Key generation mechanism

P,Q are generated as per the 2048 bit size by system.

Only input required to the key generation mechanism is the public exponent ‘e’

Output generated is in the form of PEM keys.

Step 1: Initialize Parameters

Define paths for private and public keys.

Define bit length for key generation. 2048bits

Step 2: Generate RSA Key Pair

p = generate_prime(bit_length) generate prime of length 2048bit

q = generate_prime(bit_length)

n = p * q

phi = (p - 1) * (q - 1)

e = 65537 Choose a public exponent

Ensure `e` and `phi` are coprime

while gcd(e, phi) != 1:

    e = random.randint(3, phi)

d = mod_inverse(e, phi)

Save primes data (p, q, n, e, d) to a file for reference.

Step 3: Convert RSA Key Components to PEM Format

Step 4: Save Keys to Files

Save the serialized private key PEM to the private_key_path.

Save the serialized public key PEM to the public_key_path.

Step 5: Load Keys from Files

loaded_private_key = load_key_from_file(private_key_path, key_type="private")

loaded_public_key = load_key_from_file(public_key_path, key_type="public")

Step 6: Use the Loaded Keys

The loaded keys can be used for further operations as needed.

Encryption and Decryption Algorithms with ECC Equation

Encryption Process

1. Generate RSA Key Pair:

- Action: Generate a public-private key pair using RSA.

- Output: Save the public key (`public_key.pem`) and private key (`private_key.pem`) for later use.

2. Generate ECC-Based Final AES Key:

- Input: System state information (i.e. system timestamp, process id and machine id)

- Action:

The ecc equation employed would be the one below

$$y^2=x^3 + ax + b \text{ (or) } y = \sqrt{}(x^3 + ax + b)$$

Here y is the dynamic key to be generated at the end

Value of x : 32 byte or 256 bit random integer value

Co-efficient a :

(Note: first chunk is not using ECC equation for encryption hence the value of a is required from second chunk)

Value of a is fixed for Second chunk and varies for the rest while deriving ‘y’

- Encryption of Second chunk: first 16 bytes of the first chunk is used for co-efficient ‘a’
- Value of Y obtained in second chunk is used for co-efficient ‘a’

Co-efficient b :

Steps involved in generating value of ‘b’:

1. Encode the system state information (i.e. system timestamp, process id and machine id) in utf-8 encoding.
2. Concatenate the collected system state information and generate the hash of it using sha256
3. Generate the integer value of the Hash obtained and use this as the co-ordinate ‘b’

Now that the value of ‘y’ is calculated through the ecc equation, its important to use it in bytes since the value generated is integer. This integer value is encoded in base64. The hash value of encoded value is calculated and used .

Base64 is used to encode binary data (e.g., images, audio, other binary files) as ASCII text.

UTF-8 is used to encode text data (e.g., letters, numbers, symbols, emojis) as a sequence of bytes.

These conversions are required for performing operations on video data hence cannot be skipped.

3. Chunk the Video:

- Input: Path to the original video file.

- Action: Split the original video file into smaller chunks, ensuring that each chunk is approximately equal to 1MB of size and nearest full frame of video is used.

- Output: `video_chunks` - video chunks with .enc extension.

4. Encrypt the First Chunk:

- Input: `video_chunks[0]` - path to the first video chunk, `aes_key` - the AES key, and `public_key` - the public key.

- Action:

  - Read the first video chunk and encrypt it using AES with the `aes_key`. This generates tag which is required during decryption. Nonce is randomly generated 16byte (128bit) value used in encryption.

  - Use the public key to encrypt the `aes_key` (asymmetric encryption).

- Concatenate the encrypted `aes_key`, the nonce, the tag, and the encrypted first video chunk into single data.

- Calculate the hash of (the encrypted `aes_key`, the nonce, the tag, and the encrypted first video chunk) and append the same.

- Save the combined data as an encrypted file for the first video chunk. File name consists of '_part_1_'

- Output: Encrypted data for the first video chunk.

5. Encrypt Remaining Chunks:

- Input: Remaining video chunks (`video_chunks[1:]`), `aes_key` - the AES key.

- Action:

  - Iterate over each chunk:

    - Encrypt each chunk using AES with the same `aes_key` generated during encryption of first chunk.

    - Save each encrypted chunk as a separate encrypted file.

- Output: Encrypted data for each remaining video chunk.

Decryption Process

1. Read the Encrypted Data:

- Input: Path to the encrypted video chunk file.

- Action: Read the encrypted video data from the file.

- Output: `encrypted_data` - the binary data of the encrypted video chunk.

2. Extract Encrypted Components:

- Input: `encrypted_data`.

- Action: Extract the components:

  - Extract the encrypted AES key (`encrypted_aes_key`) from the first encrypted chunk.

  - Extract the AES nonce (`nonce`) from the first encrypted chunk.

- Extract the tag (`tag`) from the first encrypted chunk.

- Extract the hash (`hash`) from the first encrypted chunk.

- Extract the encrypted video data (`encrypted_video`) from the rest of `encrypted_data`.

- Output:

- `encrypted_aes_key` - encrypted AES key (only from the first chunk).

- `nonce` and `tag` - nonce and tag used in AES encryption.

- `encrypted_video` - encrypted video data, and hash

* calculate the hash of (the encrypted `aes_key`, the nonce, the tag, and the encrypted first video chunk) and compare it with the one extracted to verify the integrity

3. Read the Private Key:

- Input: Path to the private key file (`private_key_file`).

- Action: Read the private key file and deserialize it.

- Output: `private_key` - deserialized private key object.

4. Decrypt the AES Key:

- Input: `encrypted_aes_key` (only for the first chunk), and `private_key`.

- Action:

- For the first chunk, decrypt the `encrypted_aes_key` using RSA private key decryption..

- Output: `initial_aes_key` - decrypted initial AES key.

5. Regenerate the Final AES Key:

- Input: System state information, and the `initial_aes_key`.

- Action:

- Use the same ECC-based equation as in the encryption process to regenerate the final AES key from the `initial_aes_key` and system state information.

- Output: `aes_key` - the regenerated AES key.

6. Decrypt the Video Data:

- Input: `encrypted_video`, `aes_key`, `nonce`, and `tag`.
- Action: Decrypt the `encrypted_video` using AES decryption:
  - Initialize the AES cipher with the `aes_key`, `nonce`, and `tag`.
  - Decrypt the `encrypted_video` to produce decrypted video data.
- Output: `decrypted_video` - decrypted video data.

7. Combine Decrypted Video Chunks:
- Input: List of decrypted video chunks.
- Action: Combine the decrypted video chunks into a single video file using FFmpeg.
- Output: `final_video` - the reconstructed video file.

Summary:

- Encryption: Utilizes RSA and ECC equation based key derivation with system state information to generate the final AES key, encrypts video chunks using AES, and combines encrypted chunks along with hash value.

- Decryption: Extracts and decrypts components from encrypted video data, regenerates the final AES key using the ECC equation and system state information. Verifies the extracted hash with the calculated hash and then decrypts each video chunk and combines decrypted chunks into a single video file.