BLACKBOXAI

Generated on: 5/24/2025 8:39:00 AM

User

for my lab exam to write aim,requirements,algorithm,procedure,configuration,code/implementation,expected output/result for this test case Port Range Analysis.

BLACKBOXAI

Certainly! Below is a structured outline for a lab exam on "Port Range Analysis," including the aim, requirements, algorithm, procedure, configuration, code/implementation, and expected output/result.

Aim

To analyze the availability and status of a specified range of network ports on a target host, determining which ports are open, closed, or filtered.

Requirements

- 1. Software Requirements:

- Python 3.x - `socket` library (standard library) - `threading` library (standard library)

- 2. Hardware Requirements:

- A computer with network access. - Target host (IP address or hostname) for port scanning.

- 3. Network Requirements:

- Permission to scan the target host. - Knowledge of the target host's IP address.

Algorithm

- 1. Define the target host and port range.
- 2. Create a function to check the status of a single port:

- Use the `socket` library to attempt a connection to the port. - Return the status (open, closed, or filtered).

- 3. Create a function to scan the specified range of ports:

- Use threading to speed up the scanning process. - Store the results in a list or dictionary.

- 4. Print the results after scanning all ports.

Procedure

- 1. Set up the Python environment and ensure all required libraries are available.
- 2. Write the code to implement the port scanning algorithm.
- 3. Execute the code with the desired target host and port range.
- 4. Analyze the output to determine the status of each port.

Configuration

- Define the target host and port range in the code.
- Example configuration:

```
python
 target_host = "192.168.1.1"  # Replace with the target IP address
 start_port = 20               # Starting port number
 end_port = 80                 # Ending port number
```

Code/Implementation

Here is a sample implementation in Python:

```
 python
import socket
import threading

Function to check the status of a single port


def check_port(host, port, results):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.settimeout(1)  # Set timeout for the connection attempt
    try:
        sock.connect((host, port))
        results[port] = 'Open'
    except (socket.timeout, ConnectionRefusedError):
        results[port] = 'Closed'
    except Exception:
        results[port] = 'Filtered'
```

```
    finally:
        sock.close()
```

Function to scan a range of ports

```
def scan_ports(host, start_port, end_port):
    results = {}
    threads = []
    for port in range(start_port, end_port + 1):
        thread = threading.Thread(target=check_port, args=(host, port, results))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()  # Wait for all threads to complete

    return results
```

Configuration

```
target_host = "192.168.1.1"  # Replace with the target IP address
start_port = 20               # Starting port number
end_port = 80                 # Ending port number
```

Execute the port scan

```
port_results = scan_ports(target_host, start_port, end_port)
```

Print the results

```
for port, status in port_results.items():
    print(f"Port {port}: {status}")
```

Expected Output/Result

The expected output will be a list of ports within the specified range along with their status. For example:

```
 Port 20: Closed
Port 21: Open
Port 22: Open
Port 23: Closed
Port 24: Filtered
...
Port 80: Open
```

This output indicates which ports are open, closed, or filtered for the specified target host.

User

test case 2 : input logging validation

BLACKBOXAI

Certainly! Below is a structured outline for a lab exam on "Input Logging Validation," including the aim, requirements, algorithm, procedure, configuration, code/implementation, and expected output/result.

Aim

To validate user input for a logging system, ensuring that the input meets specified criteria (e.g., format, length, and content) before being logged.

Requirements

- 1. Software Requirements:

- Python 3.x - `re` library (for regular expressions, standard library)

- 2. Hardware Requirements:

- A computer with Python installed.

- 3. Input Requirements:

- User input must meet specific validation criteria (e.g., alphanumeric, length restrictions).

Algorithm

- 1. Define the validation criteria for the input (e.g., must be alphanumeric, length between 5 and 20 characters).
- 2. Create a function to validate the input:

- Check if the input matches the criteria using regular expressions. - Return a success message if valid, or an error message if invalid.

- 3. If the input is valid, log it to a file.
- 4. Print the result of the validation.

Procedure

- 1. Set up the Python environment and ensure all required libraries are available.
- 2. Write the code to implement the input validation and logging.
- 3. Execute the code and provide various inputs to test the validation.
- 4. Analyze the output to determine if the input was logged correctly.

