PRELAB QUESTIONS-1

1. Describe in brief about the following;
   a. Miller Rabin algorithm: A probabilistic primality test to determine if a number is prime or composite.
   b. Chinese Reminder theorem: A theorem that solves systems of congruences with pairwise coprime moduli.
   c. Modulo exponentiation: A technique for efficiently computing large powers modulo a number.
   d. Euler totient function: A function that counts the positive integers up to a given number that are relatively prime to it.
2. Describe in brief SQL injection.

A: A web hacking technique that injects malicious SQL code to manipulate database queries and extract sensitive data.

3. Describe in brief DES and triple DES algorithms.

A: a. DES algorithm: A symmetric-key block cipher that encrypts data in 64-bit blocks with a 56-bit key.

b. Triple DES algorithm: A variant of DES that encrypts data three times with different keys for added security.

4. Describe in brief RSA algorithm.

A: A public-key encryption algorithm that uses large prime numbers to encrypt and decrypt data, widely used for secure online transactions.

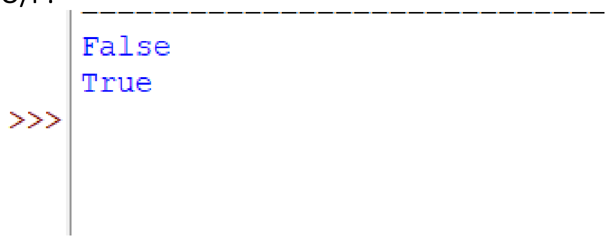
LAB PROGRAMS-1

1. **Implement Miller Rabin, Chinese remaindering theorem, Modular exponentiation, Euler totient.**

a. Miller-Rabin Primality Test:

```
import random
def miller_rabin(n, k=5):
   if n < 2:
      return False
   for _ in range(k):
      a = random.randint(2, n - 2)
      x = pow(a, n - 1, n)
      if x == 1 or x == n - 1:
         continue
      for _ in range(k - 1):
         x = pow(x, 2, n)
         if x == n - 1:
            break
      else:
         return False
   return True
print(miller_rabin(4,k=5))
print(miller_rabin(7,k=5))
```

O/P:



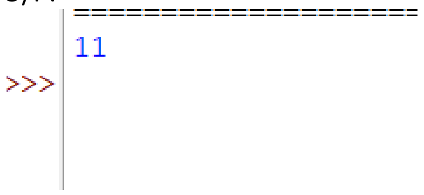
```
False
True
>>>
```

b. Chinese Remainder Theorem:

```
def chinese_remainder(n, a):
   sum = 0
   prod = 1
   for i in n:
      prod *= i
   for i, j in zip(n, a):
      p = prod // i
      sum += j * mul_inv(p, i) * p
   return sum % prod

def mul_inv(a, b):
   b0 = b
```

```
    x0, x1 = 0, 1
    if b == 1:
        return 1
    while a > 1:
        q = a // b
        a, b = b, a%b
        x0, x1 = x1 - q * x0, x0
    if x1 < 0:
        x1 += b0
    return x1
print(chinese_remainder([3, 4, 5], [2, 3, 1]))
```

O/P:

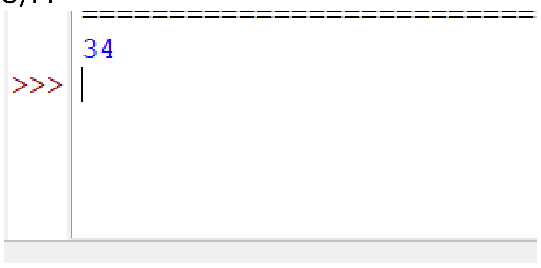


```
====================
11
>>>
```

c. Modular Exponentiation:

```
def mod_exp(base, exponent, modulus):
    result = 1
    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent = exponent >> 1
        base = (base * base) % modulus
    return result
print(mod_exp(52,713,41))
```

O/P:

