E2EE messaging service

Sam Hadow

December 14, 2025

git repository

Abstract

E2EE client-server messaging application proof of concept. Asymmetric cryptography used to login and identify accounts. Parts of Signal protocol used to secure conversations. Keccak AEAD and X25519 key exchange.
I absolutely do not recommend using such an application in a production environment.

Contents

1 Accounts

Asymmetric cryptography is used to identify each account. Ed25519 protocol is used for the following reasons:

- It offers 2^{128} bits of security.
- Faster than RSA. Just comparing key sizes, Ed25519 use 256-bit keys, for the same security 3072-bit RSA keys are needed. Comparing operations for signature and verification, Ed25519 is based on curve 25519 and uses computationally efficient operations. Whereas RSA uses modular exponentiation, which is much slower especially as key sizes increase.
- Curve 25519 is considered secure. [BL13]
- NIST P-curves don't seem to be trusted by everyone because of unexplained parameters choice [Gro15]

One downside is Curve 25519 is supported on Firefox and Safari, but not chromium browsers as indicated in the WebAPI subtle crypto documentation. With chromium browsers, the flag #enable-experimental-web-platform-features needs to be enabled, only NIST P-curves are supported by default.

The server stores public keys and never has access to corresponding private keys.

1.1 Registration

When registering the user need to provide a shared secret. This shared secret is used to restrict who can create an account. This shared secret is an environment variable server side and is supposed to be changed regularly by the system administrator. It isn't hashed server side as an attacker accessing the server could change it or create an account in the database anyway.

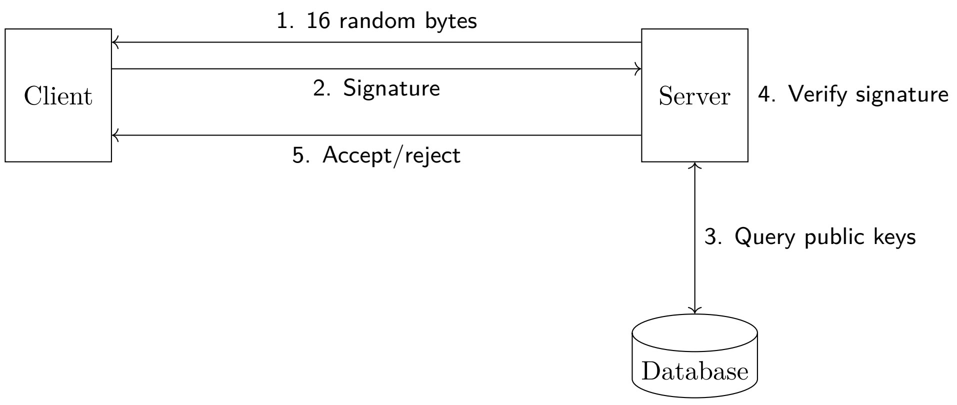
The user can either input a public key in PEM-format, or leave the public key box empty. If the public key box is left empty, a browser script will generate a key pair for the user, send the public key to the server for registration and display the private key letting the user save it (for example in a password manager).

In the implementation, the PEM-formatted public key is then converted to an hexadecimal representation on the server as it is easier to manipulate than a string potentially containing special characters. Since this string is added to the database, a parameterized SQL query is used to prevent any SQL injection, although the escape character ';' isn't a valid character in an hexadecimal string anyway.

1.2 Login

The login procedure relies on asymmetric cryptography. The server sends 16 randomly generated bytes, crypto.randomBytes() is used instead of Math.random() for cryptographically secure random numbers. The client signs this data with its private key and send the signature to the server. Then the server tries to verify the signature with every public key stored in the database, assuming not too many accounts are in the database this operation isn't too long. If the signature can be verified with a public key then the client can be identified, otherwise the login attempt is rejected.

Login procedure



2 Messages

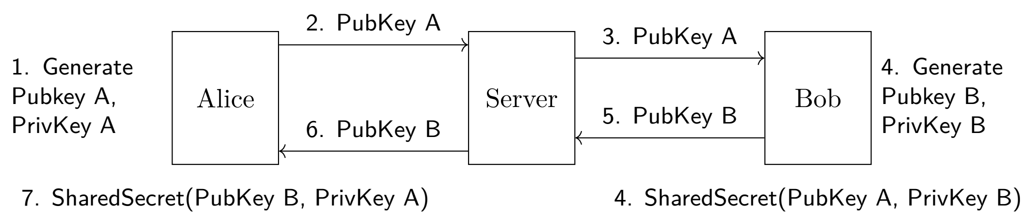
Conversations are encrypted and decrypted in the browser and the server never has access to the messages in plaintext. The protocol used is similar to Signal protocol.

