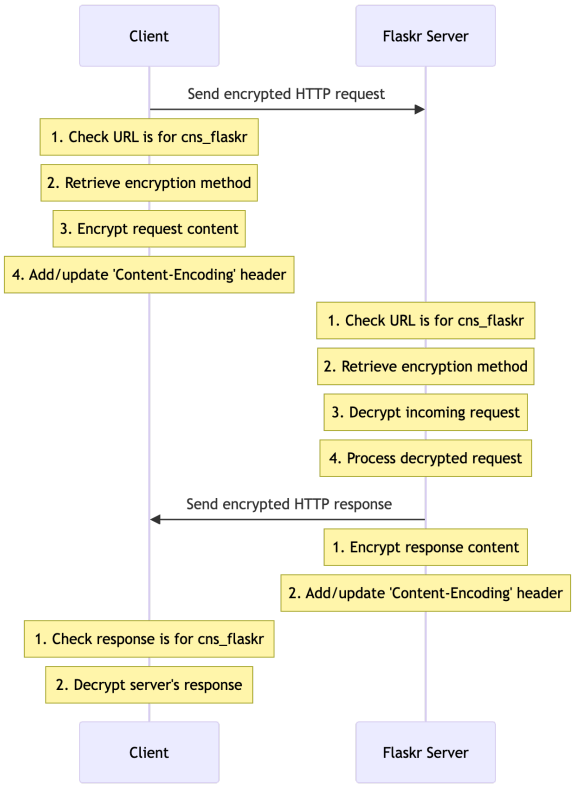
Implementation

AES: Used an existing implementation from the Hashlib library.
Salsa20: Using specifications from slides given in class and pseudo-code found on Wikipedia.

How we ensure confidentiality

1. The content is encrypted using Salsa20 or AES.
2. Use of a preshared key ensures only parties with this key can decrypt the content.
3. We only process traffic for http://cns_flaskr/
4. We update (or insert) Content Encoding Header, so the receiving party can decrypt the content.
5. The receiving party is the Flaskr server, which is equipped to decrypt the content following the instructions above.
6. And vice versa (other direction)



MitM Attack Overview

An attacker intercepts communication between two parties, relaying and potentially altering the conversation.

1. Interception: Intercept the communication between the client and the Flaskr server without detection. This can be done using the cns_student proxy.
2. Key Reuse Attack:

    a. Observation: Note if the client sends multiple messages using the same encryption key and nonce.
    b. Exploit: Use XOR on two ciphertexts encrypted with the same key to reveal the XOR of their plaintexts.
3. Dictionary Attack:
    a. Observation: Identify if the client uses a simple encryption key.
    b. Dictionary Preparation: Prepare a dictionary of commonly used words.
    c. Brute-force: Try decrypting the message using each word in the dictionary as a potential key. Since the content is in key=value format, look for deciphered content that matches this pattern.
    d. Exploit: Once the right key is identified, modify or re-encrypt the client's future messages.
4. Traffic Manipulation: Using the identified vulnerabilities, manipulate the data packets or alter the content of messages.
5. Re-encryption and Forwarding: Encrypting the altered message and sending it to the intended recipient.
6. Defense: Always use unique encryption keys, pick strong keys, and monitor traffic for anomalies to combat MiTM threats.

Implementation

MAC:

H(K+N+C) where H is the SHA1 hashing algorithm, K is a key and N is a random nonce. We used the SHA1 given in class. We added padding according to the specs given in the slides. It first converts the message to its binary representation. A '1' bit is appended, followed by enough '0' bits to ensure the length is 448 mod 512. Finally, a 64-bit binary representation of the original message length is appended to the end.

