CSE350-
NETWORK SECURITY
ASSIGNMENT NO. 3

Data Encryption Standard

By –Angadjeet (2022071)
Arav amawate(2022091)

Introduction

Objective:

- Establish a secure communication environment using RSA encryption.
- Create a trusted CA to issue and verify public-key certificates.

Key Components:

- Certification Authority (CA) Server
- Client Applications (Client A and Client B)
- RSA Cryptography Module (Key generation, encryption, decryption)

Functionality:

- Clients register with the CA.
- CA issues digitally signed certificates.
- Clients exchange secure messages using RSA encryption.

Client Code

(client.py)

- Purpose:
  - Implements client-side operations for registration, certificate requests, and secure message exchange.
- Key Methods:
  - register(): Registers the client with the CA and receives the CA's public key.
  - request_certificate(): Requests a certificate from the CA for the client's public key.
  - get_peer_certificate(peer_id): Retrieves a peer's certificate from the CA.
  - validate_certificate(certificate): Validates a received certificate by contacting the CA.
  - send_message(recipient_public_key, message): Encrypts a message using the recipient's public key.
  - receive_message(encrypted_message): Decrypts an incoming encrypted message using the client's private key.
- Workflow:
  - The client connects to the CA using sockets.
  - Data is serialized using pickle for transmission.
  - Secure message exchange is enabled once certificates are validated.

```
class CAClient:
    def __init__(self, server_host, server_port, client_id, public_key, private_key, ca_public_key):
        self.server_host = server_host
        self.server_port = server_port
        self.client_id = client_id
        self.public_key = public_key
        self.private_key = private_key
        self.ca_public_key = ca_public_key
        self.rsa = RSA(1024)
        self.certificate = None
```

CA Code

(ca.py)

- Purpose:
  - Acts as the trusted authority that manages client public keys and issues/validates certificates.
- Key Components:
  - Certification Authority Class:
    - Registers clients and stores their public keys.
    - Signs certificates by hashing certificate data and encrypting the hash with the CA's private key.
    - Verifies certificates by decrypting the signature and comparing hashes.
  - CA Server:
    - Listens for client requests over TCP using socket programming.
    - Handles registration, certificate signing, certificate retrieval, and verification.
- Data Flow:
  - Client sends a request (e.g., register, sign, get_certificate, verify).
  - CA processes the request, performs cryptographic operations, and sends back a response.

```python
# Certification Authority: stores client public keys and issues certificates.
class CertificationAuthority:
    def __init__(self):
        self.rsa = RSA(1024)
        self.ca_public_key, self.ca_private_key = self.rsa.generate_keys()
        self.client_public_keys = {}
        self.certificates = {}

    def register_client(self, client_id, client_public_key):
        """Register a client by storing its public key."""
        self.client_public_keys[client_id] = client_public_key
        print(f"[CA] Registered client {client_id} with public key: {client_public_key}")
        return "Client registered successfully."

    def sign_certificate(self, client_id, duration=600):
        if client_id not in self.client_public_keys:
            return None, "Client not registered"

        public_key = self.client_public_keys[client_id]
        issue_time = int(time.time())
        certificate_data = f"ClientID: {client_id}, PublicKey: {public_key}, IssueTime: {issue_time}, Duration: {duration}"
        certificate_hash = hashlib.sha256(certificate_data.encode()).digest()
        hash_int = int.from_bytes(certificate_hash, byteorder='big')
        encrypted_signature = self.rsa.encrypt(str(hash_int), self.ca_public_key)

        certificate = {
            "client_id": client_id,
            "public_key": public_key,
            "issue_time": issue_time,
            "duration": duration,
            "signature": encrypted_signature
        }
        self.certificates[client_id] = certificate
        print(f"[CA] Issued certificate for client {client_id}: {certificate}")
        return certificate, "Certificate Issued"

    def verify_certificate(self, client_id):
        if client_id not in self.certificates:
            return False, "Certificate not found!"
```

Encryption RSA

- RSA Module Functionality:
- Key Generation:
  - Generates two distinct primes, computes nnn and Euler’s Totient ϕ.
  - Determines public exponent e(coprime with ϕ) and computes private key ddd.
- Encryption Process:
  - Input: Plaintext string and recipient's public key (e,n).
  - Block Processing:
    - Splits plaintext into blocks based on block size (with room for padding overhead).
    - Applies PKCS#1-like padding to each block.
    - Converts padded block to an integer.
    - Encrypts each block using modular exponentiation: encrypted_block=block^emod n
    - Encodes encrypted block to Base64 for transmission.
- Key Points:
- Ensures confidentiality by using the recipient's public key.
- Each block is encrypted individually to handle larger messages.

```
def encrypt(self, plaintext, public_key):
    """Encrypt plaintext (str) with the given public key (block-wise)."""
    e, n = public_key
    ciphertext = ""
    # For RSA with PKCS#1-like padding, we leave 11 bytes for overhead
    for i in range(0, len(plaintext), self.block_size - 11):
        block = plaintext[i : i + self.block_size - 11]
        block_bytes = block.encode('utf-8')
        block_bytes = pad(block_bytes, self.block_size)
        block_int = int.from_bytes(block_bytes, byteorder='big')
        encrypted_block = pow(block_int, e, n)
        # Convert to base64
        enc_b64 = b64encode(
            encrypted_block.to_bytes((encrypted_block.bit_length() + 7) // 8, byteorder='big')
        ).decode('utf-8')
        ciphertext += enc_b64 + " "
    return ciphertext.strip()
```

Decryption RSA

- RSA Module Functionality (Continued):

Decryption Process:

- Input: Encrypted message (Base64 string) and recipient's private key (d,n).

