

CYBR372 – Assignment 2

greenthom – 300536064

Part 1: TLS Client and Server Implementation Using Java Sockets and SSL APIs

Secure communication between a client and a server is established using TLS (Transport Layer Security). Both parties are configured using Java Sockets and JSSE (Java Secure Socket Extension) APIs to ensure secure, encrypted connections. A custom Certificate Authority (CA) is used to authenticate the server, ensuring trust in the communication process.

The first step is becoming 'our' own CA. Requires generating a root CA certificate and a private key. This root CA is then used to sign the server's certificate. For CA key and certificate generation, I generate a private key using the following command:

```
openssl genrsa -out ca.key.pem 2048
```

Next step is generating the CA's self-signed certificate using that private key:

```
openssl req -x509 -new -nodes -key ca.key.pem -sha256 -days 1024 -out ca.crt.pem
```

The root CA certificate will later be used by the client to verify the server's identity. The server needs a certificate signed by the custom CA. First, we generate a private key for the server:

```
openssl genrsa -out server.key.pem 2048
```

A CSR (certificate signing request) is created using the server's private key:

```
openssl req -new -key server.key.pem -out server.csr
```

The server's certificate is signed using the CA's private key:

```
openssl x509 -req -in server.csr -CA ca.crt.pem -CAkey ca.key.pem -CAcreateserial -out server.crt.pem -days 500 -sha256
```

This step binds the server's identity to the certificate, and the client will later validate the server against this certificate. In Java, both the server and client need keystores to manage certificates. The server's private key and certificate are converted into the PKCS 12 format and then stored in a Java KeyStore. Converting the server certificate and key into PKCS12 format:

```
openssl pkcs12 -export -inkey server.key.pem -in server.crt.pem -name "server" -out server.p12 -CAfile ca.crt.pem -caname root
```

Creating the server keystore:

```
keytool -importkeystore -srckeystore server.p12 -srcstoretype PKCS12 -destkeystore server_keystore.jks -deststoretype JKS
```

The CA certificate is imported into the client's trust store to ensure the client trusts the server's certificate:

```
keytool -import -file ca.crt.der -alias ca -keystore ca_truststore.jks
```

Once the certificates are set up, the client authenticates the server by verifying the server's certificate against the trusted CA certificate in the client's trust store. In clientTLS.java, TrustManagerFactory is initialized using the client's trust store, which contains the CA certificate:

```
TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");  
Tmf.init(ks);
```

The SSL Context is initialized with the TrustManager, which handles the server's certificate verification:

```
SSLContext sc = SSLContext.getInstance("TLSv1.2");  
sc.init(null, tmf.getTrustManagers(), null);
```

When the client establishes a connection to the server, the server presents its certificate. The client verifies this certificate by checking if it is signed by the trusted CA in the trust store:

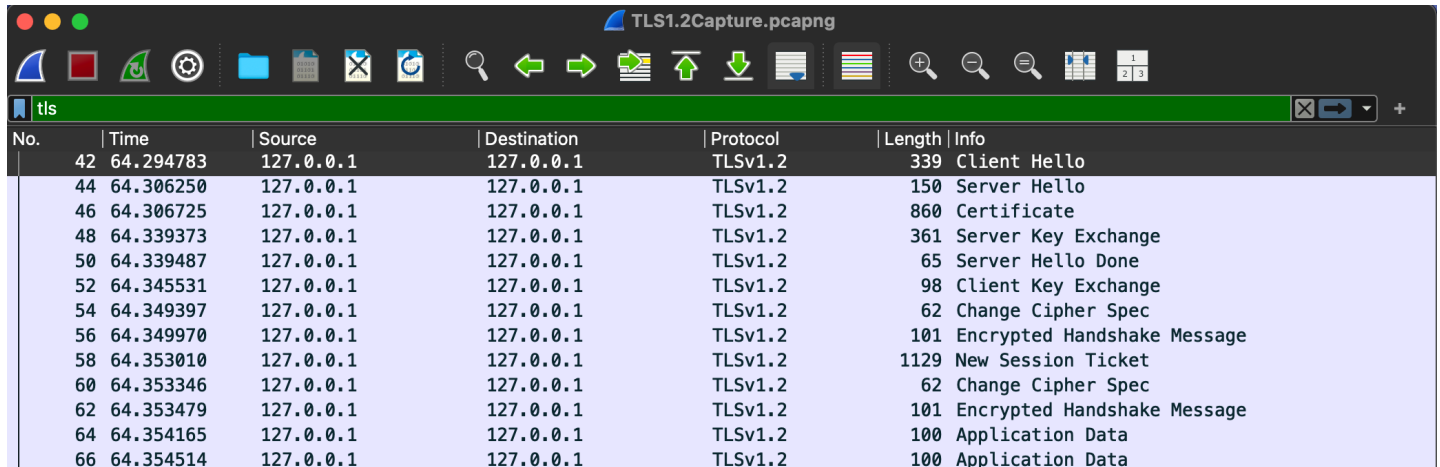
```
SSLSocket sSocket = (SSLSocket) ssf.createSocket("localhost", 8443);
```

If the server's certificate is invalid, expired, or not signed by the CA, the client will reject the connection and throw an SSLHandshakeException. The client ensures secure communication by authenticating the server using the CA's trusted certificate, ensuring that the server is genuine before any communication occurs.

Part 2: Capture and Break-down TLS Handshake with Wireshark

Based on Wireshark Screenshot from TLS 1.2 capture:

TLS 1.2 handshake establishes a secure communication channel between the client and server. Used the server.java and client.java code established in Part 1 of this assignment to carry out this server and client communication using 1.2.