HMAC:

This HMAC implementation uses the SHA-512 hashing algorithm. It creates an HMAC by first XORing the key with inner and outer padding values, then hashing the concatenated results of the inner-padded key, nonce, and content, and finally hashing the concatenated results of the outer-padded key and the previous hash. If the key is longer than the block size, it's hashed; if shorter, it's padded. Specs were given in

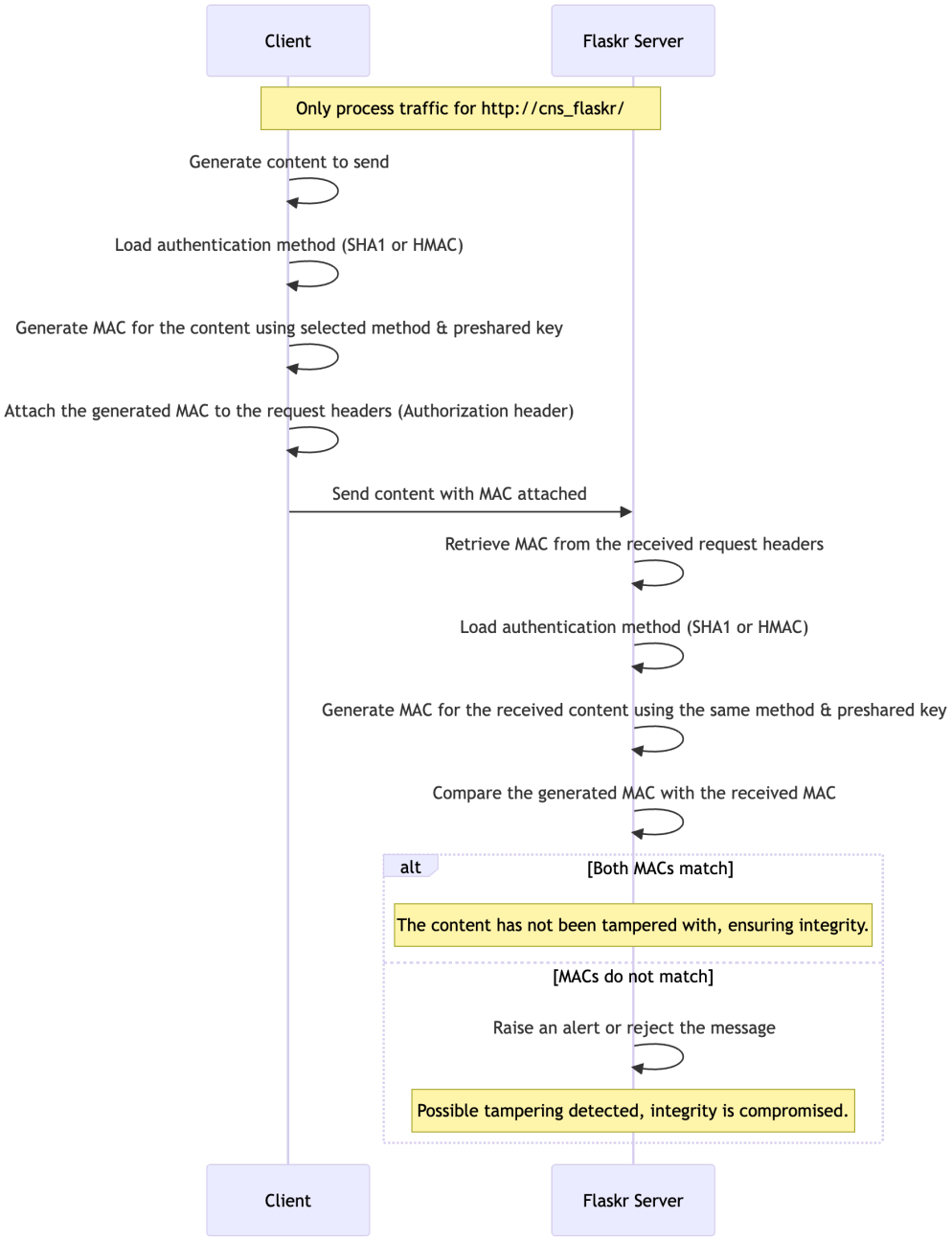
class. Both the key and nonce are checked on length and if necessary either shortened or lengthened so no problems occur when the hashing is done.

