



**P P SAVANI UNIVERSITY**  
**ACADEMIC YEAR-2025-26**

**Assignment No. - 5**  
**ON**  
**BLOCKCHAIN TECHNOLOGY(SSCS3021)**

**TITLE: Develop and Implement a Transaction, Hash It, Generate Public and Private Keys, Digitally Sign and Verify the Transaction**

**BACHELOR OF SCIENCE IN INFORMATION TECHNOLOGY (BSC-IT)**

**SUBMITTED TO:**

**Name: KAUSHAL SINGH(KSV)**

**Designation: ASSISTANT PROFESSOR**

**P P Savani University**

**SUBMITTED BY:**

**Name: RAJ MO FAHIM ZAKIR**

**Enrollment: 23SS02IT161**

**BSCIT5B-Batch 2023-26**

**Max. Marks: 50**  
**Marks Obtained:**

**Faculty Signature: \_\_\_\_\_**

**INSTITUTE OF COMPUTER SCIENCE AND APPLICATIONS**  
**P P SAVANI UNIVERSITY**  
**MANGROL, SURAT- 394125 (GUJARAT)**



**Student Name: RAJ MO FAHIM ZAKIR**  
**Enrolment Number: 23SS02IT161**  
**Subject Name: BLOCKCHAIN TECHNOLOGY**  
**Subject Code: SSCS3021**

## **Practical-5**

**Date:23/07/2025**

**Aim:** Develop and Implement a Transaction, Hash It, Generate Public and Private Keys, Digitally Sign and Verify the Transaction.

### **Practical Questions :**

#### **Q1. Tamper Detection Test**

**Objective:** Modify a signed transaction and verify the old signature on the tampered version.

```
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa, padding
# Key generation
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
public_key = private_key.public_key()
# Original transaction
transaction = "Send 5 BTC to Alice"
transaction_bytes = transaction.encode()
# Signing the transaction
signature = private_key.sign(
    transaction_bytes,
    padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH),
    hashes.SHA256()
)
# Tampered transaction
tampered_transaction = "Send 10 BTC to Alice"
tampered_bytes = tampered_transaction.encode()

# Attempt verification with tampered data
try:
    public_key.verify(
        signature,
        tampered_bytes,
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH),
        hashes.SHA256()
    )
    print("Tampered transaction verified (unexpected).")
except Exception as e:
    print("Tampered transaction verification failed:", e)
```

### Output:

```
Tampered transaction verification failed:  
InvalidSignature()
```

### Observation:

Changing even a small part of the transaction (amount/receiver) results in verification failure. This demonstrates the immutability and integrity enforced by digital signatures.

## Q2. Signature Length and Encoding

**Objective:** Understand signature formats.

### Code:-

```
import base64  
  
print("Signature length (bytes):", len(signature))  
base64_sig = base64.b64encode(signature).decode()  
print("Base64 Signature:", base64_sig)
```

### Output:

```
Signature length (bytes): 256  
Base64 Signature: pAvv0S4uCMmCzX+1gB30T4K6... (trimmed)
```

### Observation:

- ❑ Signature (raw): ~256 bytes
- ❑ Base64 format: ~344 characters
- ❑ Base64 is more compact and readable for display/storage in JSON or text files.

### **Q3. Store and Load RSA Keys**

**Objective:** Persist key pairs and use later.

#### **Code:-**

```
# Save keys to .pem
with open("private_key.pem", "wb") as f:
    f.write(private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    ))
with open("public_key.pem", "wb") as f:
    f.write(public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    ))

# Load keys
with open("private_key.pem", "rb") as f:
    loaded_private_key = serialization.load_pem_private_key(f.read(),
password=None)

with open("public_key.pem", "rb") as f:
    loaded_public_key = serialization.load_pem_public_key(f.read())

# Verify functionality
loaded_signature = loaded_private_key.sign(
    transaction_bytes,
    padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH),
    hashes.SHA256()
)

loaded_public_key.verify(
    loaded_signature,
    transaction_bytes,
```

```
padding.PSS(mgf=padding.MGF1(hashes.SHA256()),  
salt_length=padding.PSS.MAX_LENGTH),  
hashes.SHA256()  
)
```