2.1 Key exchange

X25519 (or ECDH on Curve 25519) is used for the key exchange. The package noble curves is used, the most recent version of the script is pulled directly from the git repository every time the container is built. An external package was used instead of re-implementing this protocol for the following reasons:

- Noble Curves is independently audited.
- ECDH relies on point operations on the curve instead of modular exponentiation like in a classic Diffie Hellman key exchange. Mistakes are easy to make and a homemade implementation would most likely end up being vulnerable to timing attacks.

Key exchange *(here Alice is the one initiating the key exchange)*



1. Generate Pubkey A, PrivKey A
Alice
2. PubKey A
Server
3. PubKey A
Bob
4. Generate Pubkey B, PrivKey B
5. PubKey B
6. PubKey B
7. SharedSecret(PubKey B, PrivKey A)
4. SharedSecret(PubKey A, PrivKey B)

A new key exchange can happen in 3 cases:

- Alice initiates a conversation with Bob (adding him in a room). Alice also initiates the key exchange.
- Alice and Bob are in a room. Bob websocket connection is lost for whichever reason. When Bob reconnects to the server, the server informs Alice (and any other contact Bob has). Alice then initiate a new key exchange with Bob.
- Alice and Bob are exchanging messages. Once Alice (or Bob) has sent 5 messages a new key exchange is initiated.

Renewing the shared secret regularly is necessary to prevent an attacker recovering one shared secret from being able to decrypt too many messages as the encryption keys are derived from the shared secret.

2.2 Key Derivation Function

Keccak sponge construction is used to derive keys. For this, the SHA-3 standard (Keccak) was re-implemented in JavaScript with the help of the pseudo-code and the python implementation provided by Team Keccak. [oST15] [BDH+18]

Keccak sponge construction is used in duplex mode to derive keys 1. In the KDF $|c| = 32$ Bytes to have at least 2^{128} of security, and $|r| = 16$ Bytes (the length of an encryption key). At each step a new salt can be added as an input in the KDF (a constant is used), and a 16 Bytes output is returned which can then be used as an encryption key. This KDF structure relies on the Keccak permutation which is assumed to be non-reversible, provided that c is large enough, to guarantee forward secrecy.

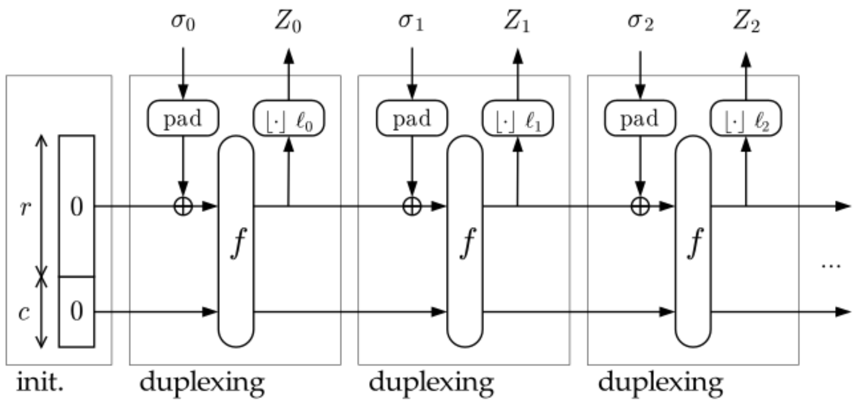


Figure 1: Keccak duplex sponge construction (source)

2.3 Ratchets

A different encryption key is used for each message, for that ratchets similar to the ones in Matrix Olm protocol or signal protocol are used. The previously defined Keccak KDF is the underlying KDF used in the ratchets. [PM16] [Com18] [Fou23]

Three ratchets are used:

- A Diffie Hellman ratchet.
- A receiving ratchet.
- A sending ratchet.

2.3.1 Diffie Hellman ratchet

The Diffie Hellman ratchet is used to derive 2 keys. In the implementation 6 keys are actually derived and then concatenated in 2 keys to have 48 Bytes of data for each key. The initial key is the shared secret between Alice and Bob, and the 2 keys derived are used as initial keys for the sending and receiving ratchets.

2.3.2 Receiving/Sending ratchet

Alice receiving ratchet corresponds to Bob sending ratchet. Similarly Bob receiving ratchet corresponds to Alice sending ratchet.

