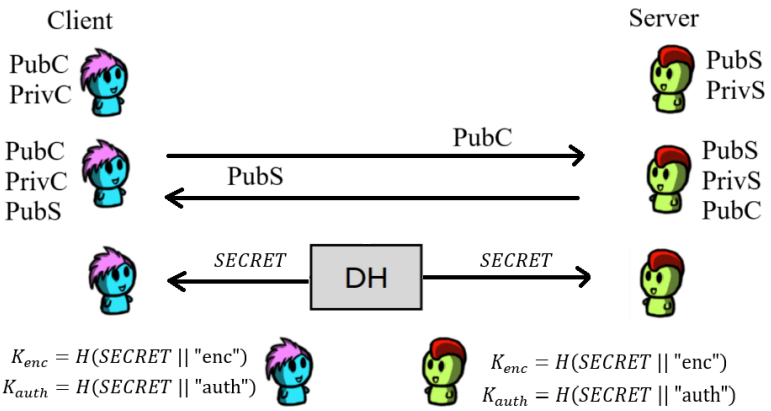
CPEN 442 - Assignment 5 - Task 1

Group: password123

Key Agreement Protocol

New keys can be generated by running the "generate_keys.py" script in our repository, then updating keys.py with the values output by that script.

For the first exchange, the client and the server exchange their ECC public keys as plaintexts. Since we are guaranteed that this interaction is uninterrupted, we can be sure that the client and the server exchange authentic public keys. We use the DH protocol to generate a shared $SECRET$. Each side computes a SHA-256 hash on a concatenation of $SECRET$ with each of the strings "enc" and "$auth$", to generate two 256-bit AES keys K_{enc} and K_{auth} (encryption and authentication keys respectively) to separate concerns. This protocol is protected against MitM attacks, since the adversary doesn't have one of the private keys to generate $SECRET$, and therefore can't compute the keys. Also, because SHA-256 is a strong hash function, even if an adversary gets ahold of one of the encryption key or the authentication key, they can't derive the other key from it.

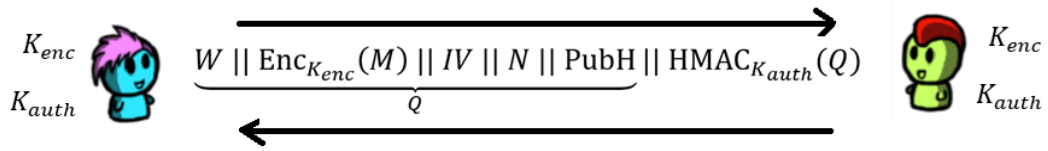


Communication Protocol

As a base protocol we use the OTR Rachet with three message types, one for user-generated messages and the two other for integrity warnings and general warnings.
After the key agreement, the server and the client both have their first K_{enc} and K_{auth} key pair. We use K_{enc} to encrypt the messages using AES-CBC (with an IV in plaintext alongside the encrypted message) and K_{auth} to authenticate them by generating an HMAC of the warnings, encrypted message, IV, nonce, and the new public key together. Along with each message, we include the warnings, IV, nonce, and the new public key as plaintext. The new public key is included in every message and signed using the HMAC, ensuring that even if it is modified in transit, such changes will be detected by the receiving party during HMAX verification. Since the public key is signed, it is safe to allow modification during transit, and these modifications don't compromise security. With every message exchange we derive a new K_{enc} and K_{auth} key pair the same way as in the key agreement protocol.

This protocol provides confidentiality because the adversary can't get access to the keys and our encryption function is strong. The HMAC provides integrity and authentication because the adversary does not have K_{auth}. K_{auth} is used for integrity by generating an HMAC from Q and comparing it to the HMAC provided in the payload to ensure the message hasn't been tampered with, and it ensures authentication by using it to verify that the messages were only sent by the parties participating in the specific session. Because either party can generate an HMAC, repudiation is provided. The HMAC is validated before decryption to ensure that we comply with the "verify first" principle. To protect against replay attacks and ensure message uniqueness, we implemented a nonce-based

logic alongside a hashing message mechanism. The server accepts any nonce that is greater than the last nonce it responded to, ensuring that every message from the client is unique. This approach prevents replayed messages by rejecting nonces that have already been used, allows the server to handle dropped messages gracefully by accepting retries with higher nonces, and simplifies validation by guaranteeing messages are processed in an order defined by their nonce. To further strengthen replay protection, we hash message identifiers and store these hashes in a set. This hash acts as a unique fingerprint for each message and enables the server to detect replay attack attempts.
Additionally the server sends acknowledgements to detect message drops. We issue a warning for every integrity and general issue on the server side. Symmetric encryption with AES ensures efficient communication. Using an HMAC rather than a MAC ensures that the overhead of all messages are the same length, which also ensures efficient communication.



W = Warnings
N = Nonce
PubH = new public key for next message (for OTR ratchet)

Change-log

Features added:

- We specified the public keys to be ECC-keys.
- We decided to use the OTR Rachet.
- We added the public key to the message format to support the OTR Rachet.
- We have three different message types which are also indicated in the message format.
- We added integrity and general warnings.

Features changed:

- We decided to use acknowledgements for assuring that no messages are dropped, modified or resent.

Features deleted:

- We deleted the nonce in the first key exchange.
- We no longer use timestamps or any other timing logic.
- We no longer cover delay attacks.

How to set up the code

The required packages can be found in the requirements.txt file.
To run the project, the port in the server, on which the server is listening and the port in the client on which the server is listening must be configured to the same port. After that you can run `python3 server_wrapper.py`, and `python3 client_wrapper.py` in two different terminals.
The legitimate messages and the security warnings will be logged in `server_output.txt`