No.	Time	Source	Destination	Protocol	Length	Info
42	64.294783	127.0.0.1	127.0.0.1	TLSv1.2	339	Client Hello
44	64.306250	127.0.0.1	127.0.0.1	TLSv1.2	150	Server Hello
46	64.306725	127.0.0.1	127.0.0.1	TLSv1.2	860	Certificate
48	64.339373	127.0.0.1	127.0.0.1	TLSv1.2	361	Server Key Exchange
50	64.339487	127.0.0.1	127.0.0.1	TLSv1.2	65	Server Hello Done
52	64.345531	127.0.0.1	127.0.0.1	TLSv1.2	98	Client Key Exchange
54	64.349397	127.0.0.1	127.0.0.1	TLSv1.2	62	Change Cipher Spec
56	64.349970	127.0.0.1	127.0.0.1	TLSv1.2	101	Encrypted Handshake Message
58	64.353010	127.0.0.1	127.0.0.1	TLSv1.2	1129	New Session Ticket
60	64.353346	127.0.0.1	127.0.0.1	TLSv1.2	62	Change Cipher Spec
62	64.353479	127.0.0.1	127.0.0.1	TLSv1.2	101	Encrypted Handshake Message
64	64.354165	127.0.0.1	127.0.0.1	TLSv1.2	100	Application Data
66	64.354514	127.0.0.1	127.0.0.1	TLSv1.2	100	Application Data

First, the client sends a ClientHello message to the server. We can observe this in Wireshark:

no: 42, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, Info: Client Hello

Based on the information in RFC 5246, the client starts the handshake by sending the ClientHello message to the server. This message is made up of these components:

- client_version: the highest version of TLS the client supports. Is using TLSv1.2 based on the implementation of the clientTLS.java script.

Version: TLS 1.2 (0x0303)

- random: random number generated by the server. Used in the key generation process.

23985b6a14315ea3bab48b65d74d7ca3c7cb5ee3f4a56f3d049e55a2a80e7422

- session_id: If the client wishes to resume a session, it includes the ID of the session. At this point, it is 0 due to no session to resume.
- cipher_suites: A list of cryptographic algorithms supported by the client ordered by preference. Sent 34 suites in total in the capture:

Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0x009f)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x009f)

Cipher Suite: TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xccaa)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_GCM_SHA384 (0x00a3)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_GCM_SHA256 (0x00a2)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0x006b)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA256 (0x006a)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 (0x0067)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA256 (0x0040)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA256 (0x003d)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)

- supported_groups: indicates the supported groups for key exchange.

Supported Group: x25519
Supported Group: secp256r1 (0x0017)
Supported Group: secp384r1 (0x0018)
Supported Group: secp521r1 (0x0019)
Supported Group: x448 (0x001e)
Supported Group: ffdhe2048 (0x0100)
Supported Group: ffdhe3072 (0x0101)
Supported Group: ffdhe4096 (0x0102)
Supported Group: ffdhe6144 (0x0103)
Supported Group: ffdhe8192 (0x0104)

- Signature Hash Algorithms: list of supported signature algorithms for verifying the digital signatures.

Signature Algorithm: ecdsa_secp256r1_sha256 (0x0403)
Signature Algorithm: ecdsa_secp384r1_sha384 (0x0503)
Signature Algorithm: ecdsa_secp521r1_sha512 (0x0603)
Signature Algorithm: ed25519 (0x0807)

Signature Algorithm: ed448 (0x0808)
Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
Signature Algorithm: rsa_pss_rsae_sha384 (0x0805)
Signature Algorithm: rsa_pss_rsae_sha512 (0x0806)
Signature Algorithm: rsa_pss_rsae_sha256 (0x0809)
Signature Algorithm: rsa_pss_rsae_sha384 (0x080a)
Signature Algorithm: rsa_pss_rsae_sha512 (0x080b)
Signature Algorithm: rsa_pkcs1_sha256 (0x0401)
Signature Algorithm: rsa_pkcs1_sha384 (0x0501)
Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
Signature Algorithm: SHA256 DSA (0x0402)
Signature Algorithm: SHA224 ECDSA (0x0303)
Signature Algorithm: SHA224 RSA (0x0301)
Signature Algorithm: SHA224 DSA (0x0302)
Signature Algorithm: ecdsa_sha1 (0x0203)
Signature Algorithm: rsa_pkcs1_sha1 (0x0201)
Signature Algorithm: SHA1 DSA (0x0202)

- supported_versions: informs the server of all TLS versions supported by the client

Supported Version: TLS 1.2 (0x0303)

After sending the ClientHello, the client waits for a response from the server. The next step that can be observed is the ServerHello message from the server to the client. We can observe this in the Wireshark screenshot:

no: 44, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: ServerHello

Based on information in RFC 5246 -> The server responds with the ServerHello message if it can support one of the client's cipher suites and TLS version. This message is made up of these components:

- server_version: TLS version selected by the server (typically the highest supported version common to both parties), but in this case, using TLSv1.2 based on the implementation of the serverTLS.java script.

Version: TLS 1.2 (0x0303)

- random: random number generated by the server, used in the key generation process.

64f50184297c4265872d522bed2386bc50d29ae2b5491b6a444f574e47524401

- session_id: if the client requests session resumption and the server agrees, it echoes the session ID. If not a new session ID is generated or left blank.

421757f4cffc7d75e4db790a0ffb4b318673270c794b64ec1836959d091dab70

- cipher_suite: cryptographic algorithm selected by the server from the client's suite list.

TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)

The server selects security parameters based on the client's preferences (and what it supports) and then prepares for authentication. The next step that can be observed is 'Certificate', which can be seen in the Wireshark screenshot:

no: 46, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: certificate

Based on information from RFC 5246, the server sends its digital certificate, issued by a 'trusted' Certificate Authority (CA), to authenticate itself. This certificate allows the client to verify the server's identity using the server's public key and certificate chain. The certificate is signed using the server's private key (the client will use the corresponding public key to verify authenticity). The certificate is encrypted when sent to the client (using SHA256 with RSA encryption).

sha256WithRSAEncryption

Also has a validity time associated with the certificate indicating certificate validity not before and not after times.

validity

notBefore: utcTime (0)

utcTime: 2024-10-06 23:05:38 (UTC)

notAfter: utcTime (0)

utcTime: 2026-02-18 23:05:38 (UTC)

The next step that can be observed is the Server Key Exchange, which can be seen in the Wireshark screenshot:

no: 48, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: Server Key Exchange

Based on information from RFC 5246, if the selected cipher suite requires it (for example, with Diffie-Hellman or Elliptic-curve algorithms), the server sends a ServerKeyExchange message containing its Diffie-Hellman public key to the client so the client can compute the pre-master secret. This allows both parties (client and server) to compute a shared secret. It uses the Elliptic Curve Diffie Hellman cipher suite which requires key exchange and contains the server public key of length 32. In this case, the server is using the x25519 (Curve25519) elliptic curve to exchange keys with the client. This was provided by the client hello in supported signature algorithms to verify digital signatures.

Curve Type: named_curve (0x03)

Named Curve: x25519 (0x001d)

Pubkey Length: 32

The signature hash algorithm used is rsa_pss_rsae_sha256 (0x0804), which provides strong security using RSA-PSS for the signature and SHA-256 for hashing. This hashes the key exchange that is done using the Curve25519 providing integrity and authenticity on the key exchange. The server chooses this signature hashed algorithm based on the provided supported signature hashing algorithms from the client. The next step that can be observed is the ServerHello Done message, which can be seen in Wireshark:

no: 50, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: Server Hello Done

Based on information from RFC 5246, the server signals that it has finished sending its portion of the handshake with the Server Hello Done message and waits for the client's response. The next step is the Client Key Exchange which can be observed in the Wireshark screenshot:

no: 52, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: Client Key Exchange

Based on information from RFC 5246, the client responds by sending the ClientKeyExchange message. This 'handshake' message contains key exchange information (pre-master secret) that allows both the client and server to generate session keys (master secret). In this case, it is using the Elliptic Curve Diffie Hellman cipher suite which requires key exchange, we can see that based on the capture the client sends its public key of length 32. The next step that can be observed is 'Change Cipher Spec' which occurs client side. This can be seen in the Wireshark screenshot:

no: 54, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: Change Cipher Spec

Based on information from RFC 5246, the client sends the ChangeCipherSpec message to indicate that it will now use the agreed-upon encryption keys for further communication using the negotiated cipher suite. The next step that can be observed is 'Encrypted Handshake Message' which occurs client side. This can be seen in the Wireshark screenshot:

no: 56, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: Encrypted Handshake Message

Based on information from RFC 5246, after switching to encrypted communication, the client sends the final Finished message, encrypted with the negotiated keys. The next step that can be observed is 'New Session Ticket' which can be seen in the Wireshark screenshot:

no: 58, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: new session ticket

Based on information from RFC 5246, the server may send a NewSessionTicket message to the client, allowing session resumption in the future without a full handshake. Can observe that this ticket has a lifetime hint associated with it, 1 day, which helps the client understand how long this ticket is usable before it needs to establish a new session ticket with the server.

Session Ticket Lifetime Hint: 86400 seconds (1 day)

Session Ticket Length: 1058

The next step that can be seen is 'Change Cipher Spec' which can be seen in the Wireshark screenshot:

no: 60, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: Change Cipher Spec

Based on information from RFC 5246, the server sends the ChangeCipherSpec message to indicate that it will now use the agreed-upon encryption keys for further communication using the negotiated cipher suite. The next step that can be seen is 'Encryption Handshake Message' which occurs server side. This can be seen in the Wireshark screenshot:

no: 62, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: Encrypted Handshake Message

