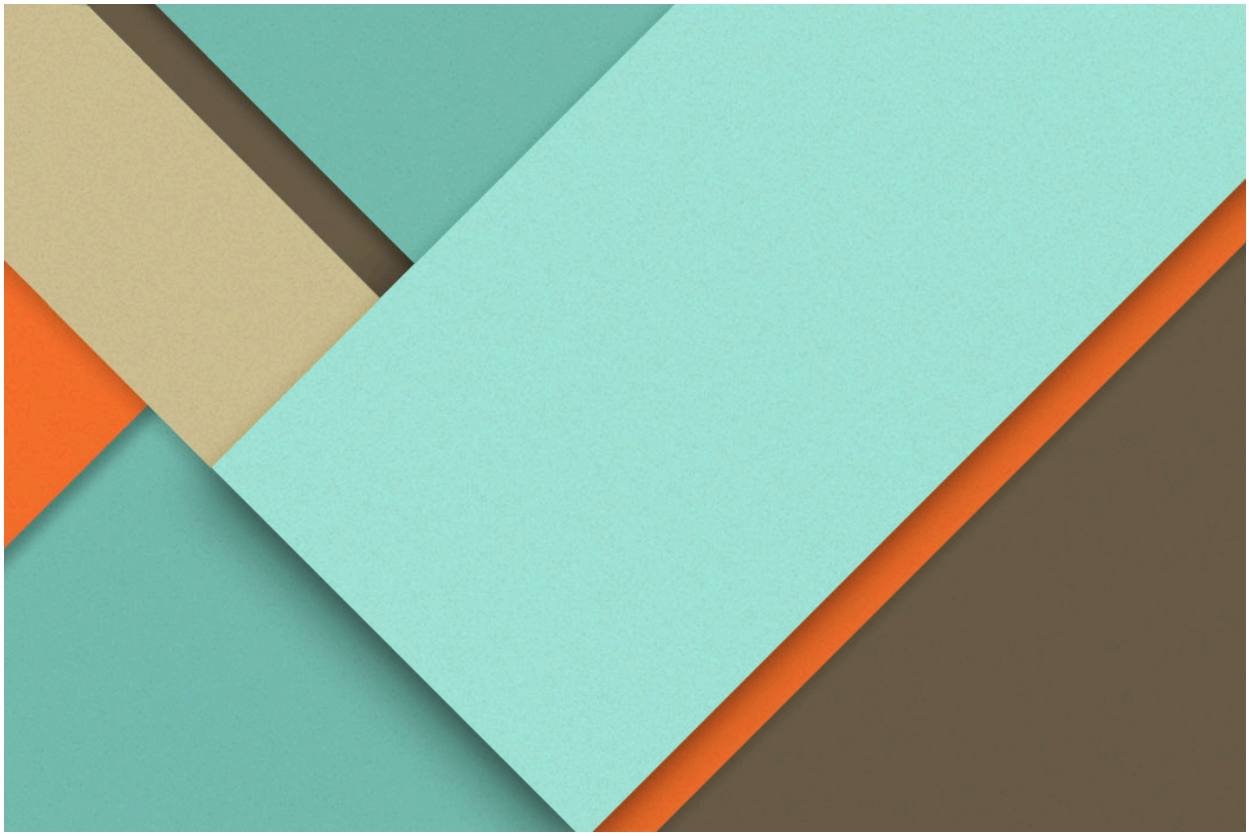


RSA-based Public-key Certification Authority (CA)

—

Parveen 2021079

Shubham Sharma 2021099

Description

This Assignment Explores the Implementation of a secure communication channel where clients can exchange messages confidentially, with the assurance of the recipient's identity, by leveraging RSA encryption and digital certificates issued by a trusted Certification Authority.

I. In our system, clients indirectly obtain each other's public keys by requesting certificates from the CA. The CA, having verified the clients' identities, issues certificates containing their public keys and then encrypts them with their private key. This ensures that only the CA can create valid certificates. Clients can then use these certificates to securely exchange messages, as they can verify the authenticity of each other's public keys through the CA's signature on the certificates.
II. We used gRPC to communicate efficiently between the CA and other clients.
III. Using the Python `time` module, we recorded issuance time and calculated certificate validity duration. During certificate verification, we compared the current time with the issuance time and duration to determine validity.

