SCHOOL OF COMPUTER SCIENCE ENGINEERING AND INFORMATION SYSTEMS

FALL SEMESTER 2024 -2025

SWE3002 - INFORMATION AND SYSTEM SECURITY

COURSE FACULTY – PROF. PRABUKUMAR. M

SLOT – C2 + TC2

REVIEW - 2

TITLE:- SECURING SENSITIVE PATIENT DATA USING ENCRYPTION ALGORITHMS AND SHA-256

GROUP MEMBERS NAME – REGISTER NUMBER

AGASTHYA A – 22MIS0272

PAVAN P – 22MIS0337

RAGHUL N.S – 22MIS0508

1.) HASHING ALGORITHM:-

SHA-256 (SECURE HASH ALGORITHM 256-BIT)

CODE:-

```
import hashlib

import os

def hash_patient_data(patient_data):

    """

    Hashes patient data using SHA-256.

    Parameters:

    - patient_data (str): The data string containing sensitive patient information.

    Returns:

    - str: A hashed hexadecimal representation of the data.

    """

    # Generate a random salt for each record to increase security

    salt = os.urandom(16)  # 16 bytes of random salt

    # Combine salt with patient data

    salted_data = salt + patient_data.encode('utf-8')

    # Create SHA-256 hash of the salted data

    hash_object = hashlib.sha256(salted_data)

    # Convert to hexadecimal format

    hashed_data = hash_object.hexdigest()

    # Return the salt and hash (they'll need to be stored together for verification)

    return salt.hex(), hashed_data
```

EXPLANATION:

Salt Generation:

The code generates a unique random salt for each patient record to prevent attackers from using precomputed hash tables (rainbow tables) to reverse-engineer the data.

SHA-256 Hashing:

It uses SHA-256 to hash the patient data along with the salt, which ensures data integrity and is computationally difficult to reverse.

Hexadecimal Output:

The output hash is converted to hexadecimal format for storage, making it easier to store in databases or files.

NOTES:

Storing the Salt: The salt and hash need to be stored together because the salt will be required when verifying the data.

Data Security: Always use secure storage for the hashed values and salts, such as an encrypted database, to further protect patient data.

CONCLUSION:-

This hashing approach helps keep patient data secure by making it challenging to retrieve the original information from the hash alone.

CODE SCREEN SHOT:-

```
import hashlib
import os

def hash_patient_data(patient_data):
    """
    Hashes patient data using SHA-256.

    Parameters:
    - patient_data (str): The data string containing sensitive patient information.

    Returns:
    - str: A hashed hexadecimal representation of the data.
    """
    # Generate a random salt for each record to increase security
    salt = os.urandom(16)  # 16 bytes of random salt
    # Combine salt with patient data
    salted_data = salt + patient_data.encode('utf-8')
    # Create SHA-256 hash of the salted data
    hash_object = hashlib.sha256(salted_data)
    # Convert to hexadecimal format
    hashed_data = hash_object.hexdigest()
    # Return the salt and hash (they'll need to be stored together for verification)
    return salt.hex(), hashed_data
```

SAMPLE OUTPUT:-

```
# Example usage
patient_data = "John Doe, ID: 123456789, DOB: 01-01-1980"
salt, hashed_data = hash_patient_data(patient_data)
print(f"Salt: {salt}")
print(f"Hashed Data: {hashed_data}")
```

```
Salt: 5cb8c8b3179d449b878b390043a4fe63
Hashed Data: 6bb00e1113fcc51588634737e7cdd982a3fd6b6eb2750e0237c515abaa25b188
```

```
# Example usage1
patient_data = "John Doe, ID: 123456789, DOB: 01-01-1980"
salt, hashed_data = hash_patient_data(patient_data)
print(f"Salt: {salt}")
print(f"Hashed Data: {hashed_data}")

#Example usage2
patient_data = "Ram Kumar, ID: 123496789, DOB: 01-012-1985"
salt, hashed_data = hash_patient_data(patient_data)
print(f"Salt: {salt}")
print(f"Hashed Data: {hashed_data}")
```

```
Salt: 2698a5d2e47786b312d1fad36c024ae7
Hashed Data: 4f6dff5ca48c7aa8f045a97934a755123794a59b6dbadd0d94f04a6b202a9e4a
Salt: 7bcc73f6d366dc0bd3eaf252a75de6fe
Hashed Data: e5b87fca3e0432fe4fcf889a764cc5b758d0e740789831d48d3c00cba9724cb3
```

