|  |
| --- |
| In [Noise Framework's](https://github.com/noiseprotocol/noise_spec/blob/v34/output/noise.pdf) terminology, XX means that the static (long term identity) public keys, of both parties involved in a key agreement transaction, are transmitted over the channel as part of the key agreement messages.  This pattern is useful when two parties that do not know each others static public keys but have a means to exchange a sequence of message encounter each other in the real world.  For example:   * I buy a new connected doorbell/speaker/outlet/fire alarm/printer etc. * This new device D, when turned on, starts a WIFI access point temporarily so it can be onboarded * My phone P connects with this access point using WIFI * D is running a TCP server at some well known address in that temporary WIFI network * P makes a TCP connection and initiates an XX handshake.   **Description**  **Initiator**: The party that initiates the handshake i.e send the first message **Responder**: The party that is responding to a handshake message  Noise Protocol Framework  XX:  -> e  <- e, ee, s, es  -> s, se  **Variants**  X1X:  -> e  <- e, ee, s, es  -> s  <- se  XX1:  -> e  <- e, ee, s  -> es, s, se  X1X1:  -> e  <- e, ee, s  -> es, s  <- se |

[](https://github.com/mrinalwadhwa)

Author

[**mrinalwadhwa**](https://github.com/mrinalwadhwa)**commented**[**6 days ago**](https://github.com/ockam-network/proposals/issues/14#issuecomment-570505343)**•**

**edited**

|  |
| --- |
| Breaking down exactly as per the noise paper:  XX:  -> e  <- e, ee, s, es  -> s, se  Both parties maintain the following state variables:   * s, e: The local party’s static and ephemeral key pairs (which may be empty). * rs, re: The remote party’s static and ephemeral public keys (which may be empty). * h: A handshake hash value that hashes all the handshake data that’s been sent and received * ck: A chaining key that hashes all previous DH outputs. Once the handshake completes, the chaining key will be used to derive the encryption keys for transport messages * k, n: An encryption key k (which may be empty) and a counter-based nonce n. Whenever a new DH output causes a new ck to be calculated, a new k is also calculated. The key k and nonce n are used to encrypt static public keys and handshake payloads   Expands to:  Initiator | Responder  ------------------------------------------|----------------------------------------  M1 (Send)  -> e  1. Pick a static 25519 keypair for  this handshake and set it to s  2. Generate an ephemeral 25519  keypair for this handshake and set  it to e  3. Set k to empty, Set n to 0  4. Set h and ck to  'Noise\_XX\_25519\_AESGCM\_SHA256'  5. h = SHA256(h || prologue),  prologue is empty  6. h = SHA256(h || e.PublicKey),  Write e.PublicKey to outgoing message  buffer, BigEndian  7. h = SHA256(h || payload),  payload is empty  ------------------------------------------|----------------------------------------  M1 (Receive)  -> e  1. Pick a static 25519 keypair for  this handshake and set it to s  2. Generate an ephemeral 25519  keypair for this handshake and set  it to e  3. Set k to empty, Set n to 0  4. Set h and ck to  'Noise\_XX\_25519\_AESGCM\_SHA256'  5. h = SHA256(h || prologue),  prologue is empty  6. Read 32 bytes from the incoming  message buffer, parse it as a public  key, set it to re  h = SHA256(h || re)  7. read remaining message as payload  h = SHA256(h || payload),  payload should be empty  ------------------------------------------|----------------------------------------  M2 (Send)  <- e, ee, s, es  1. h = SHA256(h || e.PublicKey),  Write e.PublicKey to outgoing message  buffer, BigEndian  2. ck, k = HKDF(ck, DH(e, re), 2)  n = 0  3. c = ENCRYPT(k, n++, h, s.PublicKey)  h = SHA256(h || c),  Write c to outgoing message  buffer, BigEndian  4. MixKey(DH(s, re))  5. c = ENCRYPT(k, n++, h, payload)  h = SHA256(h || c),  payload is empty  ------------------------------------------|----------------------------------------  M2 (Receive)  <- e, ee, s, es  1. Read 32 bytes from the incoming  message buffer, parse it as a public  key, set it to re  h = SHA256(h || re)  2. ck, k = HKDF(ck, DH(e, re), 2)  n = 0  3. Read 48 bytes the incoming  message buffer as c  p = DECRYPT(k, n++, h, c)  h = SHA256(h || c),  parse p as a public key,  set it to rs  4. ck, k = HKDF(ck, DH(e, rs), 2)  n = 0  5. Read remaining bytes of incoming  message buffer as c  p = DECRYPT(k, n++, h, c)  h = SHA256(h || c),  parse p as a payload,  payload should be empty  ------------------------------------------|----------------------------------------  M3 (Send)  -> s, se  1. c = ENCRYPT(k, n++, h, s.PublicKey)  h = SHA256(h || c),  Write c to outgoing message  buffer, BigEndian  2. ck, k = HKDF(ck, DH(s, re), 2)  n = 0  3. c = ENCRYPT(k, n++, h, payload)  h = SHA256(h || c),  payload is empty  ------------------------------------------|----------------------------------------  M3 (Receive)  -> s, se  1. Read 48 bytes the incoming  message buffer as c  p = DECRYPT(k, n++, h, c)  h = SHA256(h || c),  parse p as a public key,  set it to rs  2.ck, k = HKDF(ck, DH(e, rs), 2)  n = 0  3. Read remaining bytes of incoming  message buffer as c  p = DECRYPT(k, n++, h, c)  h = SHA256(h || c),  parse p as a payload,  payload should be empty  ------------------------------------------|----------------------------------------  1. k1, k2 = HKDF(ck, zerolen, 2)  n1 = 0, n2 = 0  Use (k1, n1) to encrypt outgoing  Use (k2, n2) to decrypt incoming  1. k1, k2 = HKDF(ck, zerolen, 2)  n1 = 0, n2 = 0  Use (k1, n1) to decrypt incoming  Use (k2, n2) to encrypt outgoing |

handshake=Noise\_XX\_25519\_AESGCM\_SHA256

init\_static=000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f

resp\_static=0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20

gen\_init\_ephemeral=202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f

gen\_resp\_ephemeral=4142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f60

msg\_0\_payload=

msg\_0\_ciphertext=358072d6365880d1aeea329adf9121383851ed21a28e3b75e965d0d2cd166254

msg\_1\_payload=

msg\_1\_ciphertext=64b101b1d0be5a8704bd078f9895001fc03e8e9f9522f188dd128d9846d484665393019dbd6f438795da206db0886610b26108e424142c2e9b5fd1f7ea70cde8767ce62d7e3c0e9bcefe4ab872c0505b9e824df091b74ffe10a2b32809cab21f

msg\_2\_payload=

msg\_2\_ciphertext=e610eadc4b00c17708bf223f29a66f02342fbedf6c0044736544b9271821ae40e70144cecd9d265dffdc5bb8e051c3f83db32a425e04d8f510c58a43325fbc56

msg\_3\_payload=79656c6c6f777375626d6172696e65

msg\_3\_ciphertext=9ea1da1ec3bfecfffab213e537ed1791bfa887dd9c631351b3f63d6315ab9a

msg\_4\_payload=7375626d6172696e6579656c6c6f77

msg\_4\_ciphertext=217c5111fad7afde33bd28abaff3def88a57ab50515115d23a10f28621f842