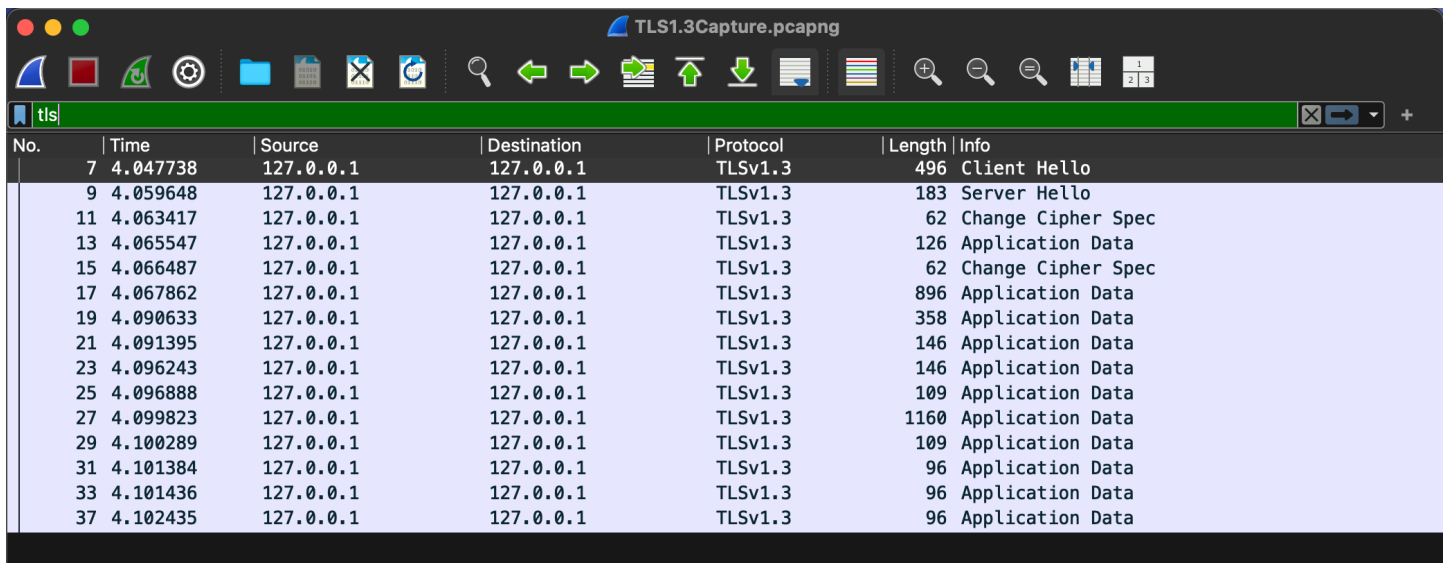
Based on information from RFC 5246, after switching to encrypted communication, the server sends the final Finished message to complete the handshake, encrypted with the negotiated keys. The last thing we can see in the Wireshark screenshot is the Application Data being transferred. This can be seen in Wireshark:

no: 64 – 66, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.2, info: Application Data

Based on information from RFC 5246, now that the handshake is complete and both sides have exchanged encrypted Finished messages, encrypted application data can be sent securely between the client and server using the established TLS session.

Wireshark Screenshots from TLS 1.3 capture:

TLS 1.3 handshake is a more streamlined version of TLS 1.2. Used the server.java and client.java code established in Part 1 of this assignment to carry out this server and client communication using 1.3.



No.	Time	Source	Destination	Protocol	Length	Info
7	4.047738	127.0.0.1	127.0.0.1	TLSv1.3	496	Client Hello
9	4.059648	127.0.0.1	127.0.0.1	TLSv1.3	183	Server Hello
11	4.063417	127.0.0.1	127.0.0.1	TLSv1.3	62	Change Cipher Spec
13	4.065547	127.0.0.1	127.0.0.1	TLSv1.3	126	Application Data
15	4.066487	127.0.0.1	127.0.0.1	TLSv1.3	62	Change Cipher Spec
17	4.067862	127.0.0.1	127.0.0.1	TLSv1.3	896	Application Data
19	4.090633	127.0.0.1	127.0.0.1	TLSv1.3	358	Application Data
21	4.091395	127.0.0.1	127.0.0.1	TLSv1.3	146	Application Data
23	4.096243	127.0.0.1	127.0.0.1	TLSv1.3	146	Application Data
25	4.096888	127.0.0.1	127.0.0.1	TLSv1.3	109	Application Data
27	4.099823	127.0.0.1	127.0.0.1	TLSv1.3	1160	Application Data
29	4.100289	127.0.0.1	127.0.0.1	TLSv1.3	109	Application Data
31	4.101384	127.0.0.1	127.0.0.1	TLSv1.3	96	Application Data
33	4.101436	127.0.0.1	127.0.0.1	TLSv1.3	96	Application Data
37	4.102435	127.0.0.1	127.0.0.1	TLSv1.3	96	Application Data

The first step that can be observed is a ClientHello from the client to the server. This can be seen in the Wireshark screenshot:

no: 7, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.3, info: Client Hello

Based on information from RFC 8446, the ClientHello is the first message sent by the client, however, TLS 1.3 introduces several optimizations compared to 1.2 ClientHello. The ClientHello message contains:

- client_version: the highest version of TLS the client supports, in this case, using TLS 1.3 based on the implementation of the clientTLS.java script.
- random: random number generated by the client, used in the key generation process.

ccc8d1c56f6439ab5cab6021132bf7fa786a35ba008b3548ab738679f3096e89

- session_id: If the client wishes to resume a session, it includes the ID of the session.

5f62592b5b21779a0ca476bd37f75f6171d58baf66661ce5b0de640ebe2cacbc

- cipher_suites: A list of supported cryptographic algorithms supported by the client, ordered by preference. Sent 37 suites in total in the capture.

Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x009f)
Cipher Suite: TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xccaa)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_GCM_SHA384 (0x00a3)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_GCM_SHA256 (0x00a2)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0x006b)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA256 (0x006a)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 (0x0067)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA256 (0x0040)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA256 (0x003d)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)

- supported_groups: indicates the supported groups for key exchange.