2.ENCRYPTION ALGORITHM:-

RSA – RIVEST-SHAMIR-ADLEMAN:

CODE:-

```
# Import necessary libraries
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.backends import default_backend
# Key Generation
def generate_keys():
  private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=default_backend()
  )
  public_key = private_key.public_key()
  # Serialize keys for storage or sharing
  pem_private_key = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.NoEncryption()
  )
  pem_public_key = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
  )
```

```
    return pem_private_key, pem_public_key
# Encrypt data with the public key
def encrypt_data(public_key_pem, data):
    public_key = serialization.load_pem_public_key(
        public_key_pem,
        backend=default_backend()
    )
    encrypted_data = public_key.encrypt(
        data.encode(),
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return encrypted_data
# Decrypt data with the private key
def decrypt_data(private_key_pem, encrypted_data):
    private_key = serialization.load_pem_private_key(
        private_key_pem,
        password=None,
        backend=default_backend()
    )
    decrypted_data = private_key.decrypt(
        encrypted_data,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
```

```
        label=None
    )
)
return decrypted_data.decode()
```

EXPLANATION:-

1. **Key Generation**:
    - generate_keys() creates a public and private key pair.
    - Both keys are serialized in PEM format, allowing for easy storage or sharing.
2. **Encryption**:
    - encrypt_data() takes the public key and the data to be encrypted.
    - Data is encoded to bytes and encrypted with OAEP padding (for security).
3. **Decryption**:
    - decrypt_data() takes the private key and the encrypted data.
    - Data is decrypted using the private key and returned in readable format.

NOTE:-

Security Level:-

- **Larger Key Size**: Increasing the key size (e.g., from 2048 bits to 4096 bits) provides stronger encryption. This makes it harder for attackers to break the encryption through brute-force attacks or cryptographic analysis.
- **Smaller Key Size**: Reducing the key size (e.g., from 2048 bits to 1024 bits) makes the encryption weaker and more vulnerable to attacks. Today, 1024-bit keys are considered insecure for most sensitive applications.

Recommendation:-

- 2048 bits is the minimum recommended key size for secure applications.
- 3072 bits or 4096 bits can be used for highly sensitive data or long-term security.
-

CODE SCREEN SHOT:-

jupyter Untitled7 Last Checkpoint: 9 minutes ago

File Edit View Run Kernel Settings Help

Code

[2]:

```python
# Import necessary libraries
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.backends import default_backend

# Key Generation
def generate_keys():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
    public_key = private_key.public_key()

    # Serialize keys for storage or sharing
    pem_private_key = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )

    pem_public_key = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    return pem_private_key, pem_public_key
```

```python
# Encrypt data with the public key
def encrypt_data(public_key_pem, data):
    public_key = serialization.load_pem_public_key(
        public_key_pem,
        backend=default_backend()
    )

    encrypted_data = public_key.encrypt(
        data.encode(),
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    return encrypted_data
```

```python
# Decrypt data with the private key
def decrypt_data(private_key_pem, encrypted_data):
    private_key = serialization.load_pem_private_key(
        private_key_pem,
        password=None,
        backend=default_backend()
    )

    decrypted_data = private_key.decrypt(
        encrypted_data,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    return decrypted_data.decode()
```

SAMPLE OUTPUT:-

1.)

```python
# Example Usage
if __name__ == "__main__":
    # Generate keys
    private_key, public_key = generate_keys()

    # Sample patient data
    patient_data = """
    Name: John Doe
    Age: 45
    Diagnosis: Hypertension
    """

    # Encrypt data
    encrypted = encrypt_data(public_key, patient_data)
    print("Encrypted data:", encrypted)

    # Decrypt data
    decrypted = decrypt_data(private_key, encrypted)
    print("\nDecrypted data:\n", decrypted)
```