To send a message to Bob, Alice derives a key from her sending ratchet, encrypts the message using that key, and sends it to Bob. Bob, derives the same key from his receiving ratchet, allowing him to decrypt the message Alice sent.

2.4 Authenticated Encryption with Associated Data

AEAD is necessary for:

- Confidentiality with the encryption. Only Alice and Bob should know the content of the message, not even the server has access to it.
- Authenticity to avoid messages being forged and replay attacks.
- Integrity with the Associated to Data to detect tampering.

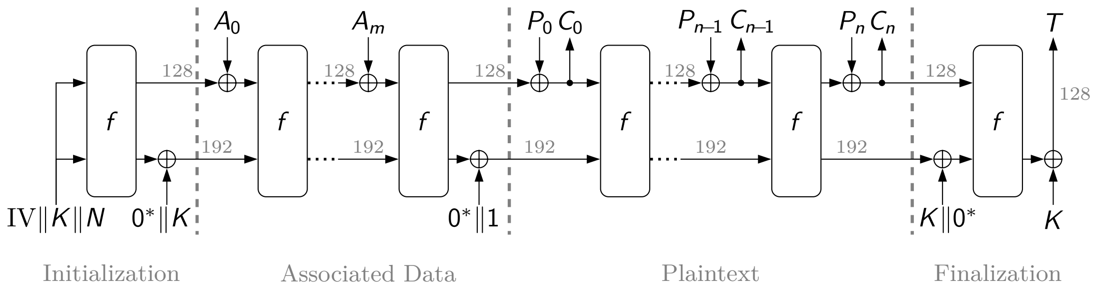
An Ascon-inspired Keccak AEAD is used to encrypt messages. The structure used is the same as the one defined in the NIST specification SP 800-232 for AEAD. But the Keccak permutation is used instead of the Ascon permutation. [TMC+24]

In the AEAD implementation |IV| = 12 Bytes, |N| = 12 Bytes and |K| = 16 Bytes to have 40 Bytes = 320 bits = 192+128 bits for the initial state. The Initialization Vector and the Nonce are randomly generated (with cryptographically secure random numbers). They're sent with the message and are therefore public but prevent 2 identical plaintext encrypted with the same key from corresponding to the same ciphertext.

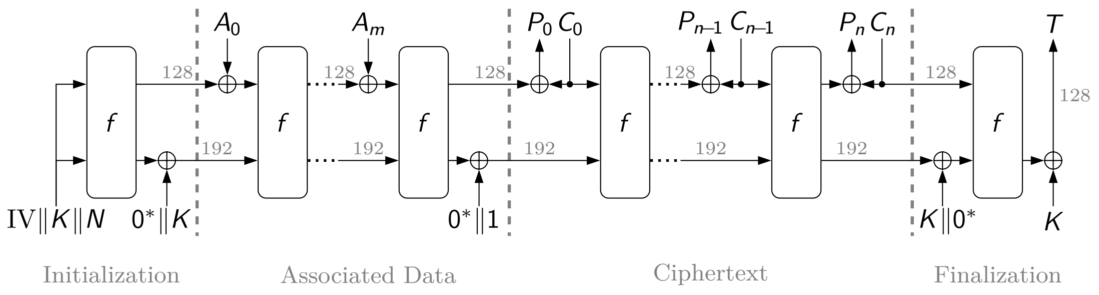
The Associated Data consists of the receiver's public key and the UTC time and date, rounded to the nearest minute. The time is rounded to the nearest minute to prevent the message from becoming invalid due to latency between its transmission and reception.

To decrypt a message, the user first verifies that the public key in the message matches the one of the user they are communicating with. Next, they decrypt the message and check if the tag received with the message matches the one obtained after decrypting. If there is a mismatch with either the public key or the tag, the message is considered corrupted.

Encryption mode



Decryption mode



(Ascon TeX resources have been used and modified for the 2 previous figures.) [DEMS19]

In the implementation, computations are done using Uint8arrays (tables of Unsigned Integers). And Bytes are transmitted between the server and the client in their hexadecimal representations.

3 Server side considerations

3.1 Deployment

A virtual machine would technically be the safest way to deploy the application. However for a better development workflow as well as easier deployments containers are used. Podman is the container engine used. Docker could be used as well. However podman has the advantage of not using a daemon with root access but a microservice approach with no root access needed instead. *(Docker can technically also be used in a rootless configuration but it's not the case by default and it's out of the scope of this document.)*

Kubernetes could also be used instead of a podman pod but the features are very similar and differences between a kubernetes and a podman pod are out of the scope of this document. A kubernetes pod yaml can be generated from a podman pod anyway.