Configuration

- Define the validation criteria in the code.
- Example configuration:

```
python
 min_length = 5
 max_length = 20
```

Code/Implementation

Here is a sample implementation in Python:

```
python
import re

Function to validate input


def validate_input(user_input):
    # Define the validation criteria
    pattern = r'^[a-zA-Z0-9]{5,20}$'  # Alphanumeric, length between 5 and 20
    if re.match(pattern, user_input):
        return True
    return False

Function to log valid input


def log_input(user_input):
    with open("input_log.txt", "a") as log_file:
        log_file.write(user_input + "n")

Main function to get user input and validate


def main():
    user_input = input("Enter your input (5-20 alphanumeric characters): ")

    if validate_input(user_input):
```

```
        log_input(user_input)
        print("Input is valid and has been logged.")
    else:
        print("Invalid input. Please ensure it is alphanumeric and between 5 and 20
characters.")

Execute the main function


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

Expected Output/Result

The expected output will depend on the user input. Here are a few scenarios:

- 1. Valid Input:

- Input: `Test123` - Output:

```
Input is valid and has been logged.
```

- The input will be logged in `input_log.txt`.

- 2. Invalid Input (too short):

- Input: `123` - Output:

```
Invalid input. Please ensure it is alphanumeric and between 5 and 20 characters.
```

- 3. Invalid Input (too long):

- Input: `ThisInputIsWayTooLong123` - Output:

```
Invalid input. Please ensure it is alphanumeric and between 5 and 20 characters.
```

- 4. Invalid Input (non-alphanumeric):

- Input: `Test@123` - Output:

```
Invalid input. Please ensure it is alphanumeric and between 5 and 20 characters.
```

This output indicates whether the input was valid and whether it was logged successfully. The log file will contain all valid inputs.

User

test case : url vulnarability assessment

BLACKBOXAI

Certainly! Below is a structured outline for a lab exam on "URL Vulnerability Assessment," including the aim, requirements, algorithm, procedure, configuration, code/implementation, and expected output/result.

Aim

To assess the security of a given URL by checking for common vulnerabilities such as SQL injection, Cross-Site Scripting (XSS), and Open Redirect vulnerabilities.

Requirements

- 1. Software Requirements:

- Python 3.x - `requests` library (for making HTTP requests) - `re` library (for regular expressions, standard library)

- 2. Hardware Requirements:

- A computer with network access. - Target URL for vulnerability assessment.

- 3. Network Requirements:

- Permission to test the target URL for vulnerabilities.

Algorithm

- 1. Define the target URL.
- 2. Create functions to test for specific vulnerabilities:

- SQL Injection: Append common SQL injection payloads to the URL and check for error messages. - XSS: Append common XSS payloads to the URL and check for reflected input. - Open Redirect: Check if the URL redirects to an external site when a specific parameter is manipulated.

- 3. Collect and display the results of the assessments.

Procedure

- 1. Set up the Python environment and ensure all required libraries are available.

- 2. Write the code to implement the vulnerability assessment algorithm.
- 3. Execute the code with the desired target URL.
- 4. Analyze the output to determine the presence of vulnerabilities.

Configuration

- Define the target URL and the payloads for testing in the code.
- Example configuration:

```
python
 target_url = "http://example.com/vulnerable_endpoint"
 sql_payloads = ["' OR '1'='1", "'; DROP TABLE users; --"]
 xss_payloads = ["", "'>"]
```

Code/Implementation

Here is a sample implementation in Python:

```
 python
import requests

Function to test for SQL Injection


def test_sql_injection(url, payloads):
    for payload in payloads:
        response = requests.get(url + payload)
        if "error" in response.text.lower() or "sql" in response.text.lower():
            print(f"SQL Injection vulnerability found with payload: {payload}")
            return True
    return False

Function to test for XSS


def test_xss(url, payloads):
    for payload in payloads:
        response = requests.get(url + payload)
        if payload in response.text:
            print(f"XSS vulnerability found with payload: {payload}")
            return True
    return False

Function to test for Open Redirect
```

```
def test_open_redirect(url):
    redirect_url = url + "?redirect=http://malicious.com"
    response = requests.get(redirect_url)
    if response.url == "http://malicious.com/":
        print("Open Redirect vulnerability found.")
        return True
    return False

Main function to perform vulnerability assessment


def main():
    target_url = "http://example.com/vulnerable_endpoint"  # Replace with the target URL
    sql_payloads = ["' OR '1'='1", "'; DROP TABLE users; --"]
    xss_payloads = ["", "'>"]
    print("Starting vulnerability assessment...")

    if test_sql_injection(target_url, sql_payloads):
        print("SQL Injection vulnerability detected.")
    else:
        print("No SQL Injection vulnerabilities found.")
    if test_xss(target_url, xss_payloads):
        print("XSS vulnerability detected.")
    else:
        print("No XSS vulnerabilities found.")
    if test_open_redirect(target_url):
        print("Open Redirect vulnerability detected.")
    else:
        print("No Open Redirect vulnerabilities found.")

