2.2.7 End-to-end encryption

Historically, the process of encryption is considered to be symmetric one [Bellare et al., 1997]. It means that prior the communication, the parties conclude on the common secret key to be used in an encryption and decryption. This process is similar to the sharing keys first, then sharing the locked chest with the message. Such approach is highly cost since it requires to share the defined keys between each party taking place in secured communication. Much more simpler is to think about secured communication channel in terms of asymmetric encryption [Simmons, 1979]. The real life example would be

Alice shares with all the actors an opened lock, but keeps the key with herself.

So that Bob receives an opened lock, writes letter to Alice, puts letter to the chest, locks the chest with received from Alice lock. This way only Alice would be able to open the chest and to read the letter. This is an idea of the asymmetric encryption. However, such a simple communication concept sometimes requires complex number theory approach. A concept of opened lock may be interpreted in terms of one-way functions [Rompel, 1990]. One way function is the function that is easy to compute on every input, but hard to invert given the image of a random input. Thus, it is much simpler to close the lock without key, but very difficult to open lock combining various keys.

End-to-end encryption [Schillinger and Schindelhauer, 2019] is an asymmetric encryption such that the only communicating parties are able to decrypt the data. It means that even system administrators are not able to decrypt the messages transmitted between parties via their communication channel. End-to-end encryption can be reached via numerous approaches. Generally, there are two ways to implement E2E encryption

- Sharing public key to be used in encryption of the secret message, then encryption is done by the public key's owner, so-called asymmetric encryption. Public key owner is able then to decrypt secret message. For example, RSA algorithm.

- Asymmetric key exchange where parties exchanging the keys first, then symmetrically encrypting and decrypting the transferred data. For instance, Diffie–Hellman key exchange and AES256 encryption using common secret.

The most important aspect here is to securely store the secrets on the user's client application. Looking to the Telegram example, we can conclude that it does not make sense to implement end-to-end encryption for web and desktop clients [Job, Naresh, and Chandrasekaran, 2015; Sušánka and Kokeš, 2017; Lee et al., 2017], due to the storage security issues. Telegram uses the huge and heavy `MTProto 2.0` cryptographic protocol based on

Diffie—Hellman key exchange and further AES256 symmetric encryption. According to the project concerns, the E2E encryption via Diffie–Hellman key exchange and AES256 to be considered and implemented, the next section is about.

Diffie—Hellman key exchange. Diffie–Hellman (DH) protocol is a method of asymmetric exchange of the cryptographic keys for a group of two or more participants, developed in 1976 by cryptographers Ralph Merkle, Whitfield Diffie and Martin Hellman. In contrast to symmetric key exchange, the Diffie—Hellman protocol eliminates the direct transfer of the shared secret between the participants, each participant computes a shared secret with his own private-public key pair. The Diffie—Hellman protocol is based on a one-way function of the form

$$A = G^a \bmod P \tag{2.1}$$

where A is the user's public key, a is the user's private key, $P = 2Q + 1$ is modulus, such that 2048 bits safe-prime because Q is also prime, G is generator such that G is primitive root modulo P. We say that G is primitive root modulo P if for each $1 \leq a \leq P - 1$ the $A = G^a \bmod P$ is unique and belong to the set $\{1, 2, \ldots, P - 1\}$. The period of such cyclic group $\mathbb{Z}_P$ is $P - 1$ then.

Thus, the safety of the Diffie–Hellman protocol is based on the discrete logarithm problem, which is unsolvable in polynomial time if the constants G and P are chosen correctly. Graphically, the flow of the Diffie–Hellman protocol can be expressed through the analogy with mixing paints, as below picture shows