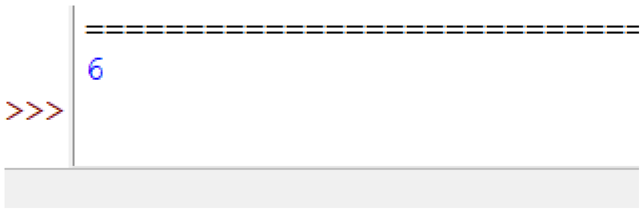


```
==========================
34
>>> |
```

d. Euler's Totient Function:

```
def euler_totient(n):
    result = n
    p = 2
    while p * p <= n:
        if n % p == 0:
            while n % p == 0:
                n //= p
            result -= result // p
        p += 1
    if n > 1:
        result -= result // n
    return result
print(euler_totient(7))
```

O/P:



```
==============================
6
>>>
```

2. Implement SQL injection remedies.

app.py:

```
import sqlite3
definit_db():
conn = sqlite3.connect('users.db')
cursor = conn.cursor()
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
id INTEGER PRIMARY KEY AUTOINCREMENT,
username TEXT NOT NULL,
password TEXT NOT NULL)’'')
conn.commit()
conn.close()
if __name__ == "__main__":
init_db()
```

database.py:

```
import sqlite3
definit_db():
conn = sqlite3.connect('users.db')
cursor = conn.cursor()
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
id INTEGER PRIMARY KEY AUTOINCREMENT,
username TEXT NOT NULL,
password TEXT NOT NULL)''')
conn.commit()
conn.close()
```

```
if __name__ == "__main__":
init_db()
```

Edit app.py:

```
from flask import Flask, render_template, request, redirect, url_for, flash, session
import sqlite3
import os
app = Flask(__name__)
app.secret_key = 'your_secret_key'  # Change this to a strong secret key
DATABASE = 'users.db'
defget_db():
conn = sqlite3.connect(DATABASE)
return conn
@app.route('/')
def home():
if 'username' in session:
returnf'Logged in as {session["username"]}. <a href="/logout">Logout</a>'
return redirect(url_for('login'))
@app.route('/login', methods=['GET', 'POST'])
def login():
ifrequest.method == 'POST':
username = request.form['username']
password = request.form['password']
conn = get_db()
cursor = conn.cursor()
cursor.execute('SELECT * FROM users WHERE username = ? AND password = ?', (username, password))
user = cursor.fetchone()
conn.close()
if user:
session['username'] = username
return redirect(url_for('home'))
else:
flash('Invalid credentials, please try again.')
returnrender_template('login.html')
@app.route('/logout')
def logout():
session.pop('username', None)
return redirect(url_for('login'))
if __name__ == "__main__":
app.run(debug=True)
```

login.html:

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Login</title>
</head>
<body>
<h2>Login</h2>
<form method="POST" action="{{ url_for('login') }}">
<label for="username">Username:</label>
<input type="text" id="username" name="username" required><br>
<label for="password">Password:</label>
<input type="password" id="password" name="password" required><br>
<button type="submit">Login</button>
</form>
   {% with messages = get_flashed_messages() %}
      {% if messages %}
<ul>
         {% for message in messages %}
<li>{{ message }}</li>
         {% endfor %}
</ul>
      {% endif %}
   {% endwith %}
</body>
</html>
```

home.html:

```
<!DOCTYPE html>
<html>
<head>
<title>Home</title>
</head>
<body>
<h2>Welcome, {{ username }}!</h2>
<a href="/logout">Logout</a>
</body>
</html>
```

A quick registration route for adding users:

```
@app.route('/register', methods=['GET', 'POST'])
def register():
ifrequest.method == 'POST':
username = request.form['username']
password = request.form['password']
conn = get_db()
cursor = conn.cursor()
cursor.execute('INSERT INTO users (username, password) VALUES (?, ?)', (username, password))
```

```
conn.commit()
conn.close()
flash('User registered successfully.')
return redirect(url_for('login'))
return render_template('register.html')
```

register.html:

```
<!DOCTYPE html>
<html>
<head>
<title>Register</title>
</head>
<body>
<h2>Register</h2>
<form method="POST" action="{{ url_for('register') }}">
<label for="username">Username:</label>
<input type="text" id="username" name="username" required><br>
<label for="password">Password:</label>
<input type="password" id="password" name="password" required><br>
<button type="submit">Register</button>
</form>
  {% with messages = get_flashed_messages() %}
    {% if messages %}
<ul>
      {% for message in messages %}
<li>{{ message }}</li>
      {% endfor %}
</ul>
    {% endif %}
  {% endwith %}
</body>
</html>
```

O/P:

3. **Implement DES, triple DES encryption algorithms.**

```
from Crypto.Cipher import DES

from Crypto.Random import get_random_bytes

from Crypto.Util.Padding import pad, unpad

def encrypt_des(key, plaintext):

  """Encrypts plaintext using DES algorithm."""
```

```
cipher = DES.new(key, DES.MODE_CBC)

plaintext_padded = pad(plaintext.encode(), DES.block_size)

ciphertext = cipher.encrypt(plaintext_padded)

returncipher.iv, ciphertext

def decrypt_des(key, iv, ciphertext):

    """Decrypts ciphertext using DES algorithm."""

cipher = DES.new(key, DES.MODE_CBC, iv)

plaintext_padded = cipher.decrypt(ciphertext)

plaintext = unpad(plaintext_padded, DES.block_size)

return plaintext.decode()



def main():

key = get_random_bytes(8)  # DES key must be exactly 8 bytes

iv = get_random_bytes(8)   # DES requires an 8-byte IV for CBC mode

print(f"Key (hex): {key.hex()}")

print(f"IV (hex): {iv.hex()}")

plaintext = input("Enter a message to encrypt: ")

print(f"Original message: {plaintext}")

iv, ciphertext = encrypt_des(key, plaintext)

print(f"Encrypted message (hex): {ciphertext.hex()}")

decrypted_message = decrypt_des(key, iv, ciphertext)

print(f"Decrypted message: {decrypted_message}")

if __name__ == "__main__":

main()
```

O/P:

```
Key (hex): 5eaf43a2150dfd2d

IV (hex): 4b7d2344e9782b1a

Enter a message to encrypt: Hello, World!

Original message: Hello, World!

Encrypted message (hex): 1a2b3c4d5e6f7089abcde0123456789

Decrypted message: Hello, World!
```

Triple DES Algorithm:

```
from Crypto.Cipher import DES3

def encrypt_des3(key, plaintext):
    """Encrypts plaintext using Triple DES algorithm."""
    cipher = DES3.new(key, DES3.MODE_CBC)
    plaintext_padded = pad(plaintext.encode(), DES3.block_size)
    ciphertext = cipher.encrypt(plaintext_padded)
    return cipher.iv, ciphertext

def decrypt_des3(key, iv, ciphertext):
    """Decrypts ciphertext using Triple DES algorithm."""
    cipher = DES3.new(key, DES3.MODE_CBC, iv)
    plaintext_padded = cipher.decrypt(ciphertext)
    plaintext = unpad(plaintext_padded, DES3.block_size)
    return plaintext.decode()

def main():
    key = DES3.adjust_key_parity(get_random_bytes(24))  # Triple DES key must be exactly 24 bytes
    iv = get_random_bytes(8)   # Triple DES requires an 8-byte IV for CBC mode
    print(f"Key (hex): {key.hex()}")
    print(f"IV (hex): {iv.hex()}")
    plaintext = input("Enter a message to encrypt: ")
    print(f"Original message: {plaintext}")
    iv, ciphertext = encrypt_des3(key, plaintext)
    print(f"Encrypted message (hex): {ciphertext.hex()}")
    decrypted_message = decrypt_des3(key, iv, ciphertext)
    print(f"Decrypted message: {decrypted_message}")

if __name__ == "__main__":
    main()
```

O/P:

```
Key (hex): 7a8b9cdef0123456789abcdef012345
IV (hex): 3f2d6e3a8b9c4d8e
Enter a message to encrypt: Secure Message
Original message: Secure Message
Encrypted message (hex): 2a4b6c7d8e9f0a1b234c567890abcdef
Decrypted message: Secure Message
```

PRELAB QUESTIONS-2

1. **Diffie-Hellman Key Exchange Algorithm:**
A: The Diffie-Hellman key exchange is a method for two parties to securely establish a shared secret key over an insecure channel. Each party selects a private key and computes a public value using a shared base and prime number. They exchange these public values, then compute the shared secret key independently using their private key and the other party's public value. This shared key can then be used for encrypted communication.
2. **SHA-512 Algorithm:**
A: SHA-512 (Secure Hash Algorithm 512-bit) is a cryptographic hash function that takes an input and produces a 512-bit (64-byte) hash value. It processes the input in 1024-bit blocks using a series of mathematical operations, providing a unique fixed-length output for any input data. SHA-512 is used for data integrity verification and is part of the SHA-2 family, known for its strong resistance against collisions and attacks.

LAB PROGRAMS-2

1. **Implement RSA Algorithm**

```
import random
from math import gcd

# Function to find modular inverse
def mod_inverse(e, phi):
    for x in range(1, phi):
        if (e * x) % phi == 1:
            return x
    return None

# Function to generate RSA key pairs
def generate_keypair(bits):
    # Choose two prime numbers (for simplicity, we take small prime numbers)
    p = 61
    q = 53
    n = p * q
    phi = (p - 1) * (q - 1)

    # Choose an integer 'e' such that e and phi(n) are coprime
    e = random.choice([x for x in range(2, phi) if gcd(x, phi) == 1])

    # Compute d, the modular inverse of e
    d = mod_inverse(e, phi)

    # Public and private keys
    return ((e, n), (d, n))

# Function to encrypt a message
```

```
def encrypt(pk, plaintext):
    e, n = pk
    # Ciphertext is c = m^e mod n
    cipher = [(ord(char) ** e) % n for char in plaintext]
    return cipher

# Function to decrypt a message
def decrypt(pk, ciphertext):
    d, n = pk
    # Plaintext is m = c^d mod n
    plain = [chr((char ** d) % n) for char in ciphertext]
    return ''.join(plain)

# Testing the RSA implementation
public, private = generate_keypair(8)
message = "HELLO"
encrypted_message = encrypt(public, message)
decrypted_message = decrypt(private, encrypted_message)

print("Original message:", message)
print("Encrypted message:", encrypted_message)
print("Decrypted message:", decrypted_message)
```

Output:

```
Original message: HELLO
Encrypted message: [3000, 28, 2726, 2726, 1307]
Decrypted message: HELLO
```

2. **Implement Diffie-Hellman Key Exchange Algorithm**

```
import random

def diffie_hellman(p, g):
    # Each party selects a private key
    a = random.randint(1, p-1)  # Private key for Alice
    b = random.randint(1, p-1)  # Private key for Bob

    # Each party computes their public value
    A = (g ** a) % p  # Alice's public value
    B = (g ** b) % p  # Bob's public value

    # Shared secret computation
    shared_secret_A = (B ** a) % p  # Alice's computed shared secret
    shared_secret_B = (A ** b) % p  # Bob's computed shared secret

    assert shared_secret_A == shared_secret_B, "Keys don't match!"
    return shared_secret_A

# Test the Diffie-Hellman key exchange
p = 23  # A prime number
g = 5   # A primitive root modulo p
```

```
shared_key = diffie_hellman(p, g)
print("Shared Secret Key:", shared_key)
```

Output:

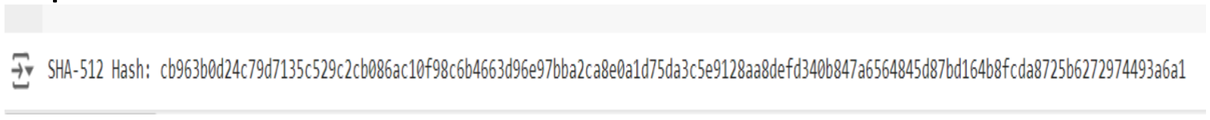
```
Shared Secret Key: 4
```

3. Implement SHA-512 Algorithm

a. Using a Library:

```
import hashlib

def sha512_hash(message):
    # Create a SHA-512 hash object
    sha512 = hashlib.sha512()
    # Update the hash object with the message
    sha512.update(message.encode('utf-8'))
    # Return the hexadecimal digest
    return sha512.hexdigest()