Execute the main function


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

Expected Output/Result

The expected output will depend on the target URL and its vulnerabilities. Here are a few scenarios:

- 1. Vulnerable URL:

- Output:

```
Starting vulnerability assessment...
     SQL Injection vulnerability found with payload: ' OR '1'='1
     SQL Injection vulnerability detected.
     XSS vulnerability found with payload:
     XSS vulnerability detected.
     Open Redirect vulnerability found.
     Open Redirect vulnerability detected.
```

- 2. Non-Vulnerable URL:

- Output:

```
Starting vulnerability assessment...
     No SQL Injection vulnerabilities found.
     No XSS vulnerabilities found.
     No Open Redirect vulnerabilities found.
```

This output indicates whether the target URL is vulnerable to SQL injection, XSS, or Open Redirect attacks. Always ensure you have permission to test the target URL for vulnerabilities.

User

test case : otp-based 2fa verification

BLACKBOXAI

Certainly! Below is a structured outline for a lab exam on "OTP-Based 2FA Verification," including the aim, requirements, algorithm, procedure, configuration, code/implementation, and expected output/result.

Aim

To implement an OTP (One-Time Password) based two-factor authentication (2FA) system that enhances security by requiring users to enter a password and a time-sensitive OTP sent to their registered mobile number or email.

Requirements

- 1. Software Requirements:

- Python 3.x - `random` library (for generating OTPs, standard library) - `time` library (for handling time-sensitive operations, standard library) - `smtplib` library (for sending emails, standard library) or a third-party SMS API for sending OTPs via SMS.

- 2. Hardware Requirements:

- A computer with Python installed. - Access to an email account or SMS service for sending OTPs.

- 3. User Requirements:

- User must have a registered email or mobile number for receiving OTPs.

Algorithm

- 1. User logs in with their username and password.
- 2. If the username and password are correct, generate a random OTP.
- 3. Send the OTP to the user's registered email or mobile number.
- 4. Prompt the user to enter the OTP.
- 5. Validate the entered OTP against the generated OTP.
- 6. If the OTP is valid, grant access; otherwise, deny access.

Procedure

- 1. Set up the Python environment and ensure all required libraries are available.
- 2. Write the code to implement the OTP-based 2FA system.
- 3. Execute the code and test the login and OTP verification process.
- 4. Analyze the output to determine if the authentication was successful.

Configuration

- Define the user credentials and the method for sending OTPs in the code.
- Example configuration:

```
python
 user_credentials = {"username": "user1", "password": "securepassword"}
```

Code/Implementation

Here is a sample implementation in Python:

```
python
import random
import smtplib
```

```
import time
```

Function to send OTP via email

```
def send_otp_via_email(email, otp):
    sender_email = "your_email@example.com"  # Replace with your email
    sender_password = "your_password"          # Replace with your email password
    subject = "Your OTP Code"
    body = f"Your OTP code is: {otp}"

    message = f"Subject: {subject}nn{body}"

    with smtplib.SMTP("smtp.gmail.com", 587) as server:
        server.starttls()
        server.login(sender_email, sender_password)
        server.sendmail(sender_email, email, message)
```

Function to generate a random OTP

```
def generate_otp():
    return random.randint(100000, 999999)  # 6-digit OTP
```

Main function for OTP-based 2FA

```
def main():
    # User credentials
    user_credentials = {"username": "user1", "password": "securepassword"}

    # User login
    username = input("Enter your username: ")
    password = input("Enter your password: ")

    if username == user_credentials["username"] and password ==
user_credentials["password"]:
        print("Login successful. Generating OTP...")

        otp = generate_otp()
        print(f"Generated OTP: {otp}")  # For testing purposes, you can comment this out
in production
        send_otp_via_email("user_email@example.com", otp)  # Replace with the user's
email

        # Wait for the user to enter the OTP
        user_otp = int(input("Enter the OTP sent to your email: "))

        if user_otp == otp:
```

```
            print("OTP verified successfully. Access granted.")
        else:
            print("Invalid OTP. Access denied.")
    else:
        print("Invalid username or password.")

Execute the main function


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

Expected Output/Result

The expected output will depend on the user input and the OTP verification process. Here are a few scenarios:

- 1. Successful Login and OTP Verification:

- Input:

```
Enter your username: user1
     Enter your password: securepassword
     Enter the OTP sent to your email: 123456  (Assuming this is the generated OTP)
```

- Output:

```
Login successful. Generating OTP...
     Generated OTP: 123456
     OTP verified successfully. Access granted.
```

- 2. Invalid Username or Password:

- Input:

```
Enter your username: user1
     Enter your password: wrongpassword
```

- Output:

```
Invalid username or password.
```

- 3. Invalid OTP:

- Input:

```
Enter your username: user1
    Enter your password: securepassword
    Enter the OTP sent to your email: 654321  (Assuming this is not the generated OTP)
```

- Output:

```
Login successful. Generating OTP...
    Generated OTP: 123456
    Invalid OTP. Access denied.
```

This output indicates whether the user successfully logged in and whether the OTP verification was successful. Make sure to replace the email and password with valid credentials for sending the OTP.

User

test case 5: caesar cipher functionality check

BLACKBOXAI

Certainly! Below is a structured outline for a lab exam on "Caesar Cipher Functionality Check," including the aim, requirements, algorithm, procedure, configuration, code/implementation, and expected output/result.

Aim

To implement and verify the functionality of the Caesar Cipher, which is a basic encryption technique that shifts the letters of the alphabet by a fixed number of positions.

Requirements

- 1. Software Requirements:

- Python 3.x

- 2. Hardware Requirements:

- A computer with Python installed.

- 3. Input Requirements:

- A plaintext message to be encrypted. - A shift value (key) for the Caesar Cipher.

Algorithm

- 1. Define a function to encrypt the plaintext using the Caesar Cipher:

- For each character in the plaintext: - If it is a letter, shift it by the specified key. - If it is not a letter, leave it unchanged.

- 2. Define a function to decrypt the ciphertext using the Caesar Cipher:

- For each character in the ciphertext: - If it is a letter, shift it back by the specified key. - If it is not a letter, leave it unchanged.

- 3. Return the encrypted or decrypted message.

Procedure

- 1. Set up the Python environment and ensure all required libraries are available.
- 2. Write the code to implement the Caesar Cipher functionality.
- 3. Execute the code with various test cases to verify the encryption and decryption processes.
- 4. Analyze the output to ensure correctness.

Configuration

- Define the plaintext message and the shift value in the code.
- Example configuration:

```
python
 plaintext = "Hello, World!"
 shift = 3
```

Code/Implementation

Here is a sample implementation in Python:

```
python
def encrypt(plaintext, shift):
    encrypted_message = ""
    for char in plaintext:
        if char.isalpha():  # Check if the character is a letter
            shift_base = ord('A') if char.isupper() else ord('a')
            encrypted_char = chr((ord(char) - shift_base + shift) % 26 + shift_base)
            encrypted_message += encrypted_char
        else:
            encrypted_message += char  # Non-letter characters remain unchanged
```

```
    return encrypted_message

def decrypt(ciphertext, shift):
    decrypted_message = ""
    for char in ciphertext:
        if char.isalpha():  # Check if the character is a letter
            shift_base = ord('A') if char.isupper() else ord('a')
            decrypted_char = chr((ord(char) - shift_base - shift) % 26 + shift_base)
            decrypted_message += decrypted_char
        else:
            decrypted_message += char  # Non-letter characters remain unchanged
    return decrypted_message

Main function to test the Caesar Cipher


def main():
    plaintext = "Hello, World!"  # Example plaintext
    shift = 3                    # Example shift value

    print("Plaintext:", plaintext)
    encrypted = encrypt(plaintext, shift)
    print("Encrypted:", encrypted)

    decrypted = decrypt(encrypted, shift)
    print("Decrypted:", decrypted)

Execute the main function


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

Expected Output/Result

The expected output will show the original plaintext, the encrypted message, and the decrypted message. Here are the results for the provided example:

- 1. Input:

- Plaintext: `Hello, World!` - Shift: `3`

- 2. Output:

```
Plaintext: Hello, World!
  Encrypted: Khoor, Zruog!
  Decrypted: Hello, World!
```

Additional Test Cases

You can also test with different inputs and shift values:

- 1. Test Case 1:

- Input: `Caesar Cipher!` - Shift: `5` - Output:

```
Plaintext: Caesar Cipher!
     Encrypted: Hfjxfw Hnumjw!
     Decrypted: Caesar Cipher!
```

- 2. Test Case 2:

- Input: `Python 3.9` - Shift: `2` - Output:

```
Plaintext: Python 3.9
     Encrypted: Rjvqpk 3.9
     Decrypted: Python 3.9
```

This output indicates that the Caesar Cipher functionality is working correctly for both encryption and decryption processes.