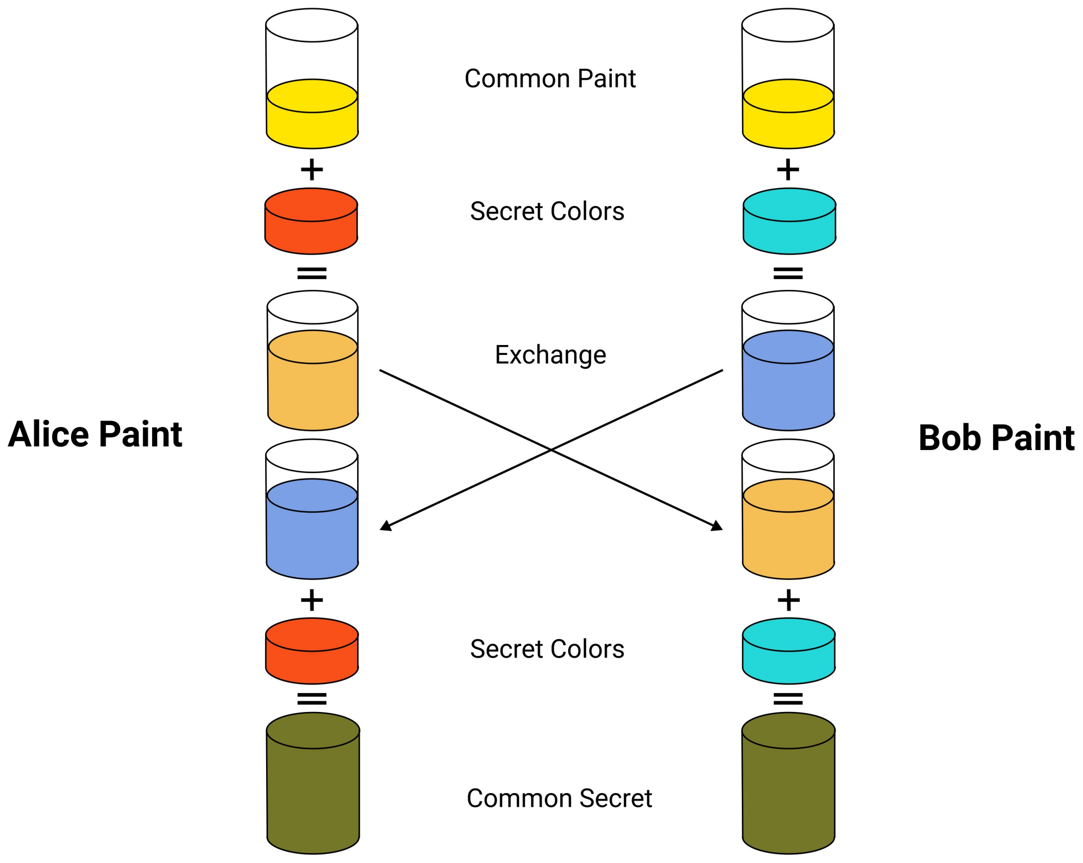


FIGURE 2.7: Diffie—Hellman key exchange concept diagram.

In contrast to the Diffie—Hellman based on discrete logarithm problem, there is an Elliptic Curve Diffie—Hellman key exchange, which based on the elliptic curve discrete logarithm problem. Although, the idea is quite same, the difference only in that Elliptic Curve Diffie—Hellman ensures the same safety as discrete logarithm Diffie—Hellman with lower value of the prime modulus P. For instance, 521 bit modulus used in Elliptic Curve Diffie—Hellman is equally safe as 2048 bit modulus in discrete logarithm Diffie—Hellman. To summarize, the flow of Diffie—Hellman key exchange is as follows

1. Given 2048 bits prime modulus P and generator G, such that G is primitive root modulo P.
2. Alice chooses her secret a.
3. Alice sends to Bob her public key $A = G^a \bmod P$.
4. Bob chooses his secret b.

5. Bob sends to Alice his public key $B = G^b \bmod P$.

6. Alice computes common secret $s = B^a \bmod P$.

7. Bob computes common secret $s = A^b \bmod P$.

8. Alice and Bob have arrived to the same value

$$A^b \bmod P = G^{ab} \bmod P \tag{2.2}$$

$$B^a \bmod P = G^{ba} \bmod P \tag{2.3}$$

Diffie—Hellman key exchange implementation via REST. Although, the idea of Diffie—Hellman key exchange looks quite simple, some remarks on the concrete implementation should be added. Firstly, it is necessary to implement the mechanism of key exchange request between two or more parties. As it discussed above, each user has his own private-public keys pair, so in order to perform request between parties, it should be implemented dedicate REST [Ong et al., 2015] web–service endpoint, for instance the `POST: api/key-exchange-requests` which takes the request body of the form

```
{
    "requestedUserId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
    "publicKey": "RUNLMSAAAAC2lkqYcTGhutQPxcjvoqUELKoy0"
}
```