Supported Group: x25519
Supported Group: secp256r1 (0x0017)
Supported Group: secp384r1 (0x0018)
Supported Group: secp521r1 (0x0019)
Supported Group: x448 (0x001e)
Supported Group: ffdhe2048 (0x0100)
Supported Group: ffdhe3072 (0x0101)
Supported Group: ffdhe4096 (0x0102)
Supported Group: ffdhe6144 (0x0103)
Supported Group: ffdhe8192 (0x0104)

- signature hash algorithms: list of supported signature algorithms for verifying the digital signatures. 21 algorithms in total:

Signature Algorithm: ecdsa_secp256r1_sha256 (0x0403)
Signature Algorithm: ecdsa_secp384r1_sha384 (0x0504)
Signature Algorithm: ecdsa_secp521r1_sha512 (0x0603)
Signature Algorithm: ed25519 (0x0807)
Signature Algorithm: ed448 (0x0808)
Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
Signature Algorithm: rsa_pss_rsae_sha512 (0x0806)
Signature Algorithm: rsa_pss_pss_sha256 (0x0809)
Signature Algorithm: rsa_pss_pss_sha384 (0x080a)
Signature Algorithm: rsa_pss_pss_sha512 (0x080b)
Signature Algorithm: rsa_pkcs1_sha256 (0x0401)
Signature Algorithm: rsa_pkcs1_sha384 (0x0501)
Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
Signature Algorithm: SHA256 DSA (0x0402)
Signature Algorithm: SHA224 ECDSA (0x0303)
Signature Algorithm: SHA224 RSA (0x0301)
Signature Algorithm: SHA224 DSA (0x0302)
Signature Algorithm: ecdsa_sha1 (0x0203)
Signature Algorithm: rsa_pkcs1_sha1 (0x0201)
Signature Algorithm: SHA1 DSA (0x0202)

- supported_versions: informs the server of all TLS versions supported by the client

Supported Version: TLS 1.3 (0x0304)
Supported Version: TLS 1.2 (0x0303)

- key_share: one major difference is the inclusion of the key_share extension, where the client sends its public key ((EC)Diffie-Hellman) to initiate the key exchange. Part of the process for negotiating encryption keys between client and server. The group used is x25519 is an elliptic curve (Curve25519) and secp256r1 which is another elliptic curve. These two groups provided will give the server the option to use for key_exchange.

```
PSK Key Exchange Mode: PSK with (EC) DHE key establishment (psk_dhe_ke) (1)
Client Key Share Length: 105
Key Share Entry: Group: x25519, Key Exchange length: 32
Key Share Entry: Group: secp256r1, Key Exchange length: 65
```

The client sends the ClientHello and immediately offers key material for the key exchange. After sending the ClientHello, the client waits for the server's response.

The next step that can be observed is the ServerHello. This can be seen in the Wireshark screenshot:

```
no: 9, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.3, info: Server Hello
```

Based on information from RFC 8446, the server responds with the ServerHello, which contains:

- **Server_version:** TLS version selected by the server (typically the highest supported version common to both parties), but in this case, using TLSv1.3 based on the implementation of the serverTLS.java script.
- **random:** random number generated by the server, used in the key generation process.

```
d89b8dd7b882eafcb3d11c0eb346ecf090b191f56824182d81989c5ced8790c2
```

- **session_id:** if the client requests session resumption and the server agrees, it echoes the session ID. If not a new session ID is generated or left blank.

```
5f62592b5b21779a0ca476bd37f75f6161d58baf66661ce5bde640ebe2cacbc
```

- **cipher_suite:** cryptographic algorithms selected by the server from the client's list.

```
TLS_AES_256_GCM_SHA384
```

- **supported_versions:** new extension in TLS 1.3 that informs the client of all TLS versions supported by the server and what TLS version the server will use, which in this case is 1.3.

```
TLS 1.3 (0x0304)
```

- **Key_share:** The server responds with its public key (Diffie-Hellman), completing the key exchange. Key share is done through using the x25519 (Curve25519) which is the elliptic curve that the client provided the server to use for key exchange in the client hello.

```
Key Share Entry: Group: x25519, Key Exchange length: 32
```

Unlike TLS 1.2, TLS 1.3 reduces round-trips by combining the key exchange into the ServerHello, allowing the session to proceed faster. The server selects the appropriate cryptographic parameters, responds with its key share, and prepares for secure communication. The next step that is seen is the change in cipher spec on the client side, which can be observed in the Wireshark screenshot:

```
no: 11, src-dest: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.3, info: change cipher spec
```

Based on information from RFC 8446, ChangeCipherSpec is optional in 1.3 and doesn't directly affect the handshake. Is part of handshake due to server and client code implementation, however, it is normally included to ensure there is backward compatibility with TLS 1.2. The handshake proceeds with the application of the new keys without requiring this message. Both sides can now prepare for encryption using the newly negotiated. The next step is the client sending application data, which can be observed in the Wireshark screenshot:

no: 13, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.3, info: application data

Based on information from RFS 8446, one of the major changes in TLS 1.3 is that application data can be sent immediately after the server has provided the key share in its ServerHello message. This means that application data can be encrypted and exchanged more quickly in the handshake process as it only requires one round trip. In TLS 1.3 after the Server Hello the Certificate message and subsequent handshake messages that are normally part of 1.2 handshakes are typically encrypted with the shared keys. So, what is within the Application Data packet would contain the subsequent client encrypted handshake messages as well as the Certificate and Certificate verification.

By this stage, both the client and server have computed a shared secret using the Diffie-Hellman key exchange and can now securely communicate using the agreed-upon cipher suite. Both parties can securely exchange encrypted application data. The next step we can observe after this is the change in cipher spec on the server side, which can be observed in the Wireshark screenshot:

no: 15, src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.3, info: change cipher spec

Based on information from RFS 8446, like the earlier ChangeCipherSpec message sent by the client, this one is also optional and used primarily for back compatibility with TLS versions and is just showing confirmation from the server that it will be using the negotiated encryption and keys for secure communication. The last step we can observe is the following transfer of application data after the handshake. We can see this in the Wireshark screenshot:

no: 17 - ..., src-address: 127.0.0.1, dest-address: 127.0.0.1, protocol: TLSv1.3, info: application data

Based on information from RFC 8446, once the handshake is complete and both sides have agreed on session keys, the actual encrypted application data is exchanged. What is within the Application Data packet would contain the subsequent server encrypted handshake messages as well as the Certificate and Certificate verification. Encrypted data exchange continues for the rest of the session, with both parties sending data securely. The difference between 1.3 and 1.2 is to reduce round trips, key share in ServerHello message, simplified handshake, and forward secrecy by default (ephemeral DH key exchanges).

