**Practical: 09**

**AIM:** **Show a practical scenario of Key Distribution. Use the separate key-sharing server that shares the secret key created using the AES-256-bit algorithm and share the secret key using the RSA algorithm with 1024/2048-bit key size. The key-sharing server produces a new secret key for each new communication between two nodes.**

**CODE:**

from cryptography import x509

from cryptography.x509.oid import NameOID

from cryptography.hazmat.primitives import hashes, serialization

from cryptography.hazmat.primitives.asymmetric import rsa, padding

from cryptography.hazmat.primitives.serialization import load\_pem\_private\_key

import datetime

import os

import base64

import json

from flask import Flask, request, jsonify

# Certificate Authority (CA) setup

class CertificateAuthority:

def \_\_init\_\_(self):

# Generate CA's private key

self.private\_key = rsa.generate\_private\_key(

public\_exponent=65537,

key\_size=2048

)

# Generate CA's self-signed certificate

self.cert = x509.CertificateBuilder().subject\_name(

x509.Name([

x509.NameAttribute(NameOID.COMMON\_NAME, u"Certificate Authority"),

x509.NameAttribute(NameOID.ORGANIZATION\_NAME, u"Secure CA Inc."),

x509.NameAttribute(NameOID.COUNTRY\_NAME, u"US"),

])

).issuer\_name(

x509.Name([

x509.NameAttribute(NameOID.COMMON\_NAME, u"Certificate Authority"),

x509.NameAttribute(NameOID.ORGANIZATION\_NAME, u"Secure CA Inc."),

x509.NameAttribute(NameOID.COUNTRY\_NAME, u"US"),

])

).not\_valid\_before(

datetime.datetime.utcnow()

).not\_valid\_after(

datetime.datetime.utcnow() + datetime.timedelta(days=365)

).serial\_number(

x509.random\_serial\_number()

).public\_key(

self.private\_key.public\_key()

).add\_extension(

x509.BasicConstraints(ca=True, path\_length=None), critical=True

).add\_extension(

x509.KeyUsage(

digital\_signature=True,

content\_commitment=False,

key\_encipherment=False,

data\_encipherment=False,

key\_agreement=False,

key\_cert\_sign=True,

crl\_sign=True,

encipher\_only=False,

decipher\_only=False

), critical=True

).sign(self.private\_key, hashes.SHA256())

print("Certificate Authority initialized with self-signed certificate")

def issue\_certificate(self, client\_name, client\_public\_key):

"""Issue a certificate for the client"""

client\_cert = x509.CertificateBuilder().subject\_name(

x509.Name([

x509.NameAttribute(NameOID.COMMON\_NAME, client\_name),

x509.NameAttribute(NameOID.ORGANIZATION\_NAME, u"Client Organization"),

x509.NameAttribute(NameOID.COUNTRY\_NAME, u"US"),

])

).issuer\_name(

self.cert.subject

).public\_key(

client\_public\_key

).serial\_number(

x509.random\_serial\_number()

).not\_valid\_before(

datetime.datetime.utcnow()

).not\_valid\_after(

datetime.datetime.utcnow() + datetime.timedelta(days=365)

).add\_extension(

x509.BasicConstraints(ca=False, path\_length=None), critical=True

).add\_extension(

x509.KeyUsage(

digital\_signature=True,

content\_commitment=True,

key\_encipherment=True,

data\_encipherment=False,

key\_agreement=False,

key\_cert\_sign=False,

crl\_sign=False,

encipher\_only=False,

decipher\_only=False

), critical=True

).sign(self.private\_key, hashes.SHA256())

print(f"Certificate issued for {client\_name}")

return client\_cert

def get\_ca\_certificate\_pem(self):

"""Return the CA certificate in PEM format"""

return self.cert.public\_bytes(serialization.Encoding.PEM)

# Digital Signature Server

class SignatureServer:

def \_\_init\_\_(self):

self.ca = CertificateAuthority()

self.clients = {} # Store client certificates and public keys

print("Digital Signature Server initialized")

def register\_client(self, client\_name):

"""Register a new client and generate their certificate"""

# Generate client's private key

client\_private\_key = rsa.generate\_private\_key(

public\_exponent=65537,

key\_size=2048

)

# Get client's public key

client\_public\_key = client\_private\_key.public\_key()

# Issue certificate for the client

client\_cert = self.ca.issue\_certificate(client\_name, client\_public\_key)

# Store client information

self.clients[client\_name] = {

'certificate': client\_cert,

'public\_key': client\_public\_key

}

# Serialize private key for client

private\_key\_pem = client\_private\_key.private\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PrivateFormat.PKCS8,

encryption\_algorithm=serialization.NoEncryption()