So, request sender generates on the client side a key pair, keeps private on in the file system and shares the public in request to receiver. Therefore, the second party has received the key exchange request. In order to display all the key exchange requests awaiting the confirmation of decline decisions, it is worth to implement another REST endpoint such that `GET: api/key-exchange-requests`, so that requested party will have the list of requests to proceed. This endpoint may return the data structure like follows

```
{
    "keyExchangeRequests": [
        {
            "requestId": "81d314c1-913f-4686-827e-ef2a65ccc370",
            "senderId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
            "senderPublicKey": "RUNLMSAAAAC2lkqYcTGhutQPxcjvoqUELKoy0"
        }
    ],
    "message": "SUCCESS",
    "success": true
}
```

Finally, requested party should be able to confirm or decline the key exchange request, the `DELETE: api/key-exchange-requests` endpoint should be implemented then. The server is able to fetch the request thanks to the body endpoint takes

```
{
    "requestId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
    "confirmed": true,
    "publicKey": "string"
}
```

Therefore, an identifier of awaiting request is passed to the server among with boolean value indicating the confirmation. Under the roof of this operation are also generation of private-public keys pair for the requested party and generation of common secret stored in client's file system. As result, the initial request sender receives a public key as confirmation from requested party. Requested side may get all his public keys via the REST web–service using the resource `GET: api/public-keys`

```
{
    "publicKeys": [
        {
            "partnerId": "ae9e10a4-0c7e-4911-8450-4139d4a114a7",
            "partnerPublicKey": "RUNLMSAAAAAbc49wfaZ+QF9J2cu1S66bkp0"
        }
    ],
    "message": "SUCCESS",
    "success": true
}
```

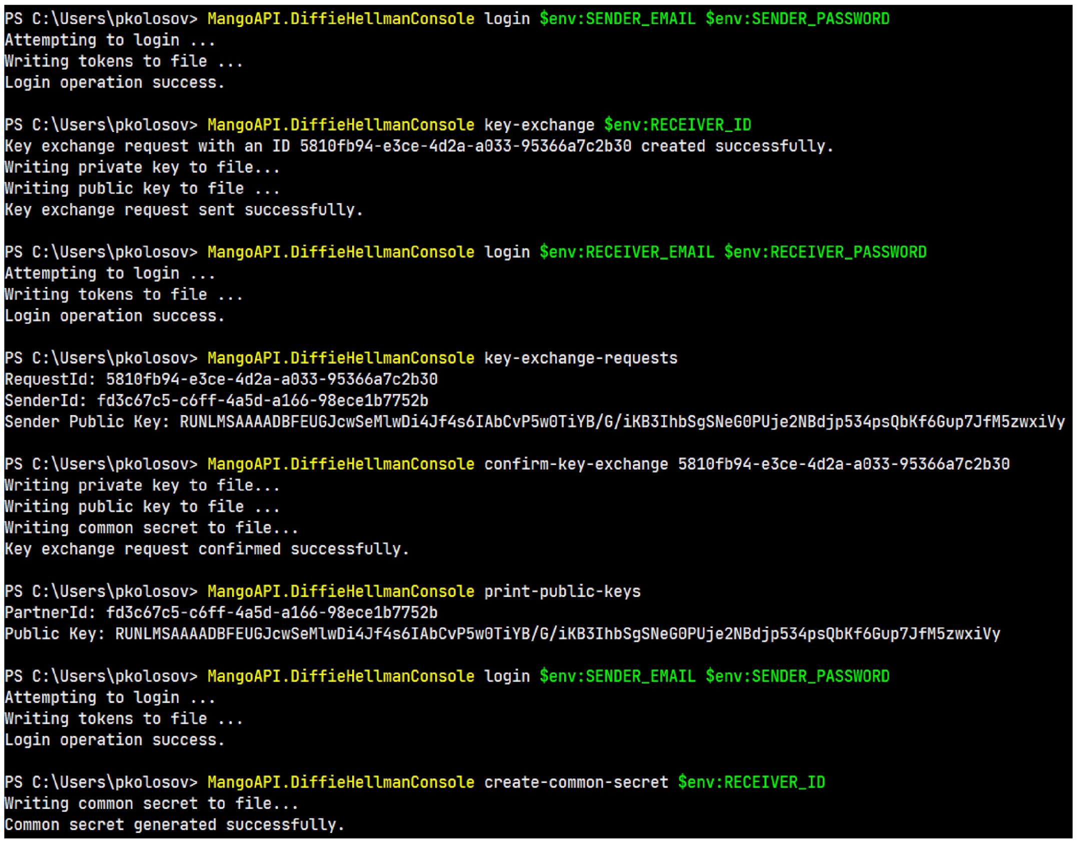
Now requested participant is able to derive the common secret. In order to provide an example, a simple command line interface is implemented. We have used an Elliptic Curve Diffie—Hellman implementation `ECDiffieHellmanCng Class` from the namespace `System.Security.Cryptography` of the .NET base class library. The `P-256` curve is used.