Build

```
podman build -t e2ee-messaging-service .
```

Run

```
podman pod create --name=e2ee -p 3333:3333 --cpu=1 --memory=512m
podman run -d --pod=e2ee \
    -e POSTGRES_PASSWORD="password" \
    -e POSTGRES_DB="e2ee" \
    -e POSTGRES_USER="e2ee" \
    -e POSTGRES_INITDB_ARGS="--encoding=UTF-8 --lc-collate=C --lc-ctype=C" \
    -v /PATH/TO/DB:/var/lib/postgresql/data:Z \
    --name=e2ee-db docker.io/library/postgres:16
podman run -d --pod=e2ee \
    -e POSTGRES_PASSWORD="password" \
    -e POSTGRES_DB="e2ee" \
    -e POSTGRES_USER="e2ee" \
    -e SHARED_SECRET="change-me" \
    --name=e2ee-app e2ee-messaging-service:latest
```

The limits –cpu=1 –memory=512m on the pod create command indicates that the containers in the pod cannot use more than 1 core on the CPU and will be killed if they use for than 512 mebibytes of memory.

3.2 Security

As mentioned previously the containers are run in rootless mode. Running the containers in rootless is important to avoid any attacker from compromising the entire server if a container is compromised.

For added security podman secrets can be used instead of writing the passwords in clear in configuration files. *(note: podman secrets only prevents a secret from being accidentally revealed in a file or the command line history, it is still visible as an environment variable inside the container* *and can be extracted from the podman secret on the host system)*

SElinux is also used to label the files used by the database with the Z option, this prevent other processes on the server from being allowed to access them. Similarly SElinux prevent the containers from accessing any data they're not supposed to have access to on the host system if one of the container is compromised.

3.2.1 reverse proxy

NGINX is used as a reverse proxy on the server to expose only the port 443 to the outside world and various security settings like the authorized SSL ciphers and SSL protocols. The following NGINX configuration is used for the app:

```
server {
    listen 443 ssl;
    listen [::]:443 ssl;

    server_name e2ee.hadow.fr;

    ssl_certificate $ssl_certificate_path;
    ssl_certificate_key $ssl_certificate_key_path;

    fastcgi_read_timeout 5m;
    proxy_read_timeout 5m;

    location / {
        proxy_pass http://127.0.0.1:3333;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $http_host;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

4 Tests

Several unit tests can be run using npm test command. The tests assert that:

- The Keccak implementation is correct using test vectors provided by the NIST.
- The login procedure works correctly.
- The AEAD works properly, the tags should correspond when encrypting and decrypting a plaintext and its corresponding ciphertext with the same key, IV, Nonce and associated data. The equality $\mathrm{Id} = D_k \circ E_k$ should be true.
- Utility functions should work correctly (Converting a string in a specific format to another format or a type to another).

Unit tests do not assert that interactions client side work correctly (buttons binded to the correct functions and so on). These tests were done manually in a browser.

References

[BDH$^+$18] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak specifications summary. https://keccak.team/keccak_specs_summary.html, 2018.

[BL13] Daniel J. Bernstein and Tanja Lange. Choosing safe curves for elliptic-curve cryptography. https://safecurves.cr.yp.to/rigid.html, 2013.

[Com18] Computerphile. Video - double ratchet messaging encryption. https://www.youtube.com/watch?v=9sO2qdTci-s, 2018.

[DEMS19] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. https://ascon.isec.tugraz.at/resources.html, 2019.

[Fou23] Matrix.org Foundation. End-to-end encryption implementation guide. https://matrix.org/docs/matrix-concepts/end-to-end-encryption/, 2023.

[Gro15] Credelius Group. Why i don't trust nist p-256. https://credelius.com/credelius/?p=97, 2015.

[oST15] National Institute of Standards and Technology. Sha-3 standard: Permutation-based hash and extendable-output functions. Technical Report FIPS 202, National Institute of Standards and Technology, aug 2015.

[PM16] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016.

[TMC$^+$24] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Jinkeon Kang, and John Kelsey. Ascon-based lightweight cryptography standards for constrained devices: Authenticated encryption, hash, and extendable output functions. Technical Report NIST SP 800-232, National Institute of Standards and Technology, November 2024.