MANAKULA VINAYAGAR INSTITUTE OF TECHNOLOGY
Approved by AICTE, New Delhi and Affiliated to Pondicherry University
Accredited by NBA & NAAC 'A' Grade
Kalitheerthalkuppam, Puducherry - 605107

SECURE VPN COMMUNICATION USING AES ENCRYPTION ALGORITHM

COMPUTER NETWORK

SUBMITTED BY

SITHARTH S

GURUSARAN S

BENNY CHRISTIYAN.J

III CSE B

COMPUTER NEWORKS

1. Generate Secure VPN Communication using AES encrytion algorithm using PYTHON

Client Program:

```
import socket
import webbrowser
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import os
import tkinter as tk
from tkinter import simpledialog
import time
import sys
# Generate a new ECDH key pair for the client
private_key = ec.generate_private_key(ec.SECP256R1())
public_key = private_key.public_key()
# Serialize public key to send to the server
def serialize_public_key(public_key):
```

```
    return public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
# Deserialize the server's public key
def deserialize_public_key(pem_data):
    return serialization.load_pem_public_key(pem_data)
# Encrypt data using AES-GCM
def encrypt_data(key, data):
    nonce = os.urandom(12)
    cipher = Cipher(algorithms.AES(key), modes.GCM(nonce))
    encryptor = cipher.encryptor()
    encrypted_data = encryptor.update(data) + encryptor.finalize()
    return nonce + encryptor.tag + encrypted_data

# Decrypt data using AES-GCM
def decrypt_data(key, encrypted_data):
    nonce = encrypted_data[:12]
    tag = encrypted_data[12:28]
    ciphertext = encrypted_data[28:]
    cipher = Cipher(algorithms.AES(key), modes.GCM(nonce, tag))
    decryptor = cipher.decryptor()
    return decryptor.update(ciphertext) + decryptor.finalize()

# Function to handle key entry and validation
```

```
def verification(expected_key, max_attempts=3, retry_delay=10):
    attempts = 0
    while attempts < max_attempts:
        root = tk.Tk()
        root.withdraw()  # Hide the main tkinter window
        # Mask the input text like a password
        user_input = simpledialog.askstring("Shared Key", "Enter the shared key:", show='*')
        if user_input is None:
            print("Operation canceled by the user.")
            sys.exit()
        if user_input == expected_key:
            print("Key entered correctly!")
            return True
        else:
            attempts += 1
            print(f"Incorrect key. {max_attempts - attempts} attempts left.")
            root.destroy()
    # Countdown after reaching maximum attempts
    print(f"Maximum attempts reached. Please wait {retry_delay} seconds before retrying.")
    for i in range(retry_delay, 0, -1):
        sys.stdout.write(f"\rRetrying in {i} seconds...")
        sys.stdout.flush()
        time.sleep(1)
    # Clear the countdown message from the terminal
    sys.stdout.write("\r" + " " * 50 + "\r")
```

```
    sys.stdout.flush()
    print("\n")
    return False
# Client setup
def start_client(server_ip='127.0.0.1', port=65432, request_url='http://google.com'):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((server_ip, port))
    print("Connected with server")
    # Send client's public key
    client_socket.send(serialize_public_key(public_key))
    # Receive server's public key
    server_public_key_pem = client_socket.recv(1024)
    server_public_key = deserialize_public_key(server_public_key_pem)
    # Generate shared key
    shared_secret = private_key.exchange(ec.ECDH(), server_public_key)
    key = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=b'salt',
        iterations=100000,
    ).derive(shared_secret)
    shared_key_hex = key.hex()
    print("Shared key:", shared_key_hex)
    # Wait for the correct key entry
    while not verification(shared_key_hex):
```

```
            continue
        print("Key exchange successful")
        # Encrypt and send the request
        encrypted_request = encrypt_data(key, request_url.encode())
        print("Encrypted request to server:", encrypted_request)
        client_socket.send(encrypted_request)
        # Receive and decrypt the response
        data = b""
        while True:
            part = client_socket.recv(4096)
            if not part:  # No more data
                break
            data += part
        encrypted_response = data
        print("Encrypted response from server:", encrypted_response)
        decrypted_response = decrypt_data(key, encrypted_response).decode()
        # Close the client socket
        client_socket.close()
        # Open the decrypted URL in a web browser
        if decrypted_response.startswith('http'):
            webbrowser.open(decrypted_response)
        else:
            print("Received response is not a valid URL.")
    if __name__ == "__main__":
        start_client(request_url='http://google.com')
```

Client Program:

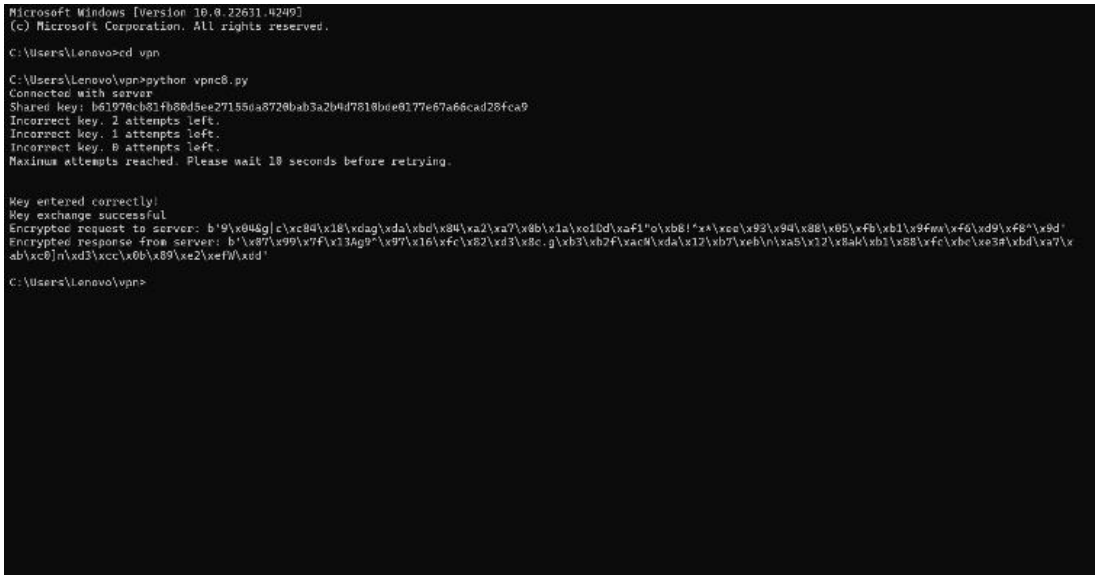
```
import socket
import requests
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import os
# Generate a new ECDH key pair for the server
private_key = ec.generate_private_key(ec.SECP256R1())
public_key = private_key.public_key()
# Serialize public key to send to the client
def serialize_public_key(public_key):
    return public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
# Deserialize the client's public key
def deserialize_public_key(pem_data):
    return serialization.load_pem_public_key(pem_data)
# Encrypt data using AES-GCM
def encrypt_data(key, data):
    nonce = os.urandom(12)
    cipher = Cipher(algorithms.AES(key), modes.GCM(nonce))
```

```
    encryptor = cipher.encryptor()
    encrypted_data = encryptor.update(data) + encryptor.finalize()
    return nonce + encryptor.tag + encrypted_data
# Decrypt data using AES-GCM
def decrypt_data(key, encrypted_data):
    nonce = encrypted_data[:12]
    tag = encrypted_data[12:28]
    ciphertext = encrypted_data[28:]
    cipher = Cipher(algorithms.AES(key), modes.GCM(nonce, tag))
    decryptor = cipher.decryptor()
    return decryptor.update(ciphertext) + decryptor.finalize()
# Handle client connection
def handle_client(client_socket):
    print(f"Connection from {client_socket.getpeername()}")
    # Send server's public key
    client_socket.send(serialize_public_key(public_key))
    # Receive client's public key
    client_public_key_pem = client_socket.recv(1024)
    client_public_key = deserialize_public_key(client_public_key_pem)
    # Generate shared key
    shared_secret = private_key.exchange(ec.ECDH(), client_public_key)
    key = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=b'salt',
```

```
        iterations=100000,
    ).derive(shared_secret)
    #print("Key exchange successful")
    # Receive encrypted request
    encrypted_request = client_socket.recv(4096)
    print(f"Encrypted request from client: {encrypted_request}")
    decrypted_request = decrypt_data(key, encrypted_request).decode()
    print(f"Decrypted request from client: {decrypted_request}")
    # Fetch the actual content
    response = requests.get(decrypted_request)
    # Encrypt and send the response
    encrypted_response = encrypt_data(key, response.url.encode())
    client_socket.send(encrypted_response)
    client_socket.close()
# Server setup
def start_server(host='0.0.0.0', port=8000):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen(5)
    print(f"Server listening on {host}:{port}")
    while True:
        client_socket, addr = server_socket.accept()
        handle_client(client_socket)
if __name__ == "__main__":
    start_server()
```

OUTPUT :

Server Output



Client Output

RESULT:

Thus the program for Secure VPN Communication using AES encrytion algorithm using PYTHON was verified.