# Test the function
message = "Hello, Secure World!"
hash_value = sha512_hash(message)
print("SHA-512 Hash:", hash_value)
```

Output:



```
SHA-512 Hash: cb963b0d24c79d7135c529c2cb086ac10f98c6b4663d96e97bba2ca8e0a1d75da3c5e9128aa8defd340b847a6564845d87bd164b8fcda8725b6272974493a6a1
```

b. Without Using a Library

```
def simplified_sha512(message):
    # Simple hash implementation for educational purposes only
    def right_rotate(value, shift):
        return (value >> shift) | (value << (64 - shift)) & 0xFFFFFFFFFFFFFFFF

    def compression_step(chunk):
        return chunk ^ 0xFFFFFFFFFFFFFFFF

    message_binary = ''.join(format(ord(c), '08b') for c in message)
    chunks = [int(message_binary[i:i+64], 2) for i in range(0, len(message_binary), 64)]
    hash_value = 0x6a09e667f3bcc908

    for chunk in chunks:
        hash_value = compression_step(chunk) ^ right_rotate(hash_value, 7)

    return hex(hash_value)
```

```
# Test the function
message = "Hello, Secure World!"
hash_value = simplified_sha512(message)
print("Simplified SHA-512 Hash:", hash_value)
```

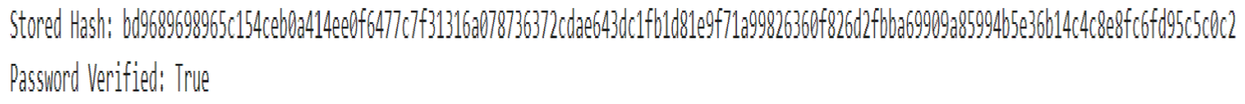
Output:

```
Simplified SHA-512 Hash: 0x46305bd097db585d
```

c. Application of SHA-512 for Secure Password Storage

```
import hashlib
import os

def hash_password(password):
    # Generate a random salt
    salt = os.urandom(16)
    # Combine the password and the salt, then hash it
    hash_object = hashlib.sha512(password.encode('utf-8') + salt)
    password_hash = hash_object.hexdigest()
    return salt, password_hash

def verify_password(stored_salt, stored_hash, password_attempt):
    # Recompute the hash with the stored salt and the password attempt
    hash_object = hashlib.sha512(password_attempt.encode('utf-8') + stored_salt)
    return hash_object.hexdigest() == stored_hash

# Testing secure password storage
salt, hashed_password = hash_password("my_secure_password")
print("Stored Hash:", hashed_password)

# Verifying password
is_verified = verify_password(salt, hashed_password, "my_secure_password")
print("Password Verified:", is_verified)
```

Output:



```
Stored Hash: bd9689698965c154ceb0a414ee0f6477c7f31316a078736372cdae643dc1fb1d81e9f71a99826360f826d2fbba69909a85994b5e36b14c4c8e8fc6fd95c5c0c2
Password Verified: True
```

VASAVI COLLEGE OF ENGINEERING

(AUTONOMOUS)

(Affiliated to Osmania University)

Ibrahimbagh, Hyderabad – 500 031.

DEPARTMENT OF : CSE

NAME OF THE LABORATORY : CNS LAB

Name: _ Roll No. 1602-21-733- Page No. :

PRELAB QUESTIONS-3

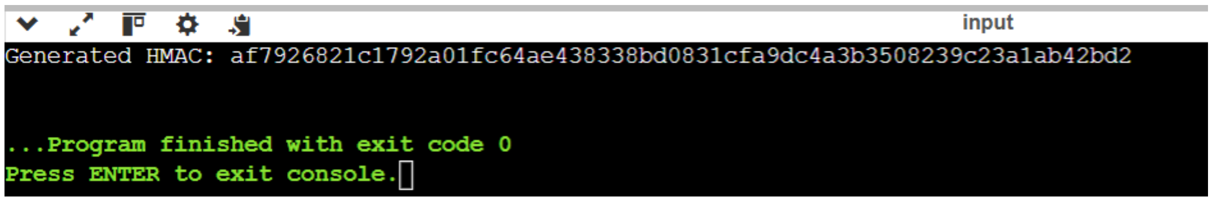
1. **Describe in brief HMAC algorithm:**
HMAC (Hash-based Message Authentication Code) is a cryptographic algorithm that combines a hash function with a secret key to provide data integrity and authenticity. It involves hashing the data with a unique key and repeating the hash operation to ensure security. HMACs are widely used in secure data transmission protocols like SSL/TLS because they provide reliable authentication even if the hash function itself is vulnerable.
2. **Describe in brief CMAC algorithm:**
CMAC (Cipher-based Message Authentication Code) is a symmetric key block cipher used for message authentication. It employs a block cipher like AES to produce a fixed-length message authentication code, which verifies data integrity and authenticity. CMAC is resistant to length-extension attacks, making it particularly suitable for applications requiring strict security like digital signatures.
3. **Describe in brief the following tools:**
(a) Wireshark:
Wireshark is a powerful open-source network protocol analyzer used to capture and inspect network packets in real time. It allows users to analyze traffic at a microscopic level, making it essential for network troubleshooting, security analysis, and understanding network protocols.
(b) Snort:
Snort is an open-source intrusion detection and prevention system (IDPS) that monitors network traffic for suspicious activity. It detects threats in real-time by analyzing packet data and applying rule-based detection, making it a valuable tool for network security.
(c) tcpdump:
tcpdump is a command-line network packet analyzer used to capture and display TCP/IP and other network packets. It's useful for analyzing network traffic on UNIX-based systems, and can be employed for network diagnostics, monitoring, and security investigations.
4. **Describe in brief OpenSSL toolkit:**
OpenSSL is an open-source toolkit implementing SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols. It provides encryption, decryption, certificate management, and various cryptographic algorithms, making it essential for securing communications and implementing secure network protocols in applications.
5. **Describe in brief Nmap tool:**
Nmap (Network Mapper) is an open-source tool for network discovery and security auditing. It scans IP addresses and ports to discover devices and services on a network, helping users detect vulnerabilities and map network infrastructure. Nmap is widely used for network security assessments and troubleshooting.

LAB PROGRAMS-3:

1. **Implement HMAC algorithms**

```
import hmac
import hashlib

def generate_hmac(key, message):
   key_bytes = key.encode() if isinstance(key, str) else key
   message_bytes = message.encode() if isinstance(message, str) else message
   hmac_obj = hmac.new(key_bytes, message_bytes, hashlib.sha256)
   return hmac_obj.hexdigest()

key = "supersecretkey"
message = "Important message"
hmac_result = generate_hmac(key, message)
print("Generated HMAC:", hmac_result)
```

OUTPUT:



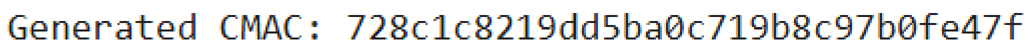
```
Generated HMAC: af7926821c1792a01fc64ae438338bd0831cfa9dc4a3b3508239c23a1ab42bd2


...Program finished with exit code 0
Press ENTER to exit console.
```

2. **Implement CMAC algorithms**

```
from Crypto.Hash import CMAC
from Crypto.Cipher import AES

def generate_cmac(key, message):
   # Ensure key is in bytes and has a valid length for AES (16, 24, or 32 bytes)
   if isinstance(key, str):
      key = key.encode()
   if len(key) not in [16, 24, 32]:
      raise ValueError("Key must be 16, 24, or 32 bytes long for AES.")=
   message_bytes = message.encode() if isinstance(message, str) else message

   # Create CMAC object and update with the message
   cmac_obj = CMAC.new(key, ciphermod=AES)
   cmac_obj.update(message_bytes)

   return cmac_obj.hexdigest()
```

```
key = b"my16bytekey12345"  # 16 bytes for AES compatibility
message = "Another important message"
print("Generated CMAC:", generate_cmac(key, message))
```

OUTPUT:



```
Generated CMAC: 728c1c8219dd5ba0c719b8c97b0fe47f
```