INDEX

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.No** | **Date** | **Practicals** | **Signature** |
| 1 |  | Write a program to store username and password in an encrypted form in a database to implement integrity lock. |  |
| 2 |  | Write a program to implement SSL. |  |
| 3 |  | Write a program to generate DSA SSH key. |  |
| 4 |  | Write a program to implement multilevel security. |  |
| 5 |  | Write a program to Demonstrates how to encrypt and decrypt the content of an XML node using 128-bit CBC AES encryption. |  |
|  |  |  |  |

**PRACTICAL NO: 1**

**Aim:** Write a program to store username and password in an encrypted form in a database to implement integrity lock.

import sqlite3

from cryptography.fernet import Fernet

def generate\_and\_store\_key():

    key = Fernet.generate\_key()

    with open('secret.key', 'wb') as key\_file:

        key\_file.write(key)

def load\_key():

    return open('secret.key', 'rb').read()

def encrypt\_data(data, key):

    f = Fernet(key)

    encrypted\_data = f.encrypt(data.encode())

    return encrypted\_data

def decrypt\_data(data, key):

    f = Fernet(key)

    decrypted\_data = f.decrypt(data).decode()

    return decrypted\_data

def create\_database():

    conn = sqlite3.connect('user\_credentials.db')

    c = conn.cursor()

    c.execute('''CREATE TABLE IF NOT EXISTS credentials

                 (username TEXT PRIMARY KEY, password TEXT)''')

    conn.commit()

    conn.close()

def store\_credentials(username, password):

    key = load\_key()

    encrypted\_username = encrypt\_data(username, key)

    encrypted\_password = encrypt\_data(password, key)

    conn = sqlite3.connect('user\_credentials.db')

    c = conn.cursor()

    c.execute('''INSERT OR REPLACE INTO credentials (username, password)

                 VALUES (?, ?)''', (encrypted\_username, encrypted\_password))

    conn.commit()

    conn.close()

def retrieve\_credentials():

    conn = sqlite3.connect('user\_credentials.db')

    c = conn.cursor()

    c.execute('''SELECT \* FROM credentials''')

    credentials = c.fetchone()

    conn.close()

    if credentials:

        key = load\_key()

        decrypted\_username = decrypt\_data(credentials[0], key)

        decrypted\_password = decrypt\_data(credentials[1], key)

        return decrypted\_username, decrypted\_password

    else:

        return None

create\_database()

generate\_and\_store\_key()

username = input("Enter username: ")

password = input("Enter password: ")

store\_credentials(username, password)

print("Credentials stored successfully.")

retrieved\_username, retrieved\_password = retrieve\_credentials()

if retrieved\_username and retrieved\_password:

    print("Retrieved username:", retrieved\_username)

    print("Retrieved password:", retrieved\_password)

else:

    print("No credentials found.")

**OUTPUT:-**

PS C:\Users\Aqsa Ulde\Downloads\NIS PRACTICAL PRACTICE\PRAC 1>

Enter username: axaa\_here

Enter password: abcdefg

Credentials stored successfully.

Retrieved username: axaa\_here

Retrieved password: abcdefg

PS C:\Users\Aqsa Ulde\Downloads\NIS PRACTICAL PRACTICE\PRAC 1>

**PRACTICAL NO: 2**

**Aim:** Write a program to implement SSL.

**INPUT:-**

from flask import Flask

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

# Routes

@app.route('/')

def index():

    return 'Hello, world!'

if \_\_name\_\_ == '\_\_main\_\_':

    # Run the Flask app with SSL enabled

    app.run(ssl\_context='adhoc')

**OUTPUT:-**

PS C:\Users\Aqsa Ulde\Downloads\NIS PRACTICAL PRACTICE\PRAC 4\Prac4.py'

\* Serving Flask app 'Prac 4'

\* Debug mode: off

WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.

\* Running on https://127.0.0.1:5000

**PRACTICAL NO: 3**

**Aim:** Write a program to generate DSA SSH key.

**INPUT:-**

import tempfile

import paramiko

def generate\_dsa\_key\_pair():

    key = paramiko.DSSKey.generate(bits=1024)

    with tempfile.NamedTemporaryFile(delete=False) as private\_key\_file:

        key.write\_private\_key\_file(private\_key\_file.name)

        private\_key\_file.close()

    with open(private\_key\_file.name, 'rb') as private\_key\_file:

        private\_key = private\_key\_file.read()

    public\_key = key.get\_base64()

    return private\_key, public\_key

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

    private\_key, public\_key = generate\_dsa\_key\_pair()

    print("Private Key:")

    print(private\_key.decode())

    print("\nPublic Key:")

    print(public\_key)

**OUTPUT:-**

Private Key:

-----BEGIN DSA PRIVATE KEY-----

MIIBuwIBAAKBgQDXm3/t15R78jpQytMHZ7N+YMZVclZl4Wh1oJQLZ0pQyTMzNxY4

3DBCCR99nBSJhNYADn3eyV+bRhlBCLJM0xgACK6dl9T336S9Buirl/IWtVF/PqaP

XuOJZEbekT+Drb+UTxB7gaal761nwH85XplzpJtOHbBkL2lNEnD9GDDwcQIVAOnq

3kpOnqpdq1Y8grXpzIKK/AItAoGAL6d3C810eyhWNn+5Fc1sJWvU4TnEHI5nXDBK

ikSZA02e4H1FlKa8NZfGQ2z7yq612qMq4rfSJxoVApqG5ahJfp86uNINwZLSEMMa **PRACTICAL NO: 4**

**Aim:** Write a program to implement multilevel security.

**INPUT:-**

class SecurityLevel:

    LOW = 0

    MEDIUM = 1

    HIGH = 2

class SecureData:

    def \_\_init\_\_(self, data, security\_level):

        self.data = data

        self.security\_level = security\_level

class SecurityManager:

    def \_\_init\_\_(self):

        self.data\_store = []

    def add\_data(self, data, security\_level):

        secure\_data = SecureData(data, security\_level)

        self.data\_store.append(secure\_data)

    def get\_data(self, user\_security\_level):

        accessible\_data = []

        for secure\_data in self.data\_store:

            if secure\_data.security\_level <= user\_security\_level:

                accessible\_data.append(secure\_data.data)

        return accessible\_data

manager = SecurityManager()

manager.add\_data("Public information", SecurityLevel.LOW)

manager.add\_data("Confidential information", SecurityLevel.MEDIUM)

manager.add\_data("Top secret information", SecurityLevel.HIGH)

print("User with LOW security clearance:", manager.get\_data(SecurityLevel.LOW))

print("User with MEDIUM security clearance:", manager.get\_data(SecurityLevel.MEDIUM))

print("User with HIGH security clearance:", manager.get\_data(SecurityLevel.HIGH))

**OUTPUT:-**

PS C:\Users\Aqsa Ulde\Downloads\NIS PRACTICAL PRACTICE\PRAC 8\prac8.py'

User with LOW security clearance: ['Public information']

User with MEDIUM security clearance: ['Public information', 'Confidential information']

User with HIGH security clearance: ['Public information', 'Confidential information', 'Top secret information']

PS C:\Users\Aqsa Ulde\Downloads\NIS PRACTICAL PRACTICE>

**PRACTICAL NO: 5**

**Aim:** Write a program to Demonstrates how to encrypt and decrypt the content of an XML node using 128-bit CBC AES encryption.

**INPUT:-**

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

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives import padding

from base64 import b64encode, b64decode

import xml.etree.ElementTree as ET

def encrypt(data, key):

    backend = default\_backend()

    iv = b'\x00' \* 16

    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)

    encryptor = cipher.encryptor()

    padder = padding.PKCS7(algorithms.AES.block\_size).padder()

    padded\_data = padder.update(data) + padder.finalize()

    ct = encryptor.update(padded\_data) + encryptor.finalize()

    return ct

def decrypt(ct, key):

    backend = default\_backend()

    iv = b'\x00' \* 16

    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)

    decryptor = cipher.decryptor()

    unpadder = padding.PKCS7(algorithms.AES.block\_size).unpadder()

    padded\_data = decryptor.update(ct) + decryptor.finalize()

    data = unpadder.update(padded\_data) + unpadder.finalize()

    return data

# Example XML data

xml\_data = """

<root>

    <sensitive\_data>Hello, this is sensitive!</sensitive\_data>

</root>

"""

key = b'SuperSecretKey1234'[:16]

# Parse XML

root = ET.fromstring(xml\_data)

sensitive\_node = root.find('sensitive\_data')

# Encrypt the content

encrypted\_data = encrypt(sensitive\_node.text.encode(), key)

# Replace the content

sensitive\_node.text = b64encode(encrypted\_data).decode()

# Print the modified XML

print(ET.tostring(root, encoding='unicode'))

# Decrypt the content

decrypted\_data = decrypt(b64decode(sensitive\_node.text.encode()), key)

# Print the decrypted data

print("Decrypted data:", decrypted\_data.decode())

**OUTPUT:-**

PS C:\Users\Aqsa Ulde\Downloads\NIS PRACTICAL PRACTICE\PRAC 9\prac9.py'

<root>

<sensitive\_data>waYpdPUaDBI9TRmn8AzE94dM1xFNnqqaISw1jNHS25Q=

</sensitive\_data>

</root>

Decrypted data: Hello, this is sensitive!

PS C:\Users\Aqsa Ulde\Downloads\NIS PRACTICAL PRACTICE>