)

# Serialize certificate for client

cert\_pem = client\_cert.public\_bytes(serialization.Encoding.PEM)

print(f"Client {client\_name} registered successfully")

# Return the client's certificate and private key

return {

'certificate': cert\_pem.decode('utf-8'),

'private\_key': private\_key\_pem.decode('utf-8'),

'ca\_certificate': self.ca.get\_ca\_certificate\_pem().decode('utf-8')

}

# Client implementation

class Client:

def \_\_init\_\_(self, client\_name, certificate\_pem, private\_key\_pem, ca\_certificate\_pem):

self.name = client\_name

self.certificate = x509.load\_pem\_x509\_certificate(certificate\_pem.encode('utf-8'))

self.private\_key = load\_pem\_private\_key(

private\_key\_pem.encode('utf-8'),

password=None

)

self.ca\_certificate = x509.load\_pem\_x509\_certificate(ca\_certificate\_pem.encode('utf-8'))

print(f"Client {client\_name} initialized with certificate")

def sign\_document(self, document):

"""Sign a document with the client's private key"""

# Calculate document hash

document\_bytes = document.encode('utf-8')

# Sign the document

signature = self.private\_key.sign(

document\_bytes,

padding.PSS(

mgf=padding.MGF1(hashes.SHA256()),

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA256()

)

# Create signed document structure

signed\_document = {

'document': document,

'document\_hash': base64.b64encode(hashes.Hash(hashes.SHA256()).update(document\_bytes).finalize()).decode('utf-8'),

'signature': base64.b64encode(signature).decode('utf-8'),

'signer': {

'name': self.name,

'certificate': self.certificate.public\_bytes(serialization.Encoding.PEM).decode('utf-8')

},

'timestamp': datetime.datetime.utcnow().isoformat()

}

print(f"Document signed by {self.name}")

return signed\_document

def verify\_signed\_document(self, signed\_document):

"""Verify a signed document"""

# Extract document and signature

document = signed\_document['document']

document\_bytes = document.encode('utf-8')

signature = base64.b64decode(signed\_document['signature'])

# Extract signer's certificate

signer\_cert\_pem = signed\_document['signer']['certificate']

signer\_cert = x509.load\_pem\_x509\_certificate(signer\_cert\_pem.encode('utf-8'))

signer\_public\_key = signer\_cert.public\_key()

# Verify certificate against CA

try:

# In a real system, we would perform a proper certificate validation chain

# For simplicity, we're just checking that the issuer matches our CA

if signer\_cert.issuer != self.ca\_certificate.subject:

print("Certificate issuer validation failed")

return False, "Invalid certificate issuer"

# Check certificate expiration

now = datetime.datetime.utcnow()

if now < signer\_cert.not\_valid\_before or now > signer\_cert.not\_valid\_after:

print("Certificate expired or not yet valid")

return False, "Certificate not valid at current time"

except Exception as e:

print(f"Certificate validation error: {e}")

return False, f"Certificate validation error: {e}"

# Verify the signature

try:

signer\_public\_key.verify(

signature,

document\_bytes,

padding.PSS(

mgf=padding.MGF1(hashes.SHA256()),

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA256()

)

print("Signature verification successful")

# Verify document hash

calculated\_hash = base64.b64encode(hashes.Hash(hashes.SHA256()).update(document\_bytes).finalize()).decode('utf-8')

if calculated\_hash != signed\_document['document\_hash']:

print("Document hash verification failed")

return False, "Document integrity check failed"

print("Document integrity verified")

return True, "Document signature and integrity verified successfully"

except Exception as e:

print(f"Signature verification error: {e}")

return False, f"Signature verification error: {e}"

# Create a Flask web server to demonstrate the system

app = Flask(\_\_name\_\_)

signature\_server = SignatureServer()

@app.route('/register', methods=['POST'])

def register\_client():

data = request.json

client\_name = data.get('client\_name', 'Unknown Client')

client\_data = signature\_server.register\_client(client\_name)

return jsonify({

'status': 'success',

'message': f'Client {client\_name} registered successfully',

'data': client\_data

})

@app.route('/sign', methods=['POST'])

def sign\_document():

data = request.json

client\_name = data.get('client\_name')

document = data.get('document')

certificate\_pem = data.get('certificate')

private\_key\_pem = data.get('private\_key')

ca\_certificate\_pem = data.get('ca\_certificate')

if not all([client\_name, document, certificate\_pem, private\_key\_pem, ca\_certificate\_pem]):

return jsonify({

'status': 'error',

'message': 'Missing required parameters'