Encrypted data: b'D\xfaU\xc9\xb5\x814|\xa5\xa9&|\x91\xc8\xac\xec\x92\x97^\xcf+\xc5\x8c,\x91\xbbgq\xd8\x94\x14\xab\xda\x94\x08\xf3X\xfa\x8a\xd7\xffl\\g\x0
5\x8c\x9c9j\xf2\t\xfe\x03w\xa6x\xd7\xa4\x1b=5\xb1\xad\xd1\x15\x89\x96\x88:\x1a\xea\x92}\xdbgL\x98d\xa9\xa2wm\x1d8\xa0N8\x11\x91\xa7\xde\x1f\x8e\xed3\x01
\x01\x92\x94\xb4\xa8\xa7\xfa\x8a7\x1a\xc8:\xc5\x81\xec*O\xb0\x02[\xdaH[[*\xfb\xf4\xac\xdf\xa1\xb5)\xf4\x89\xf5C \x9bQ\xdc\xfb\xd3XIqWs\xba\xb60\r\xa1r\x0
9&I\xec\xf3\x91\xdc\xb4\x8a\x10\xa0\xd8K\x1f\xad\x8e; K\xb9\xc2\xa0J\xc1w\x1ek\x96L\x13\xe2\x8c\x18"\xf0\xcd\xbf\xdf\xcc\xeaE\x88\r5N\xf38\xdc\xc0v\xf8
\xd3>\x9a\xda\xb9CzT\x10\x0c\xaf\x19\xed\xa2\xb2U\\\xaa[T\x858\xba\n\xa5\xc3V]\xb4\xa6\x12\x08\xaau\x83\x19^\x0b\xf8\x14\xa2c\xab\xa3\xa5\xc6\xf4d\x1d4\x
a=~\xdc\xa2'

Decrypted data:

    Name: John Doe
    Age: 45

2.)

```python
# Example Usage
if __name__ == "__main__":
    # Generate keys
    private_key, public_key = generate_keys()

    # Sample patient data
    patient_data = """
    Name: ram kumar
    Age: 35
    Diagnosis: heart patient
    """

    # Encrypt data
    encrypted = encrypt_data(public_key, patient_data)
    print("Encrypted data:", encrypted)

    # Decrypt data
    decrypted = decrypt_data(private_key, encrypted)
    print("\nDecrypted data:\n", decrypted)
```

Encrypted data: b"\xd71\x101\xea\xc1E\xaa\xae\x9f\x07-\xe9\xe1\xbc\xc0\xa5\x16f\x14\xb9\x94\xd1\xcd4\t\x9b\x15\x88\xa2\xc7/\xa3_\xf4p\x99\xa6\xce};\xe9=
\x1c$4+Y\xec\xf6c\t\xea\x1a\x9e\xd4\xe5fQ\xa8\xdf\xe6\x89\x1a\xa3\x08\x9e\xbb4\x9bTW\xf8"\xfeq\x19\xa3\xf0\xb16\x81\xab\xd6'L\xa6Y\xc4\n\xc0BM:\x18\xbba
\x8c_'\x87F\x89g\x0dr\x81\xd7\xed\x9a7w\xd9\x01\xee\x8c\x02\x99\x0c\xb99a:A\xe7Y\xf2\xa4\xa6\xbc];*`\xb5\xf8A\xb6\xd3)\t\xe7f \x8f\xaf\xb2\x9d\xa5\x11\xb
e\x81\xe4c]N\xc7\x96\xefA0\x04\xde\xa75\xc6\xab\xf0\x14\xe4\xbe\x9aw@(.\x11H\x8eY\xb3\xeda\x9b\x17\x8a4,\x94\x1f\xa9\xf8\xdb\xb4\x8fP\xbd\xff\xd0R'\xa1p1
\xe0\x11D\x87Z\x0f\xbe\x1e\x04\xc1\x15\xa3\x0b\x84\xca\x1d\xa1\x9b\xb2R2\t\xbc\xc9\xc2\xf1\xfeTs\xfc\xdd\xb9K\x95jI\xa8\xf4\x17\r\xa2\xe1\xde{\xe9{,\x1e
\xea\x12w\x10"

Decrypted data:

    Name: ram kumar
    Age: 35
    Diagnosis: heart patient