Block Processing:

- Splits the Base64 encoded message into individual blocks.
- Decodes each block from Base64 to get the encrypted integer.
- Decrypts using modular exponentiation: decrypted_block=encrypted_block^d mod n
- Converts the decrypted integer back to a byte string of fixed block size.
- Removes PKCS#1-like padding to retrieve the original plaintext block.
- Key Points:
- Private key is used to decrypt messages, ensuring that only the intended recipient can read the content.
- Correct unpadding is critical to recover the exact original message

```python
def decrypt(self, ciphertext, private_key):
    """Decrypt ciphertext (str) with the given private key (block-wise)."""
    d, n = private_key  # Use the provided private key
    decrypted_text = ""
    blocks = ciphertext.strip().split()
    for block in blocks:
        block_bytes = b64decode(block)
        block_int = int.from_bytes(block_bytes, byteorder='big')
        decrypted_block_int = pow(block_int, d, n)
        # Convert to fixed block size using self.block_size
        fixed_bytes = decrypted_block_int.to_bytes(self.block_size, byteorder='big')
        unpadded = unpad(fixed_bytes, self.block_size)
        decrypted_text += unpadded.decode('utf-8')
    return decrypted_text
```

Output code

```python
def test_message_exchange():
    # Assume the CA server is already running on 127.0.0.1:50051.
    ca_public_key = None

    # Generate RSA keys for Client A and Client B.
    rsa_A = RSA(1024)
    public_key_A, private_key_A = rsa_A.generate_keys()

    rsa_B = RSA(1024)
    public_key_B, private_key_B = rsa_B.generate_keys()

    # Instantiate Client A and Client B.
    clientA = CAClient("127.0.0.1", 50051, "A", public_key_A, private_key_A, ca_public_key)
    clientB = CAClient("127.0.0.1", 50051, "B", public_key_B, private_key_B, ca_public_key)

    # Use the same RSA instance that generated the keys.
    clientA.rsa = rsa_A
    clientB.rsa = rsa_B

    # Register both clients with the CA.
    clientA.register()
    clientB.register()

    time.sleep(0.5)

    # Each client requests a certificate from the CA.
    clientA.request_certificate()
    clientB.request_certificate()
    time.sleep(0.5)

    # Each client retrieves its peer's certificate.
    cert_for_A = clientA.get_peer_certificate("B")
    cert_for_B = clientB.get_peer_certificate("A")

    # Validate the certificates by asking the CA server to verify them.
    if not cert_for_A or not clientA.validate_certificate(cert_for_A):
        print("Peer certificate validation failed for Client B.")
        return
    if not cert_for_B or not clientB.validate_certificate(cert_for_B):
        print("Peer certificate validation failed for Client A.")
        return

    # Extract the public keys from the verified certificates.
    peer_public_key_for_A = cert_for_A["public_key"]
    peer_public_key_for_B = cert_for_B["public_key"]

    print(f"Client A's public key: {public_key_A}")
    print(f"Client B's public key: {public_key_B}")

    # Simulate the exchange of three messages based on the verified certificates.
    for i in range(1, 4):
        message = f"Hello{i}"
        print(f"\nClient A sending: {message}")
        encrypted_message = clientA.send_message(peer_public_key_for_A, message)
        received_message = clientB.receive_message(encrypted_message)
        print(f"Client B received: {received_message}")

        ack_message = f"ACK{i}"
        print(f"Client B sending: {ack_message}")
        encrypted_ack = clientB.send_message(peer_public_key_for_B, ack_message)
        received_ack = clientA.receive_message(encrypted_ack)
        print(f"Client A received: {received_ack}")

if __name__ == "__main__":
    test_message_exchange()
```

✓ 1.2s

```
Client A registration successful.
Client B registration successful.
Certificate issued for Client A: {'client_id': 'A', 'public_key': (475879853337255831884148157708
Certificate issued for Client B: {'client_id': 'B', 'public_key': (828310720727932943818954423393
Client A's public key: (475879853337255831884148157708345670016491226042609684867140761145623543
Client B's public key: (828310720727932943818954423393439041233741742543784729942603089475107017

Client A sending: Hello1
Client B received: Hello1
Client B sending: ACK1
Client A received: ACK1

Client A sending: Hello2
Client B received: Hello2
Client B sending: ACK2
Client A received: ACK2

Client A sending: Hello3
Client B received: Hello3
Client B sending: ACK3
Client A received: ACK3
```

- : Test File & Output

Test Script Overview:

Setup:

- RSA keys are generated separately for Client A and Client B.
- Both clients register with the CA and request their certificates.

Certificate Exchange:

- Clients retrieve each other's certificates from the CA.
- Certificates are validated to ensure authenticity.

Message Exchange:

- Client A sends test messages ("Hello1", "Hello2", "Hello3") encrypted with Client B's public key.
- Client B decrypts the messages, prints the output, and sends acknowledgment messages ("ACK1", "ACK2", "ACK3") encrypted with Client A's public key.

Expected Output:

- Display of each client's public key.
- Sequential logs showing message encryption, transmission, decryption, and acknowledgments.

Highlights:

- Demonstrates end-to-end secure communication.
- Validates the integrity of registration, certificate exchange, RSA encryption, and decryption in a real-world scenario.

THANKS!