Part 3: Cryptographic Inspection of Open-Source Communication Apps

Application One: Mumble (Low-Latency Voice Communication)

How are messages encrypted?

Mumble uses TLS (Transport Layer Security) to ensure confidentiality. TLS relies on public-key cryptography and secures all voice and chat communications between clients and servers. It uses AES (Advanced Encryption Standard) for symmetric encryption after the TLS handshake, which provides secure encryption that ensures confidentiality of voice and text data that is transmitted. Establishes a TLS connection between client and server using X.509 certificates to initiate a secure session. After the handshake, AES_GCM is employed to encrypt all communications using session keys derived during the handshake. In 'SSL.h', handles the initialization of the OpenSSL library and manages TLS encryption. It ensures that the TLS handshake is performed, selects strong encryption ciphers (e.g. AES-GCM -> defaultOpenSSLCipherString()), and establishes secure communication between client and server. One limitation is Mumble administrators have access to the server's private key, meaning that an admin can decrypt the session's symmetric cipher key and access the communication. Based on the config abilities of server admins and the use of groups in Mumble ACL setup, this represents a vulnerability, as the admin can technically eavesdrop on communication by decrypting the encrypted traffic. Another vulnerability that can be seen is AES encryption using 128-bit key encryption which can be deemed as too small for keys to remain secure by modern standards.

- <https://www.mumble.info/documentation/administration/acl/>
- <https://github.com/mumble-voip/mumble/blob/cb01bfa5200fce53db68b769d05995c999e7cdd8/src/SSL.cpp#L136>

What mechanisms ensure message integrity?

Mumble uses hash-based MAC as part of the TLS protocol to ensure message integrity. HMAC ensures that the messages have not been altered in transit between the client and server. MAC is an authentication code produced by running the HMAC-SHA1 algorithm on the entire message when the MAC field is set to zeros. Within the TLS protocol, Mumble uses SHA-256, which is a commonly used HMAC function. This would ensure message integrity by using the cryptographic hash. This hash is recalculated/checked on both server and client ends during communication. During the TLS handshake, Mumble negotiates the use of HMAC-SHA256 for message integrity. The client and server use this to verify that no tampering occurred during transmission. Can observe the TLS transfer in line 135->143 (SSL.cpp) where it decides on what TLS version is going to be used (TLSv: 1.0, 1.1, 1.2, 1.3). The HMAC is integrated through the use of mumble using SSL in client-server communication in the CryptographicHash.h code (can see lines 19 and 21 -> SHA1, SHA256 respectively). No major limitations with HMAC in Mumble as long as session keys are protected which would mean that the Mumble models maintain integrity through message communication. Can observe HMAC implementation in the PBKDF2.cpp code. Code implements the PBKDF2 algorithm to generate secure cryptographic hashes through the use of the getHash function. The getHash function uses the PKCS5_PBKDF2_HMAC function from OpenSSL, applying HMAC with SHA-384 to derive a cryptographic key from a password and salt with a specified number of interactions.

- <https://github.com/mumble-voip/mumble/blob/cb01bfa5200fce53db68b769d05995c999e7cdd8/src/SSL.cpp>
- <https://github.com/mumble-voip/mumble/blob/cb01bfa5200fce53db68b769d05995c999e7cdd8/src/crypto/CryptographicHash.h>
- <https://erlangonxen.org/more/mumble>

- <https://github.com/mumble-voip/mumble/blob/cb01bfa5200fce53db68b769d05995c999e7cdd8/src/murmur/PBKDF2.cpp#L99>

How are users authenticated?

Mumble uses Public-Key Infrastructure (PKI) for user authentication. User authentication is based on X.509 certificates generated during client setup. On first use, Mumble auto-generates a self-signed certificate for the client. This certificate is used for user identification on the server. Users may also choose to upgrade their certificate to a Class 1 or Class 2 certificate, issued by a trusted Certificate Authority (CA). When a user connects to the server it verifies the client certificate using its public key infrastructure. The server checks the validity of the certificate, ensuring that only legitimate users can access the server. This process occurs during the TLS handshake, where the client presents its certificate for authentication. Generation of the X.509 certificate with the use of RSA and SSL can be seen in SelfSignedCertificate.h where generation of the certificate and RSA keypair occurs. Can see the authentication parameter in Server.h on line 435 where it is an integer called authenticate which comprises the session ID, hash of the certificate, and a list of certificates. Has a Cert.cpp file which is used to generate certificates extending the private key generated/applied. This will then initialize the certification using Diffie-Hellman. Generation is done over X.509 and SSL. The auto-generated self-signed certificates provide basic security but may not guarantee a high level of identity verification. Users must take additional steps to obtain class 1 or 2 certificates for enhanced security. Additionally, the server administrator, having access to the private key, may be able to impersonate users or decrypt communication.

- <https://github.com/mumble-voip/mumble/blob/cb01bfa5200fce53db68b769d05995c999e7cdd8/src/SelfSignedCertificate.h>
- <https://github.com/mumble-voip/mumble/blob/cb01bfa5200fce53db68b769d05995c999e7cdd8/src/murmur/Server.h#L4>
- <https://github.com/mumble-voip/mumble/blob/cb01bfa5200fce53db68b769d05995c999e7cdd8/src/murmur/Cert.cpp#L18>
- <https://www.mumble.info/documentation/user/certificates/>

Are communications encrypted from sender to receiver?

