Murmly - E2EE Messaging

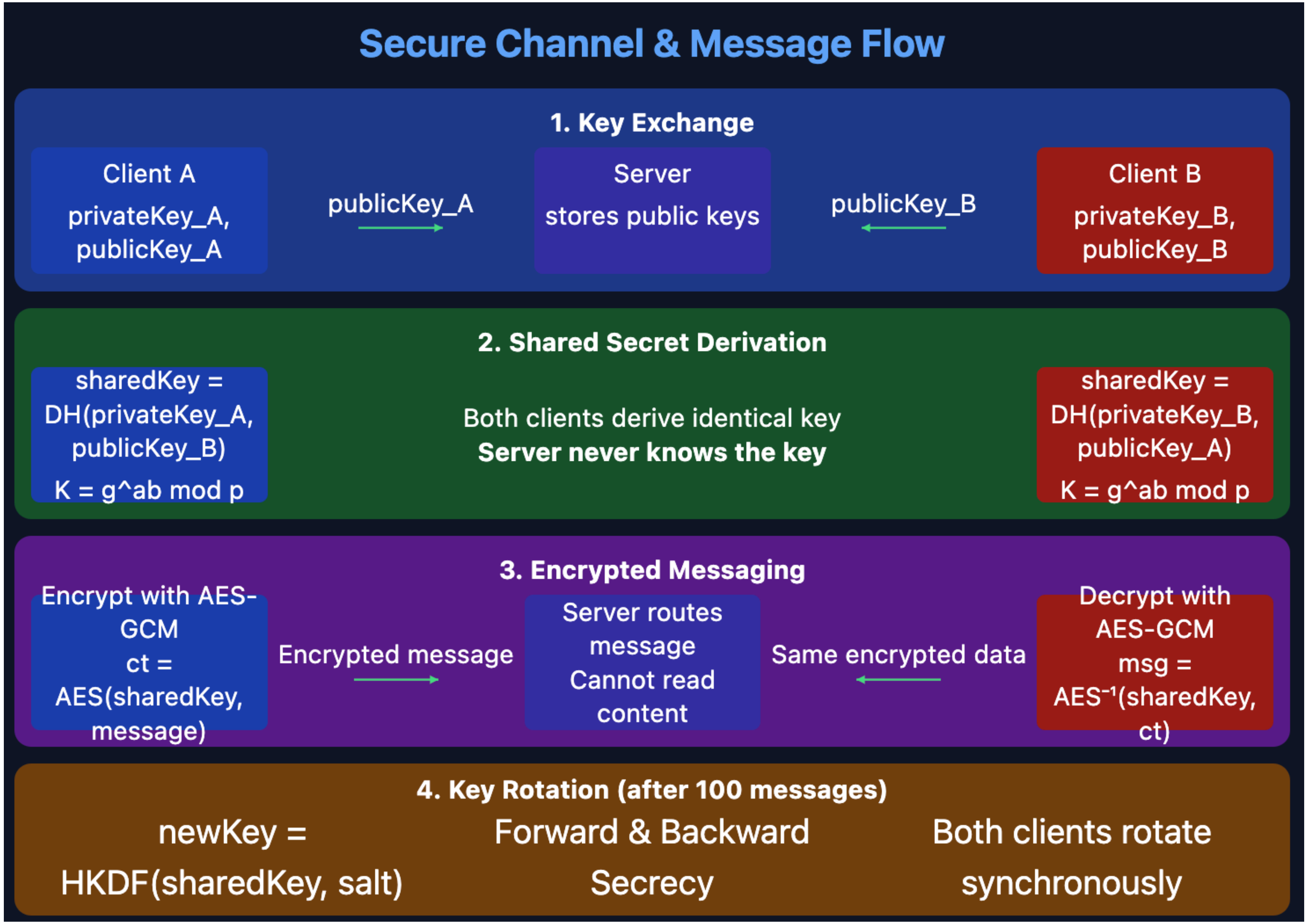
by Fabio Plunser, Cedric Sillaber

Our approach

- RESTful server using **FastAPI**
    - User authentication
    - Public key storage
    - Message routing only (no access to content)
- **WebSockets** for real-time messaging
- Python CLI client with `cryptography` library
- *in addition*: Full browser client (SvelteKit) with Web Crypto API

