of the padding mechanism. As evidenced by the *Length* column in the capture, every data packet transmitted by the client arrives with a consistent size (specifically 4162 bytes), regardless of the actual dimensions of the original message. This fixed-size transmission strategy effectively masks message metadata, neutralizing the threat of statistical traffic analysis by preventing eavesdroppers from distinguishing between short commands and lengthy communications based on packet size.



6.2 Mutual TLS (mTLS) Enforcement

To rigorously verify the authentication protocols within the **TLS 1.3** environment - where handshake messages such as the *Certificate Request* are encrypted by design - a negative testing methodology was adopted. An unauthorized connection attempt was simulated using an OpenSSL client lacking the necessary certificate credentials, forcing the server to react to an anonymous request.

```
(base) (.venv) PS C:\Users\aless\Network Security\networkSecurityProje
ct3> openssl s_client -connect localhost:8443
```

```
read:errno=10054
```
Figure A

```
(base) (.venv) PS C:\Users\aless\Networkpython src/server.py
Server started successfully. Listening on localhost:8443. Waiting for connections...
Connection error: [SSL: PEER_DID_NOT_RETURN_A_CERTIFICATE] peer did not return a certificate (_ssl.c:1007)
```

Figure B

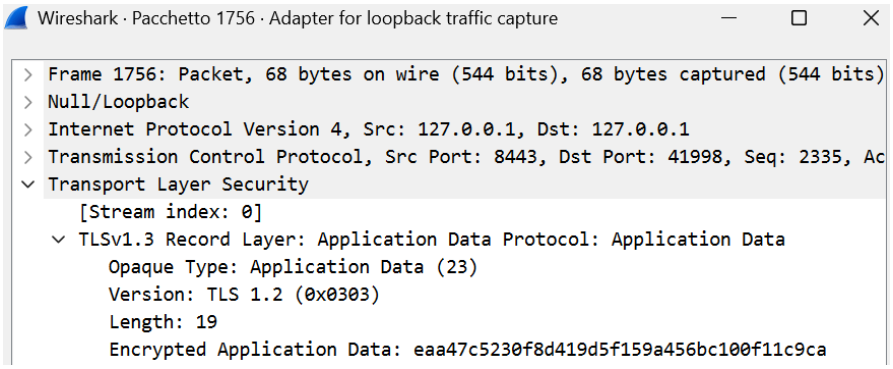
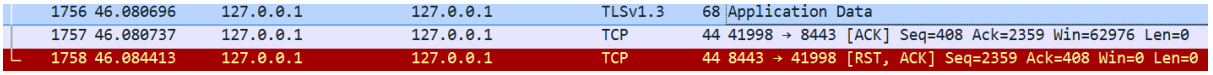


Figure C



The effectiveness of this security control is evidenced by the dual-perspective analysis shown in the figures above. As observed in the terminal output (Figure B), the server-side logs explicitly identify the violation with the error `[SSL: PEER_DID_NOT_RETURN_A_CERTIFICATE]`, confirming that the connection was rejected specifically due to the absence of client credentials. Consequently, the client session was abruptly terminated with a connection reset (Figure A, `errno=10054`).

At the network level, the Wireshark capture (Figure C) illustrates the precise mechanics of this rejection. Due to TLS 1.3 privacy features, the "Certificate Required" fatal alert is transmitted as Encrypted Application Data rather than plaintext. This is identified in Packet 1756, which shows a TLS record with a length of only 19 bytes. This distinctively small size, contrasted with the standard 4096-byte payload, indicates an encrypted Alert message. The capture confirms that the server successfully established a minimal secure channel solely to transmit this "Forbidden" signal confidentially, immediately followed by a TCP Reset (Packet 1758) to effectively sever the unauthorized connection.

