**Practical: 08**

**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:**

import os

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

from cryptography.hazmat.primitives import hashes, serialization

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

import base64

import time

class Node:

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

self.name = name

# Generate RSA key pair for this node

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

public\_exponent=65537,

key\_size=2048,

)

self.public\_key = self.private\_key.public\_key()

self.session\_keys = {} # Store session keys for different communications

def get\_public\_key\_pem(self):

"""Return public key in PEM format"""

return self.public\_key.public\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo

)

def decrypt\_session\_key(self, encrypted\_key):

"""Decrypt a session key that was encrypted with this node's public key"""

decrypted\_key = self.private\_key.decrypt(

encrypted\_key,

padding.OAEP(

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

algorithm=hashes.SHA256(),

label=None

)

)

return decrypted\_key

def encrypt\_message(self, message, other\_node\_name):

"""Encrypt a message using the session key for communication with other\_node"""

if other\_node\_name not in self.session\_keys:

raise ValueError(f"No session key established with {other\_node\_name}")

session\_key = self.session\_keys[other\_node\_name]

# Generate a random IV

iv = os.urandom(16)

# Encrypt with AES-256 in CBC mode

encryptor = Cipher(

algorithms.AES(session\_key),

modes.CBC(iv)

).encryptor()

# Pad the message to be a multiple of 16 bytes

padded\_message = message + b' ' \* (16 - len(message) % 16)

ciphertext = encryptor.update(padded\_message) + encryptor.finalize()

# Return IV + ciphertext

return iv + ciphertext

def decrypt\_message(self, encrypted\_data, other\_node\_name):

"""Decrypt a message using the session key for communication with other\_node"""

if other\_node\_name not in self.session\_keys:

raise ValueError(f"No session key established with {other\_node\_name}")

session\_key = self.session\_keys[other\_node\_name]

# Extract IV (first 16 bytes)

iv = encrypted\_data[:16]

ciphertext = encrypted\_data[16:]

# Decrypt with AES-256 in CBC mode

decryptor = Cipher(

algorithms.AES(session\_key),

modes.CBC(iv)

).decryptor()

plaintext = decryptor.update(ciphertext) + decryptor.finalize()

return plaintext.rstrip() # Remove padding

class KeySharingServer:

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

self.node\_public\_keys = {} # Store public keys of registered nodes

self.session\_counter = 0

def register\_node(self, node\_name, public\_key\_pem):

"""Register a node's public key with the server"""

self.node\_public\_keys[node\_name] = serialization.load\_pem\_public\_key(public\_key\_pem)

print(f"Node {node\_name} registered with the key server")

def create\_secure\_channel(self, node1\_name, node2\_name):

"""Generate a new AES-256 session key and distribute it to both nodes securely"""

if node1\_name not in self.node\_public\_keys or node2\_name not in self.node\_public\_keys:

raise ValueError("Both nodes must be registered with the key server")

# Generate a new AES-256 key

session\_key = os.urandom(32) # 256 bits = 32 bytes

self.session\_counter += 1

session\_id = f"session\_{self.session\_counter}\_{int(time.time())}"

# Encrypt the session key with each node's public key

encrypted\_key\_node1 = self.node\_public\_keys[node1\_name].encrypt(

session\_key,

padding.OAEP(

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

algorithm=hashes.SHA256(),

label=None

)

)

encrypted\_key\_node2 = self.node\_public\_keys[node2\_name].encrypt(

session\_key,

padding.OAEP(

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

algorithm=hashes.SHA256(),

label=None

)

)

print(f"New AES-256 session key generated for communication between {node1\_name} and {node2\_name}")

# Return the encrypted keys to be distributed to each node

return {

"session\_id": session\_id,

node1\_name: encrypted\_key\_node1,

node2\_name: encrypted\_key\_node2

}

# Simulation of the key distribution scenario

def simulate\_key\_distribution():

# Create a key sharing server

key\_server = KeySharingServer()

# Create two nodes that want to communicate

node\_a = Node("Node\_A")

node\_b = Node("Node\_B")

# Register nodes with the key server

key\_server.register\_node(node\_a.name, node\_a.get\_public\_key\_pem())

key\_server.register\_node(node\_b.name, node\_b.get\_public\_key\_pem())