}), 400

try:

client = Client(client\_name, certificate\_pem, private\_key\_pem, ca\_certificate\_pem)

signed\_document = client.sign\_document(document)

return jsonify({

'status': 'success',

'message': 'Document signed successfully',

'signed\_document': signed\_document

})

except Exception as e:

return jsonify({

'status': 'error',

'message': f'Error signing document: {str(e)}'

}), 500

@app.route('/verify', methods=['POST'])

def verify\_document():

data = request.json

signed\_document = data.get('signed\_document')

ca\_certificate\_pem = data.get('ca\_certificate')

if not all([signed\_document, ca\_certificate\_pem]):

return jsonify({

'status': 'error',

'message': 'Missing required parameters'

}), 400

try:

# Create a temporary client just for verification

# In a real system, we would have proper user sessions

temp\_client = Client(

"Verifier",

signed\_document['signer']['certificate'],

"", # No private key needed for verification

ca\_certificate\_pem

)

is\_valid, message = temp\_client.verify\_signed\_document(signed\_document)

return jsonify({

'status': 'success' if is\_valid else 'error',

'message': message,

'verification\_result': {

'is\_valid': is\_valid,

'authenticity': is\_valid,

'non\_repudiation': is\_valid,

'integrity': is\_valid,

'signer': signed\_document['signer']['name'],

'timestamp': signed\_document['timestamp']

}

})

except Exception as e:

return jsonify({

'status': 'error',

'message': f'Error verifying document: {str(e)}'

}), 500

# Usage demonstration

def run\_demo():

# 1. Set up the server and register a client

signature\_server = SignatureServer()

client\_data = signature\_server.register\_client("Alice")

# 2. Initialize the client with the received credentials

alice = Client(

"Alice",

client\_data['certificate'],

client\_data['private\_key'],

client\_data['ca\_certificate']

)

# 3. Alice signs a document

document = "This is a confidential transaction between Alice and Bob for $1000."

signed\_document = alice.sign\_document(document)

# 4. Bob (or any recipient) verifies the document

bob = Client(

"Bob",

client\_data['certificate'], # Bob doesn't need Alice's certificate for initialization

"", # Bob doesn't need Alice's private key for verification

client\_data['ca\_certificate']

)

# 5. Verify the signed document

is\_valid, message = bob.verify\_signed\_document(signed\_document)

print("\nVerification Results:")

print(f"Valid: {is\_valid}")

print(f"Message: {message}")

if is\_valid:

print("\nThe document passed all three security checks:")

print("1. Authentication: The document was indeed created by Alice")

print("2. Non-repudiation: Alice cannot deny having signed this document")

print("3. Integrity: The document has not been altered after signing")

# 6. Demonstrate tampering (modify the document)

print("\nTesting tampered document:")

tampered\_document = signed\_document.copy()

tampered\_document['document'] = "This is a confidential transaction between Alice and Bob for $2000."

is\_valid, message = bob.verify\_signed\_document(tampered\_document)

print(f"Valid: {is\_valid}")

print(f"Message: {message}")

if \_\_name\_\_ == "\_\_main\_\_":

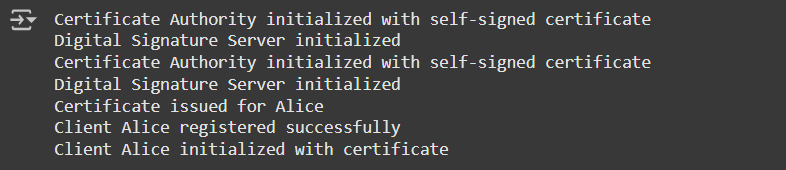
# Run the web server

# app.run(debug=True)

# Or run the demonstration

run\_demo()

**OUTPUT:**



# **LATEST APPLICATIONS:**

· **Blockchain Smart Contracts** - Digital signatures authenticate transactions and automate contract execution on blockchain networks.

· **Remote Work Document Workflows** - Organizations use digital signature systems for paperless approval processes across distributed teams.

· **Government Digital ID Systems** - National digital identity programs use signatures for citizen authentication in online government services.

· **Healthcare Records Authentication** - Medical systems use digital signatures to ensure patient record integrity and physician accountability.

**LEARNING OUTCOME:**

* Understand the three core properties of digital signatures: authentication, non-repudiation, and integrity
* Recognize the role of certificate authorities in establishing trust infrastructure
* Explain the relationship between public key infrastructure (PKI) and digital signatures
* Identify potential vulnerabilities in digital signature implementations
* Apply appropriate signature algorithms based on security requirements