Mumble's communication between clients and servers is encrypted using TLS. By using TLS, this would therefore provide encryption (of message and voice data) from client to server during transit. Mumble does not implement true End-to-End Encryption however as the server can decrypt the traffic. Mumble encrypts all traffic between client and server using TLS 1.0 or higher, depending on configuration. All communication is encrypted using symmetric encryption keys derived from TLS handshake. The establishment of a TLS tunnel for communication is managed using OpenSSL which creates a copy of this and creates a MumbleSSL to then check the TLS version used (e.g. 1.0, 1.1, 1.2, 1.3). The establishment of the TLS communication can also be seen in Connection.h where it creates an SSL socket based on the TCP, protocol, TCL message type, and whether the certificate is present. Returns the peer's chain of digital certificates, starting with the peer's immediate certificate and ending with the CA's certificate. Since Mumble does not implement full end-to-end encryption, the server acts as a trusted intermediary. This means that server administrators could decrypt traffic if malicious. This limitation is a trade-off for performance and manageability, but it exposes a risk if the server administrator is not trusted. Another limitation is the use of TLS 1.0 and 1.1 functionality in the codebase as these TLS versions have weak ciphers that can be exploited to decrypt traffic.

- <https://github.com/mumble-voip/mumble/blob/cb01bfa5200fce53db68b769d05995c999e7cdd8/src/SSL.cpp#L136>

- <https://github.com/mumble-voip/mumble/blob/cb01bfa5200fce53db68b769d05995c999e7cdd8/src/Connection.h#L37>

Application Two: Signal (Private Messenger) -> messaging, voice, and video calls

- <https://signal.org/docs/>

How are messages encrypted?

Ensures confidentiality using the Double Ratchet algorithm, which enables both parties to update their encryption keys continuously during communication. This guarantees that past communication cannot be decrypted even if keys are compromised. Encryption of the messages is done using AES-CBC (cipher block chaining) and AES-GCM (Galois/counter mode) depending on the context. Both modes offer strong encryption, but GCM provides both encryption and message authentication, while CBC primarily focuses on encryption with authentication provided separately. GCM is the primary mode used for message encryption within Signal due to its efficiency and ability to provide both confidentiality and integrity in one step. CBC is used for media encryption or where message size or performance are needed for communication. CBC requires the use of separate integrity mechanisms such as HMAC. The Double Ratchet algorithm is used for key exchange and message encryption with each message having unique keys, providing forward secrecy. The X3DH (extended triple Diffie-Hellman) key agreement protocol is used to establish a shared secret key between users when they first communicate. X3DH uses an elliptic curve (through the use of Diffie-Hellman) to establish these shared keys. Can observe the implementation of the Double Ratchet algorithm in the X3DHBundleFactory.java where it generates key pairs, that are then signed. This is done through the use of an elliptic curve to calculate the signature generate/get the private keys and serialize. The key is then stored in the record along with the associated ID (of communication) where it is then sent during communication. Can observe initialization of session in ratchet.rs where these keys are derived through the use of SHA256 to maintain integrity, where initialization of 'Alice' and 'Bob' is done, where it initialized with their corresponding key pairs that are saved. Can observe AES-CBC/GCM operations in session_cipher.rs. AES-GCM is highly secure, but when AES-CBC is used, there is a reliance on additional integrity mechanisms like HMAC to ensure that encrypted messages are not tampered with.

- <https://signal.org/docs/>
- <https://github.com/signalapp/libsignal/blob/d66f24502460fe34dbf99b5b09dbd01721da9375/java/client/src/test/java/org/signal/libsignal/protocol/X3DHBundleFactory.java#L4>
- <https://github.com/signalapp/libsignal/blob/29d32702b5cb0dc302ca2fa08377cddaaaacb60c/rust/protocol/src/ratchet.rs#L47>
- https://github.com/signalapp/libsignal/blob/d66f24502460fe34dbf99b5b09dbd01721da9375/rust/protocol/src/session_cipher.rs#L4

What mechanisms ensure message integrity?

Signal uses HMAC (Hash-based Message Authentication Code) along with AES-GCM and CBC to ensure message integrity. HMAC is responsible for verifying that the messages have not been altered in transit between the sender and the receiver. This ensures that the messages have not been tampered with during transmission. AES-GCM provides both confidentiality and integrity checks, meaning that the message is authenticated as part of the encryption process. For AES-CBC, Signal uses HMAC-SHA256 to ensure message integrity. HMAC-SHA256 generates a secure cryptographic hash of the message, allowing both parties to verify that the message has not been altered. Signal combines encryption with integrity checks to protect the entire message, including its metadata. When AES-GCM is used, the authentication happens within the same process as encryption. For AES-CBC, a separate HMAC-SHA256 process is executed to verify the message's integrity. Can observe HMAC_SHA_256 initialization in the

Signal Server repository in the HmacUtils.java class. Provides functions for generating HMAC hashes using instances of MAC and converting results to various formats (byte arrays, hex strings). Offers methods to compare two HMAC values for equality, which is served to verify data integrity and authenticity, to ensure that a message or piece of data has not been altered (if computed HMAC matches expected HMAC it confirms data has not been tampered with). A limitation that can be stated but is managed well in Signal is AES encryption while AES-GCM handles encryption and integrity in one step, AES-CBC requires additional overhead for the separate HMAC calculation and verification, which can be less efficient.