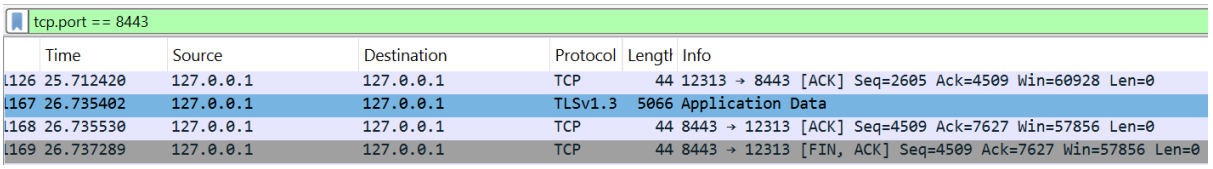
6.3 Buffer Overflow Protection Analysis

To validate the system's resilience against large payload attacks and memory exhaustion attempts, a traffic analysis was conducted during the execution of the `test_bomb.py` script.

The resulting Wireshark capture provides clear evidence of both the attack vector and the server's defensive response.

The attack footprint is visible in Packet 1167, which appears as a TLSv1.3 Application Data packet with a significant length of 5066 bytes. This specific size correlates precisely to the injected 5000-byte malicious payload combined with the necessary TLS encryption overhead, confirming that the oversized data successfully reached the server's network interface.

Crucially, the server's reaction was nearly instantaneous. As observed in Packet 1169, transmitted approximately 2 milliseconds after the payload's arrival, the server responded with a TCP packet flagged [FIN, ACK]. This immediate transmission of a termination signal demonstrates that the application logic successfully intercepted the oversized message at the socket level. Instead of attempting to buffer or parse the excess data, the server preemptively initiated a connection termination routine (`conn.close()`), effectively neutralizing the buffer overflow threat before any memory exhaustion could occur.



6.4 Identity Spoofing Mitigation (Silent Drop Analysis)

To rigorously verify the integrity of identity binding within the authenticated session, a specific test involving the injection of a spoofed payload was conducted. The network capture provides a timeline of the server's defensive behavior against this impersonation attempt.

The attack vector is identified in Packet 1907, transmitted at timestamp 34.05s. This packet appears as a 165-byte TLSv1.3 Application Data segment, carrying the encrypted JSON payload where the sender attempted to impersonate "client-bob".

The effectiveness of the mitigation is evidenced by the subsequent lack of activity. Following the injection, no outbound Application Data was observed originating from the server (Port 8443). Had the spoofing attempt been successful, the server would have immediately processed and broadcasted the message to other clients. Instead, the server maintained absolute silence at the application layer. This state persisted until timestamp 36.06s, exactly two seconds later, when the client, having received no response, initiated the connection closure via a [FIN, ACK] packet (Packet 1947). This sequence confirms that the server performed a "Silent Drop": upon validating the identity and detecting the mismatch, it deliberately ignored the malicious request without processing or acknowledging it, thereby effectively neutralizing the threat.

tcp.port == 8443

No.	Time	Source	Destination	Protocol	Lengtl	Info
1907	34.050169	127.0.0.1	127.0.0.1	TLSv1.3	165	Application Data
1908	34.050231	127.0.0.1	127.0.0.1	TCP	44	8443 → 62661 [ACK] Seq=4509 Ack=2726 Win=62720 Len=0
1947	36.066222	127.0.0.1	127.0.0.1	TCP	44	62661 → 8443 [FIN, ACK] Seq=2726 Ack=4509 Win=60928 Len=0
1948	36.066275	127.0.0.1	127.0.0.1	TCP	44	8443 → 62661 [ACK] Seq=4509 Ack=2727 Win=62720 Len=0
1949	36.066794	127.0.0.1	127.0.0.1	TCP	44	8443 → 62661 [FIN, ACK] Seq=4509 Ack=2727 Win=62720 Len=0
1950	36.066876	127.0.0.1	127.0.0.1	TCP	44	62661 → 8443 [ACK] Seq=2727 Ack=4510 Win=60928 Len=0

6.5 Denial of Service (DoS) Mitigation Analysis

To validate the system's availability protections, a "Flooding Attack" was simulated using a legitimate client account (`client-bob`) configured to transmit messages at a rate exceeding the defined threshold of 10 messages within 5 seconds.