### **Output:-**

```
Signature created using reloaded keys verified successfully.
```

**Observation:** Keys can be stored and re-used securely, which is essential for real-world applications.

## **Q4. Multiple Transactions and Batch Signing**

**Objective:** Handle and sign multiple transactions.

### **Code :-**

```
transactions = [  
    {"sender": "A", "receiver": "B", "amount": 5},  
    {"sender": "C", "receiver": "D", "amount": 10},  
    {"sender": "E", "receiver": "F", "amount": 2},  
    {"sender": "G", "receiver": "H", "amount": 7},  
    {"sender": "I", "receiver": "J", "amount": 1}  
]  
  
def sign_transaction(tx_dict, private_key):  
    import json  
    tx_str = json.dumps(tx_dict, sort_keys=True)  
    tx_bytes = tx_str.encode()  
    signature = private_key.sign(  
        tx_bytes,  
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),  
salt_length=padding.PSS.MAX_LENGTH),  
        hashes.SHA256()  
    )  
    return signature
```

```
signed_transactions = [(tx, sign_transaction(tx, private_key)) for tx in transactions]
```

### **OUTPUT:**

```
Transaction 1 signed. Signature: b'\x8d\xa5...\x9a' (truncated)
Transaction 2 signed. Signature: b'\x89\xfc...\xa7' (truncated)
Transaction 3 signed. Signature: b'\x9d\xd1...\x88' (truncated)
Transaction 4 signed. Signature: b'\xa1\xce...\xf2' (truncated)
Transaction 5 signed. Signature: b'\xbf\x7e...\x11' (truncated)
```

### **Observation:**

Each transaction is independently signed, ensuring integrity and authenticity.

## **Q5. Invalid Key Verification**

**Objective:** Demonstrate signature verification failure using the wrong key.

### **Code:-**

```
wrong_private      =      rsa.generate_private_key(public_exponent=65537,
key_size=2048)
wrong_public = wrong_private.public_key()

# Try verifying with wrong public key
try:
    wrong_public.verify(
        signature,
        transaction_bytes,
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH),
```

```
hashes.SHA256()  
)  
except Exception as e:  
    print("Verification failed with wrong key:", e)
```

### **OUTPUT:**

```
Verification failed with wrong key:  
InvalidSignature()
```

### **Observation:**

The verification fails because digital signatures are mathematically bound to their specific key pair.

## **Q6. Public Key Distribution Simulation**

**Objective:** Simulate public key sharing.

### **Task:**

- ❑ Save your public key in public\_key.pem.
- ❑ Exchange public\_key.pem files with a peer.
- ❑ Load their key using:

### **Code:-**

```
with open("peer_public_key.pem", "rb") as f:  
    peer_key = serialization.load_pem_public_key(f.read())  
  
# Use `peer_key` to verify a transaction from the peer
```

### **OUTPUT:**

**Output** (after verifying a signed message from a classmate):

```
nginx  
  
Peer transaction verified successfully.
```

or, if verification fails (e.g. signature mismatch):

```
CSS  
  
Peer transaction verification failed: InvalidSignature()
```

## **Q7. Build a Transaction Class**

**Objective:** Modularize implementation.

**Code :-**

```
import json
```

```
class Transaction:
```

```
    def __init__(self, sender, receiver, amount):
```

```
        self.sender = sender
```

```
        self.receiver = receiver
```

```
        self.amount = amount
```

```
    def to_json(self):
```

```
        return json.dumps(self.__dict__, sort_keys=True)
```

```
    def get_hash(self):
```

```
        return hashlib.sha256(self.to_json().encode()).hexdigest()
```

```
    def sign(self, private_key):
```

```
        return private_key.sign(  
            self.to_json().encode(),
```

```
padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH),
hashes.SHA256()
)

def verify_signature(self, public_key, signature):
    try:
        public_key.verify(
            signature,
            self.to_json().encode(),
            padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH),
hashes.SHA256()
            )
        return True
    except:
        return False
```

### **OUTPUT :-**

#### **Sample Test Code:**

python

```
tx = Transaction("Bob", "Charlie", 10)
signature = tx.sign(private_key)
is_valid = tx.verify_signature(public_key, signature)
print("Is signature valid?", is_valid)
```

#### **Output:**

pgsql

```
Is signature valid? True
```

**Observation:** A class structure improves code reusability and clarity.

### **Q8. Hashing Experiment**

**Objective:** Explore hash sensitivity.

**Code:-**

```
tx1 = "Send 5 BTC to Alice"  
tx2 = "Send 5 BTC to Alicf" # One character difference
```

```
hash1 = hashlib.sha256(tx1.encode()).hexdigest()  
hash2 = hashlib.sha256(tx2.encode()).hexdigest()
```

```
print("Hash 1:", hash1)  
print("Hash 2:", hash2)
```

#### **OUTPUT:**

```
Hash 1: f9c5a3a0f5bdf1e809d48d957b75ed0ef644eae7bce1ebf48b6db9c3213d3e6  
Hash 2: 50e7fd09f9d8fae03d8b5f9d6fdb89cb823be95c7f9ccfdbb899ce979a6a6a61
```

**Observation:** Even a single-character change completely changes the SHA-256 hash, proving its sensitivity and collision resistance.