# Key server creates a secure channel by generating and distributing a session key

encrypted\_keys = key\_server.create\_secure\_channel(node\_a.name, node\_b.name)

# Nodes receive their encrypted session keys and decrypt them

node\_a.session\_keys[node\_b.name] = node\_a.decrypt\_session\_key(encrypted\_keys[node\_a.name])

node\_b.session\_keys[node\_a.name] = node\_b.decrypt\_session\_key(encrypted\_keys[node\_b.name])

print(f"Session established: {encrypted\_keys['session\_id']}")

# Now nodes can communicate securely using the shared session key

message = b"Hello, this is a secure message using our shared AES-256 key."

encrypted\_message = node\_a.encrypt\_message(message, node\_b.name)

print(f"\nNode A encrypts: {message.decode()}")

print(f"Encrypted data size: {len(encrypted\_message)} bytes")

# Node B receives and decrypts the message

decrypted\_message = node\_b.decrypt\_message(encrypted\_message, node\_a.name)

print(f"Node B decrypts: {decrypted\_message.decode()}")

# For a new communication session, a new key is generated

print("\n--- Starting new communication session ---")

new\_encrypted\_keys = key\_server.create\_secure\_channel(node\_a.name, node\_b.name)

# Nodes receive their new encrypted session keys and decrypt them

node\_a.session\_keys[node\_b.name] = node\_a.decrypt\_session\_key(new\_encrypted\_keys[node\_a.name])

node\_b.session\_keys[node\_a.name] = node\_b.decrypt\_session\_key(new\_encrypted\_keys[node\_b.name])

print(f"New session established: {new\_encrypted\_keys['session\_id']}")

# Test with a different message

new\_message = b"This message uses the newly generated session key."

new\_encrypted\_message = node\_a.encrypt\_message(new\_message, node\_b.name)

new\_decrypted\_message = node\_b.decrypt\_message(new\_encrypted\_message, node\_a.name)

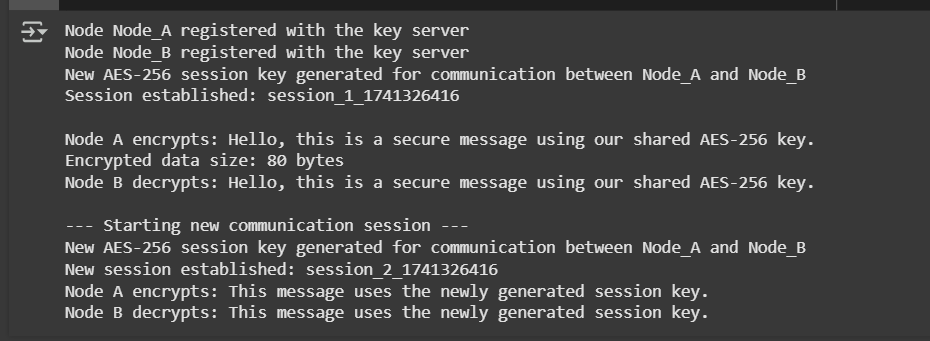
print(f"Node A encrypts: {new\_message.decode()}")

print(f"Node B decrypts: {new\_decrypted\_message.decode()}")

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

simulate\_key\_distribution()

**OUTPUT:**



# **LATEST APPLICATIONS:**

 **Zero Trust Security Models** - Organizations use modern key distribution to validate every user and device before granting access, even those inside the network.

 **IoT Device Networks** - Smart home and industrial IoT systems use key distribution to securely connect numerous low-power devices.

 **Quantum Key Distribution (QKD)** - Using quantum properties to detect eavesdropping when sharing encryption keys between locations.

 **Blockchain Technology** - Cryptocurrencies and decentralized applications use sophisticated key distribution methods to secure transactions.

 **Multi-Cloud Security** - Enterprises use key distribution systems to maintain secure connections across different cloud environments.

**LEARNING OUTCOME:**

* Explain the difference between symmetric and asymmetric key distribution techniques
* Understand how trusted third parties facilitate secure key sharing
* Recognize the importance of key rotation and session-based encryption
* Identify potential vulnerabilities in key distribution systems