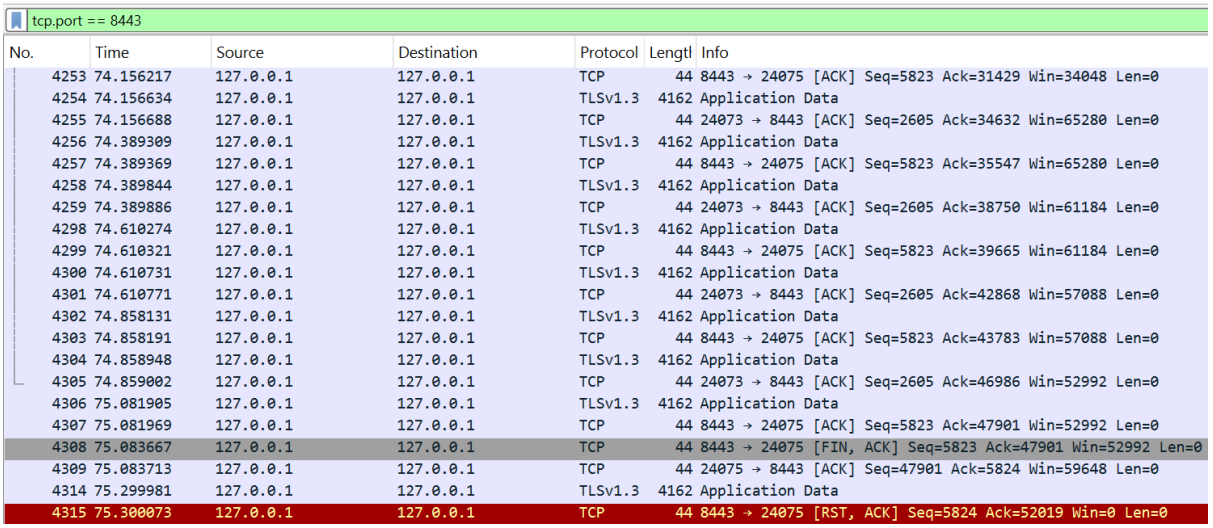
The efficacy of the defense mechanism is confirmed through a dual-perspective analysis. At the application layer (**Figure A**), the server logs demonstrate immediate detection of the anomaly. The security module successfully triggered the alert `[!] DoS Attack Detected from client-bob` and executed a forced disconnection routine to protect system resources.

This logical protection translates directly into network-level action, as visualized in the Wireshark capture (Figure B). The trace initially shows a rapid burst of `Application Data` packets (Sequence 4300-4306), representing the flooded messages; notably, traffic padding remained active during the attack, maintaining a consistent packet length of 4162 bytes. The server's mitigation response follows immediately in Packet 4308, where it transmits a TCP [FIN, ACK] signal, proactively initiating connection closure. The defense mechanism concludes decisively with Packet 4315 (highlighted in red), a TCP [RST, ACK] packet. This hard socket termination instantly severs the link, freeing up server resources and effectively neutralizing the attacker's ability to consume further bandwidth.

```
[SERVER SEES]: HZ6jylKF5pmzTPd1rHX1+KhYF9EuRyJka6oqiHg
=
[DEBUG NETWORK] Received 4096 bytes
[Relay] Message from client-bob forwarded.
[SERVER SEES]: RWek9+LyjGPOGc3s6tDejY4weom6Ugpak0nO4MY
=
[DEBUG NETWORK] Received 4096 bytes
[Relay] Message from client-bob forwarded.
[SERVER SEES]: adioQYFUOLEoxxfz1GSoDRofiJJ3bB/+kAuSf88
=
[DEBUG NETWORK] Received 4096 bytes
[Relay] Message from client-bob forwarded.
[SERVER SEES]: zdSurZajid+2oLxwkZu4Qf2niy0WcaQEinLS4vg
=
[DEBUG NETWORK] Received 4096 bytes
[!] DoS Attack Detected from client-bob. Forced discon
nection.
[-] client-bob disconnected.
```

```
Il caricamento dei profili personali e di sistema ha r
ichiesto 2873 ms.
(base) (.venv) PS C:\Users\aless\Network Security\netw
orkSecurityProject3> python src/client.py
--- CHAT STARTED ---
Identity (A/B): A
[SUCCESS] Connected as client-alice.
[Server] client-bob has joined.
[client-bob]: a
[client-bob]: a
[client-bob]: a
[client-bob]: a
[client-bob]: a
[client-bob]: a
[client-bob]: a
[client-bob]: a
[client-bob]: a
[client-bob]: a
```

```
orkSecurityProject3> python src/client.py
--- CHAT STARTED ---
Identity (A/B): B
[SUCCESS] Connected as client-bob.
[Server] Online users: ['client-alice']
You: a
You: a
You: a
You: a
You: a
You: a
You: a
You: a
You: a
You: a
You: a
You:
[!] Disconnected.
```



tcp.port == 8443

No.	Time	Source	Destination	Protocol	Lengtl	Info
4253	74.156217	127.0.0.1	127.0.0.1	TCP	44	8443 → 24075 [ACK] Seq=5823 Ack=31429 Win=34048 Len=0
4254	74.156634	127.0.0.1	127.0.0.1	TLSv1.3	4162	Application Data
4255	74.156688	127.0.0.1	127.0.0.1	TCP	44	24073 → 8443 [ACK] Seq=2605 Ack=34632 Win=65280 Len=0
4256	74.389309	127.0.0.1	127.0.0.1	TLSv1.3	4162	Application Data
4257	74.389369	127.0.0.1	127.0.0.1	TCP	44	8443 → 24075 [ACK] Seq=5823 Ack=35547 Win=65280 Len=0
4258	74.389844	127.0.0.1	127.0.0.1	TLSv1.3	4162	Application Data
4259	74.389886	127.0.0.1	127.0.0.1	TCP	44	24073 → 8443 [ACK] Seq=2605 Ack=38750 Win=61184 Len=0
4298	74.610274	127.0.0.1	127.0.0.1	TLSv1.3	4162	Application Data
4299	74.610321	127.0.0.1	127.0.0.1	TCP	44	8443 → 24075 [ACK] Seq=5823 Ack=39665 Win=61184 Len=0
4300	74.610731	127.0.0.1	127.0.0.1	TLSv1.3	4162	Application Data
4301	74.610771	127.0.0.1	127.0.0.1	TCP	44	24073 → 8443 [ACK] Seq=2605 Ack=42868 Win=57088 Len=0
4302	74.858131	127.0.0.1	127.0.0.1	TLSv1.3	4162	Application Data
4303	74.858191	127.0.0.1	127.0.0.1	TCP	44	8443 → 24075 [ACK] Seq=5823 Ack=43783 Win=57088 Len=0
4304	74.858948	127.0.0.1	127.0.0.1	TLSv1.3	4162	Application Data
4305	74.859002	127.0.0.1	127.0.0.1	TCP	44	24073 → 8443 [ACK] Seq=2605 Ack=46986 Win=52992 Len=0
4306	75.081905	127.0.0.1	127.0.0.1	TLSv1.3	4162	Application Data
4307	75.081969	127.0.0.1	127.0.0.1	TCP	44	8443 → 24075 [ACK] Seq=5823 Ack=47901 Win=52992 Len=0
4308	75.083667	127.0.0.1	127.0.0.1	TCP	44	8443 → 24075 [FIN, ACK] Seq=5823 Ack=47901 Win=52992 Len=0
4309	75.083713	127.0.0.1	127.0.0.1	TCP	44	24075 → 8443 [ACK] Seq=47901 Ack=5824 Win=59648 Len=0
4314	75.299981	127.0.0.1	127.0.0.1	TLSv1.3	4162	Application Data
4315	75.300073	127.0.0.1	127.0.0.1	TCP	44	8443 → 24075 [RST, ACK] Seq=5824 Ack=52019 Win=0 Len=0

7. Conclusion

The project addresses the full spectrum of the CIA triad (Confidentiality, Integrity, Availability). By integrating a custom Private PKI with Mutual TLS, the system ensures a trusted transport layer that is resistant to unauthorized access and surveillance. The application-layer logic provides a robust defense against DoS attacks and identity spoofing, while the End-to-End Encryption protocol guarantees that the server remains a zero-knowledge relay. The inclusion of advanced features such as AAD-based metadata integrity, anti-replay caching, and traffic padding elevates the security posture of the application. To conclude, the current architecture represents a secure, resilient, and audit-proof messaging solution capable of withstanding sophisticated network attacks.