- <https://github.com/signalapp/Signal-Server/blob/087c2b61eeb6f6aa934aa3b5f886b84b780687d7/service/src/main/java/org/whispersystems/textsecuregcm/util/HmacUtils.java#L16>

How are users authenticated?

Signal uses public-key cryptography for user authentication. The X3DH (Extended Triple Diffie-Hellman) key agreement protocol ensures users can authenticate each other using their public keys. It also utilizes Curve25519 for generating and exchanging public keys, and EdDSA (Edwards-curve Digital Signature Algorithm) to create digital signatures that authenticate users. “The X3DH Protocol establishes a shared secret key between two parties who mutually authenticate each other based on public keys. X3DH provides forward secrecy and cryptographic deniability”. EdDSA (Edwards-curve Digital Signature Algorithm) enables use of a single key pair format for both elliptic curve Diffie-Hellman and signatures. Used to sign the long-term identity keys and session setup ensuring authenticity and verification of public keys. Can observe Curve25519 and EdDSA implementation to perform cryptographic key exchange and signature operations in curve25519.rs in the libsignal repository. Code generates private keys and computes Diffie-Hellman key agreements by performing scalar multiplications on the elliptic curve using the ED25519_BASEPOINT_TABLE. For signing it uses XEdDSA which derives a Ed25519-public-key from a Curve25519 private key and uses a hashed message along with a private scalar to compute a signature. Verification of signatures involves converting the Curve25519 public key into its EdDSA form and checking the signature matches the expected result and point on the curve. If users do not verify the authenticity of public keys, there is a potential vulnerability to man-in-the-middle attacks.

- <https://signal.org/docs/specifications/x3dh/>
- <https://github.com/signalapp/libsignal/blob/d66f24502460fe34dbf99b5b09dbd01721da9375/rust/protocol/src/curve/curve25519.rs#L52>

Are communications encrypted from sender to receiver?

Signal provides true End-to-End Encryption for all messages, calls, and media using the Double Ratchet algorithm in combination with AES-GCM or AES-CBC. This ensures that messages are encrypted on the sender’s device and only decrypted on the receiver’s device, with Signal servers having no access to the plaintext content or decryption (private) keys. Signal uses either AES-GCM or AES-CBC for message encryption. The encryption is applied after a secure key exchange using X3DH, and all message content remains encrypted from the sender to the receiver. Once the key exchange is complete via X3DH, the Double Ratchet algorithm takes over, generating new keys for each message. These keys are used to encrypt the message payload using AES-GCM or AES-CBC, depending on the context. Can observe signal protocol implementation of double ratchet algorithm to achieve end-to-end communication in the lib.rs class in the libsignal library. This class represents cryptographic functionalities required for elliptic curve within the Signal Protocol. Supports key protocols like X3DH (Extended Triple Diffie-Hellman) for secure key agreement and the Double Ratchet algorithm for encrypted message exchanges that ensure forward secrecy. X3DH (initial key exchange between two parties) is used to establish a secure shared secret using a combination of pre-keys. initialize_alice_session_record and initialize_bob_session_record are used to set up secure sessions by combining these keys to derive a shared secret both parties will use.

Once the shared secret is established using X3DH, the double ratchet algorithm is responsible for encrypting and decrypting messages. The ratchet module handles this, where the keys are continuously updated for each message providing forward secrecy. Even if one message key is compromised future/past messages remain secure. `message_encrypt` and `message_decrypt_signal` methods in the `session_cipher` module handle the actual encryption and decryption of the message, which will use the corresponding AES-GCM or AES-CBC depending on the context. Signal's end-to-end encryption guarantees strong security, but users need to manually verify each other's to mitigate the risk of MiTM attacks. Signal's security relies on 'Alice' and 'Bob' using Signal to encrypt their messages. A limitation that can be stated for communication encryption is "if you communicate with someone not using Signal, your messages will not be encrypted. While your conversations within Signal are secure, your overall communication may still be vulnerable if you interact with non-Signal users."

- <https://signal.org/docs/specifications/doubleratchet/>
- <https://signal.org/docs/specifications/x3dh/>
- <https://github.com/signalapp/libsignal/blob/d66f24502460fe34dbf99b5b09dbd01721da9375/rust/protocol/src/lib.rs#L36>
- <https://bluegoatcyber.com/blog/signal-app-review-security-and-privacy-evaluated/#:~:text=If%20you%20communicate%20with%20someone,interact%20with%20non%20Signal%20users.>

References

- <https://chatgpt.com> (help debugging and correcting part 1 script implementation as well as help in using correct commands for certificate generation. help interpreting mumble and signal documentation in relation to their codebases. Helped analysis of classes that provided relevant encryption, certificate, mac and AES implementation for both signal and mumble)
- <https://www.ibm.com/docs/en/sdk-java-technology/8?topic=handshake-tls-12-protocol> (for understanding TLS 1.2 protocol for server and client scripts for part 1, as well as interpreting Wireshark outputs for part 2)
- <https://www.ibm.com/docs/en/sdk-java-technology/8?topic=handshake-tls-13-protocol> (for understanding TLS 1.3 protocol for part 1 scripts, as well as interpreting Wireshark outputs for part 2)
- <https://www.baeldung.com/java-keystore> (for understanding keystore implementation and relative java.security functions for part 1 scripts)
- <https://datatracker.ietf.org/doc/html/rfc5246> (RFC 5246 documentation for part 2 analysis of TLS 1.2)
- <https://datatracker.ietf.org/doc/html/rfc8446> (RFC 8446 documentation for part 2 analysis of TLS 1.3)
- PART 3 REFERENCES ARE UNDER THEIR RESPECTIVE SECTIONS