Key Generation

I. Selection of Two Large Prime Numbers (p and q)

A. Randomly select two large prime numbers, p and q.
B. Random prime numbers are generated using Python's built-in getPrime function for a given key size.

II. Calculation of n (Modulus)

A. Compute the modulus, n, by multiplying p and q.
B. $2^k < n \leq 2^{(k+1)}$; k = data size in bits.

III. Calculation of Euler's Totient Function (Φ)

A. Compute Euler's totient function, $\Phi(n)$, where $\Phi(n) = (p - 1)(q - 1)$.

IV. Selection of e (Public Exponent)

A. Choose a public exponent, e, that is relatively prime to $\Phi(n)$ and less than

Φ(n).

B. Select e, such that 1< e < Φ, and gcd(Φ, e) = 1

V. Calculation of d (Private Exponent)

A. Compute the private exponent, d = mod_inverse(e, Φ).

B. Also, (e * d) % Φ(n) = 1.

VI. RSA Public Key (e, n) and Private Key (d, n)

A. Public Key (e, n): Utilized for encryption by anyone with access to it.

B. Private Key (d, n): Kept confidential by the key owner for decryption purposes.

Encryption & Decryption

Encryption

I. Input Parameters

A. **Public Key:** Consists of two components, e and n, where e is the encryption exponent and n is the modulus.

B. **Message:** The plaintext message to be encrypted.

II. Encoding and Padding

A. The message is first encoded into bytes using *utf-8* encoding.

B. Padding may be applied to ensure the message length is a multiple of the block size to meet RSA's requirements.

III. Encryption Process

A. The message is divided into blocks of fixed length.

B. Each block is then encrypted using the RSA algorithm, where the plaintext block is raised to the power of the encryption exponent (e) modulo n.

C. The resulting ciphertext blocks are concatenated and encoded using Base64.

```
def encrypt(self, public_key, message):

    e, n = public_key
    encrypted_message = ""
    message = message.encode("utf-8")

    for i in range(0, len(message), self.block_length):
        encrypted_message += " " if encrypted_message != "" else ""
        power_raised = pow(int.from_bytes(
            pad(message[i: i + self.block_length],
                self.block_length), "big"), e, n)
        encrypted_message += b64encode(power_raised.to_bytes(
            power_raised.bit_length() // 8 + 1, "big")).decode("ascii")

    return encrypted_message
```

Decryption

I. Input Parameters

A. **Private Key:** Comprises two parts, d (decryption exponent) and n (modulus).

B. **Encrypted Message:** The ciphertext resulting from the RSA encryption process.

II. Decryption Process:

A. The encrypted message is split into individual blocks, usually separated by a space delimiter.

B. Each ciphertext block is decoded from Base64 and then decrypted using the RSA decryption algorithm. This involves raising the ciphertext block to the power of the decryption exponent (d) modulo n.

C. The resulting plaintext blocks are concatenated and unpadded to retrieve the original message.

```
def decrypt(self, private_key, encrypted_message):

    d, n = private_key
    decrypted_message = ""

    for message in encrypted_message.split(" "):
        power_raised = pow(int.from_bytes(b64decode(message), "big"), d, n)
        decrypted_message += unpad(power_raised.to_bytes(
            power_raised.bit_length() // 8 + 1, "big"),
            self.block_length).decode("utf-8")

    return decrypted_message
```

Certificate Issuance

I. Client Authentication

A. The CA receives a request for a certificate from a client identified by their unique client ID.

II. Certificate Generation

A. CA fetches the public key associated with the client ID from its records.

B. It generates a certificate containing relevant information such as CA ID, client ID, client's public key, timestamp, and validity duration.

C. The certificate data is hashed using **SHA-256** for integrity assurance, and the hash is encrypted with the CA's private key using **RSA** encryption before being appended to the certificate data.

III. Issuance and Response

A. The generated certificate, including the encrypted hash, is returned to the client as a response to the certificate request.

```python
def IssueCertificate(self, request, context):

    client_id = request.id

    if client_id in self.clients_public_keys:
        public_key = self.clients_public_keys[client_id]
        current_time = int(time.time())

        certificate_data = f"ID_CA: {self.id}, ID_CLIENT: {client_id},
        PU_CLIENT: {public_key}, TIME: {current_time}, DURATION: 3600 "
        certificate = certificate_data + self.rsa.encrypt(
            self.private_key, hashlib.sha256(certificate_data.encode()
                                             ).hexdigest())

        return CA_pb2.Certificate(certificate=certificate)

    else:
        return CA_pb2.Certificate(certificate="Client not Found!")
```

Client & CA Working

I. Initially, the CA verifies the client's identity and issues a certificate with the public key, issuance time, and validity duration.