HTTP-Traffic:

Authorization needs to be added to the existing HTTP traffic to make use of the implemented MAC and HMAC as specified in the config files. in both the client and flaskr request and response we will authenticate after decrypting/encrypting. The authentication is pretty simple, first the necessary request headers are made as specified in the given mac_spec.md. Then the content we need to send is created and we pass it along to the MAC/HMAC generation to create our mac. This same mac will be checked in the response authorization to see if the content has been tampered with. There are extra checks done besides the mac like checking the timestamps of both the request and response authentication to see if it has been too long in between them.

Integrity of Traffic

1. The content is authenticated using MAC with either SHA1 or HMAC.
2. Use of a preshared key ensures that only parties with this key can validate the content's integrity.
3. We only process traffic for http://cns_flaskr/.
4. The computed MAC is attached to the request headers (Authorization header) to validate the data's integrity.
5. Timestamps ("X-Authorization-Timestamp") are included in the headers to prevent replay attacks and ensure that the message is recent.
6. The Flaskr server verifies the received MAC against its own computed MAC to ensure the content hasn't been tampered with.
7. And vice versa (other direction)



Implementation

- RSA Key Generation
- RSA Encryption
- RSA Decryption

RSA Key generation

RSA key generation involves the following steps:

1. **Prime Number Generation:** The generate_prime_nr function generates a prime number with a specified number of bits. it uses the is_prime function, which employs the Rabin-Miller primality test to check if a number is prime. This test is probabilistic and is more efficient than deterministic tests, especially for large numbers.
2. **Key Generation:** The generate_keys function generates the RSA keys. it first generates two distinct prime numbers, p and q, each having half the bit size of the desired RSA modulus. The RSA modulus n is then computed as the product of p and q. The public exponent e is chosen such that it is relatively prime to (p-1)*(q-1), and the private exponent d is computed as the multiplicative inverse of e modulo (p-1)*(q-1).

The benefits of using these methods are:

- **Efficiency:** The Rabin-Miller primality test is an efficient algorithm for checking the primality of a number, especially for large numbers. it is probabilistic, meaning it may declare a composite number to be a prime, but the probability of such an error can be made arbitrarily small by increasing the number of iterations in the test.
- **Security:** The RSA algorithm is secure as long as the prime numbers are kept secret. By generating large prime numbers (1024, 2048, or 4096 bits), the security of the RSA keys is ensured.
- **Versatility:** The code allows for the generation of RSA keys of different sizes (1024, 2048, or 4096 bits), providing flexibility in terms of the level of security and computational efficiency.

RSA Encryption/ Decryption

RSA encryption is a public-key encryption technology developed by RSA Data Security. The RSA algorithm is based on the difficulty in factoring large numbers. RSA-OAEP is a form of encryption that combines the RSA algorithm with a padding scheme. Here's how it works:

1. **Masking**: The masking step in RSA-OAEP involves the use of a mask generation function (MGF) to generate a pseudo-random string of bits. This string is then combined with the message block (MB) and the seed using the XOR operation. The purpose of masking is to add an element of randomness to the encryption process. This randomness converts a deterministic encryption scheme (like traditional RSA) into a probabilistic scheme, enhancing its security. Masking also prevents an adversary from being able to

recover any portion of the plaintext without being able to invert the trapdoor one-way permutation.

2. **Hashing**: The hashing step in RSA-OAEP involves the use of a cryptographic hash function. This function takes an input (or 'message') and returns a fixed-size string of bytes. The output (the 'digest') is unique to each unique input. Even a small change in the input will produce such a drastic change in the output that the new digest will appear uncorrelated with the old digest. In RSA-OAEP, a hash of an optional label L is included in the padded message. This label can be used to authenticate data without requiring encryption. Hashing is used to ensure the integrity of this label. If the label is changed in any way, the hash will also change, alerting the recipient to the alteration.

3. **OAEP Encoding**: The OAEP encoding process uses two random oracles, G and H, to process the plaintext prior to asymmetric encryption. This processing is proven to result in a combined scheme which is semantically secure under chosen plaintext attack (IND-CPA). When implemented with certain trapdoor permutations (e.g., RSA), OAEP is also proven to be secure against chosen ciphertext attack.

In summary, the steps of masking and hashing in RSA-OAEP are crucial for enhancing the security of the RSA encryption process. They ensure the confidentiality, integrity, and authenticity of the encrypted data, making RSA-OAEP a robust and secure encryption scheme.

Decryption: The recipient uses their private key to transform the ciphertext back into the original plaintext message. This is done by raising the ciphertext to the power of the private exponent, dividing by the modulus, and converting the resulting residue back into a message.

The benefits of using RSA-OAEP are as follows:

- Randomness: OAEP adds an element of randomness to the encryption process. This can convert a deterministic encryption scheme (like traditional RSA) into a probabilistic scheme, enhancing its security.
- Semantic Security: OAEP is proven to be semantically secure under chosen plaintext attack (IND-CPA). This means that an attacker cannot gain any information about the plaintext even if they can choose the plaintext to be encrypted.

- Resistance to Partial Decryption: OAEP prevents partial decryption of ciphertexts by ensuring that an adversary cannot recover any portion of the plaintext without being able to invert the trapdoor one-way permutation.

In this project we use a nonce as a seed for the encryption. This is a great practice as it adds an additional layer of security. The nonce ensures that even if the same message is encrypted twice, the resulting ciphertexts will be different, preventing replay attacks.

Implementation

- Creating Certificates(x509)
- Key Exchange

Creating certificates

Sure, I'd be happy to explain the process of creating OpenSSL x509 certificates for your project. Here's a detailed explanation:

1. **Creating a Self-Signed Certificate**: The command `openssl req -x509 -new -nodes -key {PrivateKey} -sha1 -days 365 -out certificate.pem` is used to create a new self-signed x509 certificate. The `-x509` option specifies that a self-signed certificate is to be created instead of a certificate request. The `-new` option indicates that a new certificate request should be created. The `-nodes` option means that the private key will not be encrypted. The `-key` option specifies the file to read the private key from. The `-sha1` option specifies the hash algorithm used to sign the certificate request. The `-days 365` option specifies that the certificate will be valid for 365 days. The `-out` option specifies the output filename.
2. **Creating a Certificate Signing Request (CSR)**: The command `openssl req -new -key {PrivateKey} -out {Filename} -sha1 -out cns_flaskr.csr` is used to create a new CSR. The `-new` option indicates that a new CSR should be created. The `-key` option specifies the file to read the private key from. The `-out` option specifies the output filename. The `-sha1` option specifies the hash algorithm used to sign the certificate request.
3. **Signing the CSR**: The command `openssl x509 -req -in {certificate_request} -CA {ca_certificate} -CAkey {ca_private_key} -out {certificate} -days 365 -sha1` is used to sign the CSR with a CA certificate. The `-req` option specifies that a certificate request should be read in. The `-in` option specifies the input filename. The `-CA` option specifies the CA certificate to be used for signing. The `-CAkey` option specifies the CA private key to be used for signing. The `-out` option specifies the output filename.

The `-days 365` option specifies that the certificate will be valid for 365 days. The `-sha1` option specifies the hash algorithm used to sign the certificate request.

4. **Saving the Certificate**: The command `openssl {certificateType} -in {filename} -text > {output_filename}` is used to save the certificate. The `-in` option specifies the input filename. The `-text` option specifies that the certificate should be printed in text form.

The certificates created by these commands enable secure, encrypted communication between the client and server proxies. They ensure the authenticity of the server to the client and can also be used to authenticate the client to the server.

```
Certificate:

   Data:

      Version: 3 (0x2)

      Serial Number:

          21:a1:74:ad:30:87:70:45:fc:4f:29:b1:b9:05:31:6d:53:4b:c1:99

      Signature Algorithm: sha1WithRSAEncryption

      Issuer: C = BE, ST = Antwerpen, L = Antwerpen, O = UAntwerpen, OU = Student-CNS, CN = CA-CNS,
emailAddress = ca-cns@uantwerpen.be

      Validity

          Not Before: Dec  5 15:01:25 2023 GMT

          Not After : Dec  4 15:01:25 2024 GMT

      Subject: C = BE, ST = Antwerpen, L = Antwerpen, O = UAntwerpen, OU = Student-CNS, CN = CA-CNS,
emailAddress = ca-cns@uantwerpen.be

      Subject Public Key Info:

          Public Key Algorithm: rsaEncryption

              Public-Key: (2047 bit)

              Modulus:

                  53:80:89:9e:73:8c:78:49:da:8c:7d:0b:7d:0f:d7:

                  04:b5:ac:68:63:ad:3c:bd:d5:af:25:ae:72:82:53:

                  8b:df:52:dd:93:d0:92:48:cc:40:16:34:53:27:cf:

                  96:63:36:10:4b:9e:c9:3e:70:e3:15:6e:38:df:1b:

                  3b:21:18:ce:22:b5:54:fc:a7:11:bb:8b:ef:f9:51:

                  4b:68:fd:06:ef:c3:64:ae:e6:da:3a:bb:a0:d0:2f:

                  64:de:1b:0a:72:c2:45:1c:54:1d:da:cc:26:50:7a:

                  29:0e:df:da:64:59:9a:24:ae:c2:8d:e9:35:36:c9:

                  5f:cd:d9:b3:42:81:86:c5:d5:9e:80:28:ee:a0:61:
```

```
                02:95:50:f7:bd:96:ee:b8:30:9b:7b:22:a7:fc:02:
                0d:fd:69:ad:60:e3:86:71:6b:0c:46:d4:99:c5:38:
                58:12:0d:c3:c4:c6:d8:78:87:e8:c7:1a:2a:d7:e0:
                8b:97:82:8a:69:4d:98:2e:9d:52:b8:9f:23:eb:d8:
                b4:d7:fc:d5:83:d9:61:65:2e:9f:c6:20:a9:3e:90:
                7b:d3:80:9c:32:ff:d5:59:3c:82:a3:85:0b:bf:86:
                d6:a3:20:8d:98:99:d4:66:41:aa:6b:fc:82:30:5a:
                57:95:16:bf:17:f8:13:1c:88:92:7a:af:14:a9:99:
                77
            Exponent:
                1a:03:8f:21:f8:b1:0e:40:23:17:ed:40:aa:8d:6c:
                55:c1:14:bc:40:eb:a6:75:f1:bc:e0:3f:98:a0:ec:
                4c:3b:46:f0:0f:1d:1a:e6:dc:0c:20:70:f2:67:3e:
                0e:df:57:f8:76:90:9a:b7:fb:1c:49:2b:5b:54:61:
                71:ed:5e:e9:e3:5f:94:c6:76:06:d6:82:b2:f9:fd:
                d4:0d:79:48:4f:91:ad:22:21:59:79:95:d2:aa:1f:
                2b:51:bb:5a:08:ae:82:29:1b:89:8b:d9:95:82:57:
                fb:ab:e1:a2:94:23:85:54:7d:22:e9:16:ff:d2:99:
                51:8c:55:0f:8e:1e:c0:78:12:37:fd:13:ec:f6:9a:
                ff:05:71:40:bb:69:61:f3:c0:02:d9:0e:9d:28:b0:
                b6:51:8e:43:c4:eb:c1:ba:30:75:fd:1a:3f:9e:22:
                30:70:b6:36:0d:4f:f1:37:35:4a:e2:8f:dc:ec:59:
                9e:a3:92:3a:6c:e7:72:45:85:f7:68:cd:36:2c:6c:
                ec:6b:8d:a2:ba:e4:49:dd:f9:37:bd:dc:5c:e0:de:
                10:02:14:7f:75:d6:ce:cb:d1:34:66:39:d4:62:b1:
                c5:0c:bd:17:6f:5f:ea:43:4c:9a:46:b3:de:13:19:
                5c:15:fa:23:65:4a:31:67:ba:0b:8a:5e:95:09:6e:
                57
    X509v3 extensions:
        X509v3 Subject Key Identifier:
            88:65:3F:38:36:04:B9:97:78:37:E4:8B:5E:50:A4:B4:61:60:81:62
        X509v3 Authority Key Identifier:
            88:65:3F:38:36:04:B9:97:78:37:E4:8B:5E:50:A4:B4:61:60:81:62
```

```
        X509v3 Basic Constraints: critical

            CA:TRUE

Signature Algorithm: sha1WithRSAEncryption

Signature Value:

    24:24:c1:47:88:79:00:24:ee:88:b6:57:b3:b4:94:a9:00:97:

    b9:06:2c:73:61:c9:e7:0d:24:cd:07:c7:66:f4:90:23:e5:af:

    5e:9e:f6:94:8d:e5:fb:ea:88:90:d2:90:66:49:25:00:de:52:

    f7:e0:be:c9:ec:fb:9b:b6:99:22:85:93:81:dc:93:ea:fc:7e:

    1d:9e:de:9e:44:7c:81:ea:39:15:88:93:e3:d3:c3:ae:c9:af:

    e5:69:2a:26:1b:71:6c:ad:31:8c:df:fd:fd:ed:86:ba:c5:b2:

    ac:73:6d:18:04:23:a7:bb:6a:c3:ce:93:ce:9c:3e:f3:24:f2:

    65:cc:de:c6:6a:41:8c:8c:de:03:20:7c:1b:74:27:db:00:5d:

    81:b7:54:40:86:4e:42:b0:c8:48:54:98:a3:86:ec:21:44:c7:

    fe:c4:29:5b:40:bd:76:de:c6:28:09:5d:de:f4:10:98:f7:c1:

    50:95:52:e4:c8:40:fd:f6:c5:95:cc:bb:2d:8d:97:ea:7a:3e:

    39:92:87:7f:7c:c8:ff:f2:4b:8e:92:e9:2f:3c:a6:f5:9f:13:

    4f:e4:60:a5:f9:84:fa:72:8f:cc:6c:c1:fd:8b:64:4e:d2:6a:

    a9:07:1f:8e:1e:3b:e9:99:18:91:85:3f:dd:ed:ae:ff:31:87:

    0e:4f:7b:26
```

