

Name: Mansi Doshi
Enrollment Number: 20230905090518
Batch: B.Tech I.T. (Genius)
Subject: Cryptography And Network Security
ALA-1

Digital Signature Verifier

Title: Digital Signature Verifier using Python

Objective:

- Understand the concept of digital signatures and public-key cryptography.
- Implement a system to sign a message using a private key and verify it using the corresponding public key.
- Ensure message integrity using SHA-256 hashing.

Outcome:

- Learn how digital signatures ensure authenticity, integrity, and non-repudiation of messages.
- Hands-on experience with Python's cryptography library.

Python Installation of Required Libraries:

pip install cryptography

Key Concepts

1. Public-Key Cryptography:
 - Two keys: Private Key (kept secret) and Public Key (shared).
 - Private key is used for signing; public key is used for verifying.
2. Digital Signature:
 - A hash of the message is created using SHA-256.
 - This hash is encrypted using the private key → forms the digital signature.

- Verification is done by decrypting the signature with the public key and comparing the hash.
3. SHA-256 Hashing:
- Produces a fixed-length 256-bit hash.
 - Any tiny change in the message changes the hash completely → ensures integrity.

Source Code:

Digital Signature Verifier in Python

```
from cryptography.hazmat.primitives import hashes
```

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

```
from cryptography.hazmat.primitives import serialization
```

```
# -----
```

Step 1: Generate RSA Key Pair

```
# -----
```

```
private_key = rsa.generate_private_key(
```

```
    public_exponent=65537,
```

```
    key_size=2048
```

```
)
```

```
public_key = private_key.public_key()
```

Save keys to files (optional)

```
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
))
# -----
# Step 2: Create Message
# -----
message = b"Hello, this is a secret message from Mansi!"
# -----
# Step 3: Hash the Message
# -----
digest = hashes.Hash(hashes.SHA256())
digest.update(message)
hashed_message = digest.finalize()
# -----
# Step 4: Sign the Message
# -----
signature = private_key.sign(
    hashed_message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)

```

```

print("Digital Signature:", signature.hex())

# -----

# Step 5: Verify the Signature

# -----

try:
    public_key.verify(
        signature,
        hashed_message,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    print("Verification Successful! Signature is valid.")
except Exception as e:
    print("Verification Failed!", e)

```

Explanation of Code:

- `rsa.generate_private_key()` → Generates RSA private key (2048 bits).
- `digest.update(message)` → Hash the message using SHA-256.
- `private_key.sign()` → Sign the hash → Digital Signature.
- `public_key.verify()` → Verify the signature.

Screenshots:

This screenshot shows the first part of a Python script named `digital_signature_system.py` in a VS Code editor. The Explorer panel on the left shows the project structure with files `digital_signature_system.py`, `private_key.pem`, and `public_key.pem`. The script content includes imports for `hashes`, `rsa`, and `padding` from `cryptography.hazmat.primitives`, and `serialization` from `cryptography.hazmat.primitives.serialization`. It then proceeds to generate an RSA key pair using `rsa.generate_private_key` with a public exponent of 65537 and a key size of 2048. The generated keys are saved to `private_key.pem` and `public_key.pem` using `private_key.private_bytes` and `public_key.public_bytes` respectively, with PEM encoding and PKCS8 format. The script ends with a message: `message = b"Hello, this is a secret message from Mansil"`.

```
1 # Digital Signature Verifier in Python
2 from cryptography.hazmat.primitives import hashes
3 from cryptography.hazmat.primitives.asymmetric import rsa, padding
4 from cryptography.hazmat.primitives.serialization import
5 # -----
6 # Step 1: Generate RSA Key Pair
7 # -----
8 private_key = rsa.generate_private_key(
9     public_exponent=65537,
10    key_size=2048
11 )
12 public_key = private_key.public_key()
13
14 # Save keys to files (optional)
15 with open("private_key.pem", "wb") as f:
16     f.write(private_key.private_bytes(
17         encoding=serialization.Encoding.PEM,
18         format=serialization.PrivateFormat.PKCS8,
19         encryption_algorithm=serialization.NoEncryption()
20     ))
21
22 with open("public_key.pem", "wb") as f:
23     f.write(public_key.public_bytes(
24         encoding=serialization.Encoding.PEM,
25         format=serialization.PublicFormat.SubjectPublicKeyInfo
26     ))
27 # -----
28 # Step 2: Create Message
29 # -----
30 message = b"Hello, this is a secret message from Mansil"
```

This screenshot shows the second part of the Python script `digital_signature_system.py`. It continues from the previous part, starting with the message `message = b"Hello, this is a secret message from Mansil"`. The script then hashes the message using `hashes.Hash(hashes.SHA256())` and updates the digest with the message. The hashed message is finalized using `digest.finalize()`. Next, the message is signed using `private_key.sign` with `padding.PSS` and `mgf=padding.MGF1(hashes.SHA256())`. The resulting signature is printed as a hexadecimal string using `signature.hex()`. Finally, the script attempts to verify the signature using `public_key.verify` with the same padding and MGF1 parameters.

```
30 message = b"Hello, this is a secret message from Mansil"
31 # -----
32 # Step 3: Hash the Message
33 # -----
34 digest = hashes.Hash(hashes.SHA256())
35 digest.update(message)
36 hashed_message = digest.finalize()
37 # -----
38 # Step 4: Sign the Message
39 # -----
40 signature = private_key.sign(
41     hashed_message,
42     padding.PSS(
43         mgf=padding.MGF1(hashes.SHA256()),
44         salt_length=padding.PSS.MAX_LENGTH
45     ),
46     hashes.SHA256()
47 )
48 print("Digital Signature:", signature.hex())
49 # -----
50 # Step 5: Verify the Signature
51 # -----
52 try:
53     public_key.verify(
54         signature,
55         hashed_message,
56         padding.PSS(
57             mgf=padding.MGF1(hashes.SHA256()),
58             salt_length=padding.PSS.MAX_LENGTH
59         )
60     )
```

