1. Port Scanning Using Python: Analyzing Open and Closed Ports on a Remote IP

Aim :

To develop a Python script that scans a range of ports on a given IP address and determines whether each port is open or closed.

Requirements :

Google Colab: Used for writing and executing Python code.

Python 3.x: Default Python version in Colab.

Libraries: Built-in Python libraries (e.g., `socket`).

Algorithm :

1. Create a function port_scanner that:
   - Takes an IP address and a port number as arguments.
   - Attempts a TCP connection to the specified port using the socket library.
   - Prints whether the port is open or closed based on the connection result.
2. Loop through a range of ports (e.g., from 45 to 55).
3. Call the port_scanner function for each port in the range.

Procedure :

1. **Prepare the Environment**:
   - Set up a Python environment (local machine, Google Cloud VM, etc.).
   - If using Google Cloud, configure firewall rules to allow the necessary ports for testing.
2. **Write the Code**:

    - Develop the Python code in Google Colab or any IDE.
3. **Run the Code**:
    - Execute the script either locally, on a VM, or in Google Colab (for logging purposes).
4. **Analyze Results**:
    - Review the printed output to determine which ports are open or closed.

Code :

```
import socket
def port_scanner(ip, port):
   try:
      with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
         s.settimeout(1)  # Timeout for connection
         s.connect((ip, port))
         print(f"[+] Port {port} is OPEN on {ip}")
   except:
      print(f"[-] Port {port} is CLOSED on {ip}")
# Example Usage
ip = "8.8.8.8"  # Localhost
for port in range(45,55):
   port_scanner(ip, port)
```

Output :

```
[-] Port 45 is CLOSED on 8.8.8.8
[-] Port 46 is CLOSED on 8.8.8.8
[-] Port 47 is CLOSED on 8.8.8.8
[-] Port 48 is CLOSED on 8.8.8.8
[-] Port 49 is CLOSED on 8.8.8.8
[-] Port 50 is CLOSED on 8.8.8.8
[-] Port 51 is CLOSED on 8.8.8.8
[-] Port 52 is CLOSED on 8.8.8.8
[+] Port 53 is OPEN on 8.8.8.8
[-] Port 54 is CLOSED on 8.8.8.8
```

Result :

The program checks each port and reports whether it is open or closed. It helps identify which ports are accessible on the target IP.

2. Log Analysis: Capturing User Input with Timestamps

Aim :

To capture user input continuously, log it with a timestamp, and save it in a text file.

Requirements :

Google Colab: Used for writing and executing Python code.

Python 3.x: Default Python version in Colab.

Libraries: Built-in Python libraries (e.g., `datetime`).

Algorithm :

1. Prompt the user for input repeatedly.
2. If the input is "exit", stop the loop and exit.
3. Otherwise, get the current timestamp.
4. Log the input and timestamp to a file (keylog.txt).
5. Repeat the process until "exit" is entered.

Procedure :

1. Write the Python script that continuously captures user input.
2. Use the datetime module to get the current timestamp.
3. Open the file (keylog.txt) in append mode to log each input.
4. Capture input until the user types 'exit'.
5. After execution, the file will contain all the logged inputs with timestamps.

Code :

```
from datetime import datetime

def capture_input():
    # Prompt user for input and allow multiple entries
    while True:
        user_input = input("Please type something (type 'exit' to stop): ")

        if user_input.lower() == 'exit':
            print("Exiting input capture.")
            break

        # Open the file in append mode and log the input with a timestamp
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        with open("keylog.txt", "a") as f:
            f.write(f"[{timestamp}] {user_input}\n")

        print(f"Captured input: {user_input}")

# Call the function to start capturing input
capture_input()
```

Output :

```
Please type something (type 'exit' to stop): hello
Captured input: hello
Please type something (type 'exit' to stop): world
Captured input: world
Please type something (type 'exit' to stop): exit
Exiting input capture.
```

Result :

The program logs user input along with a timestamp in a file. It continues until the user types "exit," saving all inputs to a text file.

3. Web Application Security: Testing for SQL Injection Vulnerabilities

Aim :

To test web applications for potential SQL injection vulnerabilities by injecting a payload into URL parameters.

Requirements :

Google Colab: Used for writing and executing Python code.

Python 3.x: Default Python version in Colab.

Libraries: Built-in Python libraries (e.g., `requests`).

Algorithm :

1. Define the function check_sql_injection that:
    - Takes a URL and a parameter name as inputs.
    - Injects a SQL payload (' OR 1=1 --) into the URL.
2. Send a GET request with the payload.
3. Check the response for any error or syntax-related messages.
4. Print whether the site is vulnerable to SQL injection or not.

Procedure :

1. Import the requests module to make HTTP requests.
2. Define the check_sql_injection function that takes a URL and parameter as input.
3. For each URL, inject a SQL payload and check the response.

4. If the response contains error messages or SQL syntax issues, mark it as vulnerable.
5. Call the function with various URLs and parameters to check for SQL injection.

Code :

```
import requests


def check_sql_injection(url, param):
    payload = f"{url}?{param}=' OR 1=1 --"
    response = requests.get(payload)
    if "error" in response.text or "syntax" in response.text:
        print(f"Potential SQL injection vulnerability at: {payload}")
    else:
        print(f"Seems safe: {url}")


# Example Usage
check_sql_injection("https://portal.naanmudhalvan.tn.gov.in/login", "login")
check_sql_injection("http://example.com/search", "query")
check_sql_injection("http://testphp.vulnweb.com/login.php", "username")
check_sql_injection("http://testphp.vulnweb.com/search.php", "search")
```

Output :

```
      Seems safe: https://portal.naanmudhalvan.tn.gov.in/login
      Seems safe: http://example.com/search
      Potential SQL injection vulnerability at:
http://testphp.vulnweb.com/login.php?username=' OR 1=1 –
      Potential SQL injection vulnerability at:
http://testphp.vulnweb.com/search.php?search=' OR 1=1 --
```

Result :

The program checks for SQL injection vulnerabilities in web applications by injecting a test payload. It reports potential vulnerabilities based on the server response.

4. 2-Factor Authentication: Implementing Time-based One-Time Password (OTP) Generation

Aim :

To implement a simple 2-factor authentication system that generates a one-time password (OTP) based on the current time and a secret key.

Requirements :

Google Colab: Used for writing and executing Python code.

Python 3.x: Default Python version in Colab.

Algorithm :

1. Define a secret key (e.g., 123456).
2. Generate a 6-digit OTP using the current timestamp and the secret key.
3. Prompt the user to enter their password.
4. If the password is correct, generate and display the OTP.
5. Prompt the user to enter the OTP.
6. If the OTP entered by the user matches the generated OTP, authenticate the user.

Procedure :

1. Define the secret key and the OTP generation function.
2. Ask the user to input their password.
3. If the password matches, generate the OTP.
4. Display the OTP to the user.
5. Ask the user to input the OTP, then verify if it matches.
6. Print an authentication message based on whether the OTP matches.

Code :

```
# 2 FACTOR AUTHENTICATION
import time

# Define the secret key
secret_key = 123456

def generate_otp():
  return str((int(time.time()) + secret_key) % 1000000).zfill(6)

# User login and OTP check
password = input("Enter your password: ")
if password == "12345":
  otp = generate_otp()
  print(f"Your OTP is: {otp}")
  user_otp = input("Enter the OTP: ")
  if user_otp == otp:
    print("Authentication complete.")
  else:
    print("Invalid OTP.")
else:
  print("Incorrect password.")
```

Output :

```
Enter your password: 12345
Your OTP is: 643284
Enter the OTP: 643284
Authentication complete.
```

Result :

The program prompts the user for a password and generates an OTP for two-factor authentication. The user is authenticated only if the correct OTP is entered.

5. Simple Caesar Cipher: Implementing Encryption and Decryption with Shift Values

Aim :

To implement a simple Caesar cipher for encryption and decryption of messages based on a shift value.

Requirements :

Google Colab: Used for writing and executing Python code.

Python 3.x: Default Python version in Colab.

Algorithm :

1. Take a message and a shift value as input.
2. For each character in the message:
   - If it's a letter, apply the Caesar cipher shift (either encryption or decryption).
   - If it's not a letter, leave it unchanged.
3. Return the encrypted or decrypted message.

Procedure :

1. Define the caesar_cipher function that handles encryption and decryption.
2. Prompt the user to input the message and shift value.
3. Encrypt the message by calling the function with encrypt mode.
4. Decrypt the message by calling the function with decrypt mode.
5. Print both the encrypted and decrypted messages.

Code :

```
def caesar_cipher(message, shift, mode='encrypt'):
    result = ""
    for char in message:
        if char.isalpha():
            shift_base = 65 if char.isupper() else 97
            shift_value = shift if mode == 'encrypt' else -shift
            result += chr((ord(char) - shift_base + shift_value) % 26 + shift_base)
        else:
            result += char
    return result

# Example usage
message = input("Enter the message: ")
shift = int(input("Enter the shift value: "))

encrypted_message = caesar_cipher(message, shift, 'encrypt')
print(f"Encrypted message: {encrypted_message}")

decrypted_message = caesar_cipher(encrypted_message, shift, 'decrypt')
print(f"Decrypted message: {decrypted_message}")
```

Output :

```
Enter the message: hello world
Enter the shift value: 5
Encrypted message: mjqqt btwqi
Decrypted message: hello world
```

Result :

The program encrypts and decrypts a message using a Caesar cipher with a specified shift. It demonstrates how text can be securely encoded and decoded.