This is a example of what a certificate look like and it contains the following:

1. **Version**: This field indicates the version of the X.509 standard that the certificate uses. In this case, it's version 3.
2. **Serial Number**: This is a unique number assigned by the certificate authority (CA) when the certificate is issued. It's used to uniquely identify each certificate.
3. **Signature Algorithm**: This field specifies the algorithm used to sign the certificate. In this case, it's `sha1WithRSAEncryption`, which means SHA-1 with RSA encryption is used.
4. **Issuer**: This field contains the distinguished name (DN) of the entity that issued the certificate. It includes the country ©, state or province (ST), locality or city (L), organization (O), organizational unit (OU), common name (CN), and email address.
5. **Validity**: This field specifies the period during which the certificate is valid. It includes the `Not Before` and `Not After` dates.
6. **Subject**: This field contains the DN of the entity that the certificate identifies. It includes the same types of information as the Issuer field.
7. **Subject Public Key Info**: This field contains the public key of the entity identified by the Subject field. It includes the public key algorithm and the public key itself.

8. **X509v3 extensions**: These are additional fields that provide extra information about the certificate. They include the `X509v3 Subject Key Identifier`, which provides a unique identifier for the subject's public key, the `X509v3 Authority Key Identifier`, which provides a unique identifier for the issuer's public key, and the `X509v3 Basic Constraints`, which indicates whether the subject of the certificate is a CA.
9. **Signature Algorithm**: This field specifies the algorithm used to sign the certificate. It's the same as the Signature Algorithm field mentioned earlier.
10. **Signature Value**: This field contains the digital signature of the certificate. The digital signature is created by taking a hash of the certificate and encrypting it with the issuer's private key. The signature can be verified by decrypting it with the issuer's public key and comparing it with a newly computed hash of the certificate.

Key exchange

Our project involves establishing a secure communication protocol between a client and a server, specifically between `Cns_client` and `Cns_flaskr`. This is achieved through a process known as a handshake, which is used to exchange session keys that secure the communication. The keys are frequently changed to make it harder for an attacker to figure out the session key and eavesdrop on the communication.

Client Proxy Code:

The client proxy code begins by retrieving the paths for the certificate, CA certificate, and public key. It then performs garbage collection on the session path to remove any expired sessions. If a session path exists, it checks for active sessions.

If no active session is found, the client requests a certificate from the server and verifies the server's certificate using the CA's public key. If the certificate is valid, the client generates two 100-byte random session keys (an encryption key and an authentication key), combines them into a session key, and sends it to the server.

The server responds with the session ID and end timestamp. The client then creates a session file with the session ID, end timestamp, and the encryption and authentication keys.

If an active session is found, the client retrieves the session. If the session is not found, the client sends an HTTP response indicating that no active session is present. If the session is found, the client retrieves the encryption and authentication keys from the session.

Flaskr Proxy (Server) Code:

The server proxy code begins by retrieving the paths for the certificate and private key. It then performs garbage collection on the path to remove any expired sessions.

If the client sends a 'client_ack' request, the server decrypts the encrypted session key sent by the client. If the decryption is successful, the server separates the encryption key and the authentication key from the decrypted session key.

The server then creates a session file with a new session ID, an end timestamp (current time + 60 seconds), and the encryption and authentication keys. The server sends an acknowledgement response to the client with the session ID and end timestamp.

If the client sends a request other than ‘client_ack’, the server checks for active sessions. If an active session is found, the server retrieves the session. If the session is not found, the server sends an HTTP response indicating that it could not find the session. If the session is found but it’s due to expire in less than 10 seconds, the server sends an HTTP response indicating that the session has expired.

This code ensures that a secure session is established between the client and the server, and that session keys are frequently changed to enhance security. It also handles various error conditions, such as an invalid certificate, a failed decryption, or an expired session. This makes the communication between the client and the server robust and secure.

However, the handshake process you’re implementing is a simplified version and is susceptible to certain types of attacks. One such attack is a replay attack, where an attacker intercepts the session keys and reuses them in the future. To prevent this, you could implement a nonce in the handshake process. A nonce is a random number that is used only once, ensuring that even if the session keys are intercepted, they cannot be reused.

Both the client and the server save their active sessions in JSON files in a new sessions directory. A session file includes the following fields:

- `encr key`: The session encryption key used with AES.
- `auth key`: The session authentication key used with HMAC.
- `end`: The Unix timestamp that defines when the session keys become invalid.
- `id`: The ID number of the session, used to indicate which session is being used.

This process ensures that the communication between the client and the server is secure, even in the face of potential eavesdropping attacks. By frequently changing the session keys, it becomes much harder for an attacker to figure out the session key and listen in on the communication. This is especially important given the computational power available today, which could potentially be used to crack encryption keys. By implementing this handshake process, we’re taking a significant step towards securing the communication in our project.