Cryptography

- client logs-in/registers, gets Diffie-Hellman parameters from server
- creates private and public key, uploads public key to server
- Tries to establish secure connection with other client and performing key exchange
- If key exchange is successful, the client will generate a symmetric key using the shared secret
  $\Rightarrow$ symmetric encryption (AES-GCM)



Secure Channel & Message Flow

1. Key Exchange

Client A

privateKey_A, publicKey_A

publicKey_A →

Server

stores public keys

← publicKey_B

Client B

privateKey_B, publicKey_B

2. Shared Secret Derivation

sharedKey = DH(privateKey_A, publicKey_B)

K = g^ab mod p

Both clients derive identical key

Server never knows the key

sharedKey = DH(privateKey_B, publicKey_A)

K = g^ab mod p

3. Encrypted Messaging

Encrypt with AES-GCM

ct = AES(sharedKey, message)

Encrypted message →

Server routes message

Cannot read content

← Same encrypted data

Decrypt with AES-GCM

msg = AES^{-1}(sharedKey, ct)

4. Key Rotation (after 100 messages)

newKey = HKDF(sharedKey, salt)

Forward & Backward Secrecy

Both clients rotate synchronously

Key Exchange: Diffie Hellman details

```python
# on server
def generate_dh_parameters():
    parameters: DHParameters = dh.generate_parameters(generator=2, key_size=PRIME_BITS)
    return parameters

# on client
def exchange_and_derive(priv_key: DHPrivateKey, peer_pub_key: DHPublicKey) -> bytes:
    shared_key: bytes = priv_key.exchange(peer_public_key=peer_pub_key)
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b"handshake data",
    ).derive(shared_key)
    return derived_key
```

AES-GCM Implementation

```python
def encrypt_aes_gcm(key: bytes, data: bytes, associated_data: bytes = None) -> bytes:
    # Generate random 12-byte nonce
    nonce = os.urandom(12)
    aesgcm = AESGCM(key)

    # Encrypt with AES-GCM
    ct = aesgcm.encrypt(
        nonce=nonce,
        data=data,
        associated_data=associated_data,
    )
    # Return nonce + ciphertext
    return nonce + ct
```

- Provides both **confidentiality** and **authenticity**
- Each message uses a unique IV (nonce)
- simpler solutin than in last project

Additional: Full browser client

- implemented a Webbrowser client using SvelteKit (javascript framework)
- implements own cryptography implementation, should be similar to python implementation.
- Challenge: Ensuring cross-platform compatibility

Additional: Web Client Cryptography

```
export async function deriveSharedSecret(
  privKey: DHPrivateKey,
  peerPubKey: DHPublicKey
): Promise<CryptoKey> {
  // Shared secret: (peer_pub_key.y ^ my_priv_key.x) mod p
  const sharedSecretBigInt = power(peerPubKey.y, privKey.x, privKey.params.p);

  // Derive key using HKDF (same as Python implementation)
  return window.crypto.subtle.deriveKey(
    {
      name: "HKDF",
      salt: new Uint8Array(0),
      info: new TextEncoder().encode("handshake data"),
      hash: "SHA-256",
    },
    importedKey,
    { name: "AES-GCM", length: 256 },
    false, ["encrypt", "decrypt"]
  );
}
```

what didn't work

- chat history
- web client in javascript/svelte communication with python client - did not work

lessons learned

- python implementation with simple CLI tool was relatively easy to implement
- browser client was more challenging, interoperability issues between cli and javascript
  - used most of our time
  - messed up our final submission

Demo

Let's see it in action!