More precisely, the following CLI commands are implemented

- `MangoAPI.DiffieHellmanConsole login SENDER_EMAIL SENDER_PASSWORD`

- `MangoAPI.DiffieHellmanConsole key-exchange RECEIVER_ID`
- `MangoAPI.DiffieHellmanConsole key-exchange-requests`
- `MangoAPI.DiffieHellmanConsole confirm-key-exchange REQUEST_ID`
- `MangoAPI.DiffieHellmanConsole print-public-keys`
- `MangoAPI.DiffieHellmanConsole create-common-secret RECEIVER_ID`

Commands are self-explanatory, therefore we skip the detailed documentation on them. An example of console output straightforward



```
PS C:\Users\pkolosov> MangoAPI.DiffieHellmanConsole login $env:SENDER_EMAIL $env:SENDER_PASSWORD
Attempting to login ...
Writing tokens to file ...
Login operation success.

PS C:\Users\pkolosov> MangoAPI.DiffieHellmanConsole key-exchange $env:RECEIVER_ID
Key exchange request with an ID 5810fb94-e3ce-4d2a-a033-95366a7c2b30 created successfully.
Writing private key to file...
Writing public key to file ...
Key exchange request sent successfully.

PS C:\Users\pkolosov> MangoAPI.DiffieHellmanConsole login $env:RECEIVER_EMAIL $env:RECEIVER_PASSWORD
Attempting to login ...
Writing tokens to file ...
Login operation success.

PS C:\Users\pkolosov> MangoAPI.DiffieHellmanConsole key-exchange-requests
RequestId: 5810fb94-e3ce-4d2a-a033-95366a7c2b30
SenderId: fd3c67c5-c6ff-4a5d-a166-98ece1b7752b
Sender Public Key: RUNLMSAAAADBFEUGJcwSeMlwDi4Jf4s6IAbCvP5w0TiYB/G/iKB3IhbSgSNeG0PUje2NBdjp534psQbKf6Gup7JfM5zwxiVy

PS C:\Users\pkolosov> MangoAPI.DiffieHellmanConsole confirm-key-exchange 5810fb94-e3ce-4d2a-a033-95366a7c2b30
Writing private key to file...
Writing public key to file ...
Writing common secret to file...
Key exchange request confirmed successfully.

PS C:\Users\pkolosov> MangoAPI.DiffieHellmanConsole print-public-keys
PartnerId: fd3c67c5-c6ff-4a5d-a166-98ece1b7752b
Public Key: RUNLMSAAAADBFEUGJcwSeMlwDi4Jf4s6IAbCvP5w0TiYB/G/iKB3IhbSgSNeG0PUje2NBdjp534psQbKf6Gup7JfM5zwxiVy

PS C:\Users\pkolosov> MangoAPI.DiffieHellmanConsole login $env:SENDER_EMAIL $env:SENDER_PASSWORD
Attempting to login ...
Writing tokens to file ...
Login operation success.

PS C:\Users\pkolosov> MangoAPI.DiffieHellmanConsole create-common-secret $env:RECEIVER_ID
Writing common secret to file...
Common secret generated successfully.
```

FIGURE 2.8: Diffie—Hellman key exchange console output.

Finally, both test accounts reached the same common secret.