II. Before sending a message, the client fetches the recipient's certificate from CA.

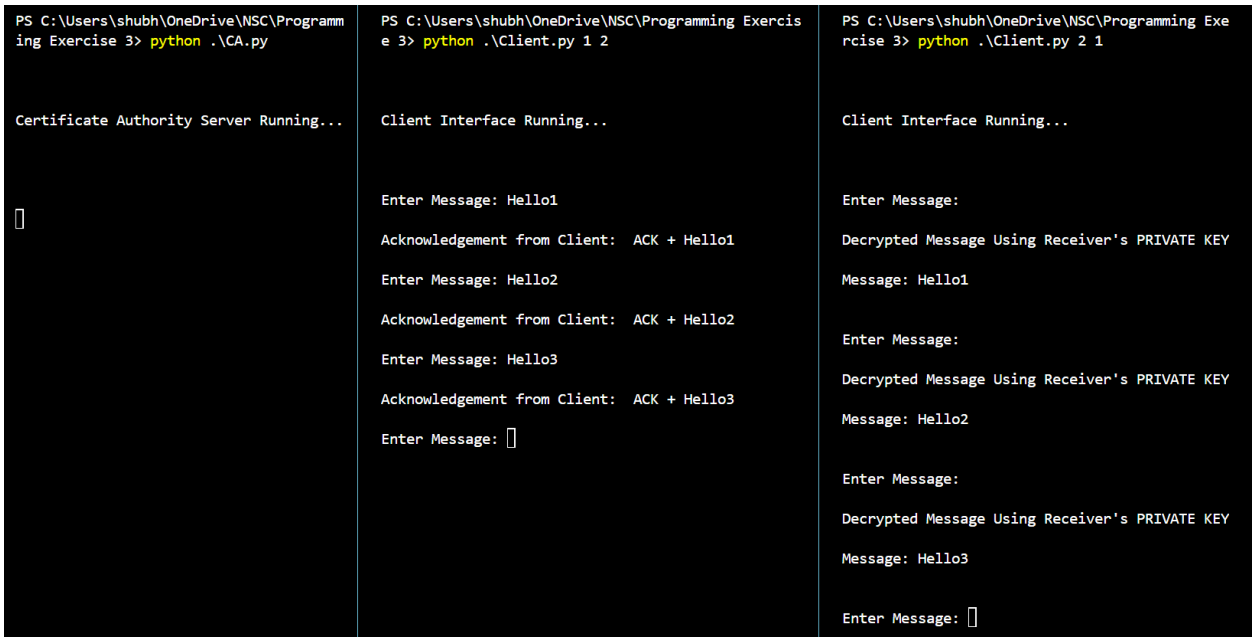
III. The client verifies the recipient's certificate integrity authenticity by correct decryption & comparing the hash value of the certificate, and also verifies the time validity.

IV. If valid, the client extracts the recipient's public key from the certificate.

V. When sending messages, the client uses a pre-fetched recipient's certificate or fetches a new one from CA.

VI. This ensures secure, authenticated communication with maintained certificate integrity and validity.

Results


PS C:\Users\shubh\OneDrive\NSC\Programming Exercise 3> python .\CA.py
Certificate Authority Server Running...
PS C:\Users\shubh\OneDrive\NSC\Programming Exercise 3> python .\Client.py 1 2
Client Interface Running...
Enter Message: Hello1
Acknowledgement from Client: ACK + Hello1
Enter Message: Hello2
Acknowledgement from Client: ACK + Hello2
Enter Message: Hello3
Acknowledgement from Client: ACK + Hello3
Enter Message:
PS C:\Users\shubh\OneDrive\NSC\Programming Exercise 3> python .\Client.py 2 1
Client Interface Running...
Enter Message:
Decrypted Message Using Receiver's PRIVATE KEY
Message: Hello1
Enter Message:
Decrypted Message Using Receiver's PRIVATE KEY
Message: Hello2
Enter Message:
Decrypted Message Using Receiver's PRIVATE KEY
Message: Hello3
Enter Message: