INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

SECURITY ARCHITECTURE AND PROTOCOLS LABORATORY								
VII Semester: CSE(CS)								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
		L	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
ACCD26	**Core**	0	0	2	1	40	60	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisites: Basic knowledge of operating systems, computer networks, programming in Java or Python, and foundational concepts in cryptography and security.								

I. COURSE OVERVIEW:

This laboratory course provides hands-on experience with security architecture, cryptographic protocols, and network security mechanisms. Students will implement and analyze security protocols, explore authentication mechanisms, and work with secure communication frameworks to understand real-world cybersecurity challenges.

II. COURSE OBJECTIVES:

The students will try to learn:

I. Fundamental security principles, cryptographic algorithms, and secure communication protocols.
II. Implementation of authentication, authorization, and access control mechanisms.
III. Secure network communication using TLS/SSL, VPN, and other cryptographic protocols.
IV. Analysis and evaluation of security vulnerabilities in network protocols.
V. Practical applications of security frameworks and tools for protocol security.

III. COURSE OUTCOMES:

CO1 Demonstrate an understanding of security architectures and cryptographic protocols

CO2 Implement authentication mechanisms and secure communication techniques.

CO3 Analyze vulnerabilities in network protocols and propose mitigation strategies.

CO4 Configure and deploy security protocols for protecting network infrastructure.

CO5 Evaluate security performance using penetration testing and threat analysis tools.

CO6 Design and implement secure system architectures and security protocols using cryptographic techniques to ensure confidentiality, integrity, and authentication in real-world applications.

SECURITY ARCHITECTURE AND PROTOCOLS LABORATORY (ACCD26)

CONTENT

S No.	Topic Name	Page No
1	Getting Started Exercises	1
	a) Overview of security models and cryptographic foundations	1
	b) Hands-on with basic encryption and decryption techniques	1
2	Symmetric And Asymmetric Cryptography Implementation	3
	a) Implementation of AES, DES, RSA algorithms	3
	b) Key generation, encryption, and decryption processes	4
3	Message Authentication and Hashing Techniques	
	a) Implementing SHA-256, HMAC, and digital signatures	6
	b) Integrity verification of transmitted messages	7
4	Secure Key Exchange Protocols	8
	a) Implementing Diffie-Hellman and ECDH key exchange	8
	b) Analysis of Man-in-the-Middle (MITM) attacks	9
5	Public Key Infrastructure (PKI) and Digital Certificates	10
	a) Generating and managing X.509 certificates	10
	b) Configuring Certificate Authority (CA) and certificate validation	11
6	Secure Socket Layer (SSL) and Transport Layer Security (TLS)	12
	a) Hands-on SSL/TLS encryption using OpenSSL	12
	b) Analyzing SSL/TLS handshake and vulnerabilities	13
7	Virtual Private Networks (VPN) and Secure Communication	14
	a) Setting up VPN using Open VPN and Wire Guard	14
	b) Secure tunneling and traffic encryption	15
8	Authentication Protocols: Kerberos and Oauth	15
	a) Implementing Kerberos authentication in a network	17
	b) Configuring OAuth for secure API access	16
9	Secure Email Communication With PGP/GPG	17
	a) Encryption and decryption of emails using GPG	17
	b) Verifying email authenticity using PGP signatures	18
10	Wireless Security and WPA2 Encryption	19
	a) Implementing WPA2 security for Wi-Fi networks	19
	b) Analyzing wireless security threats and defenses	20
11	Web Security and HTTPS Configuration	21
	a) Configuring HTTPS with TLS on web servers	22
	b) Preventing web-based attacks (XSS, CSRF, SQL Injection)	22
12	Intrusion Detection and Prevention Systems (IDS/IPS)	23
	a) Deploying Snort and Suricata for network intrusion detection	23
	b) Traffic analysis and threat detection	24
13	Penetration Testing and Ethical Hacking Techniques	25

a) Using Kali Linux tools for security testing 25
b) Exploiting and securing vulnerable network services 26
14 Security Evaluation and Final Project Presentation 27
a) Analyzing security risks in a given system 27
b) Group projects on designing secure architectures 28

IV. COURSE CONTENT:

EXERCISES FOR SECURITY ARCHITECTURE AND PROTOCOLS LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions. All exercises are for educational purposes only and use synthetic, non-sensitive sample data; no exercise involves attacking, scanning, or otherwise interacting with systems or networks without authorization.

1. Getting Started Exercises

1.1 Overview of security models and cryptographic foundations

Every security control ultimately exists to protect one or more of the three pillars of the CIA triad: Confidentiality, Integrity, and Availability. Before designing a countermeasure, a security architect first identifies which property an attack primarily threatens.

1.1 CIA Triad Violation Classifier

Write a function that accepts the name of an attack or incident type and returns the CIA property it primarily violates, using the mapping below. Recognise: "Eavesdropping" and "Data Theft" -> "CONFIDENTIALITY"; "Data Tampering" and "Man-in-the-Middle" -> "INTEGRITY"; "DDoS" and "Ransomware" -> "AVAILABILITY". Return "UNKNOWN" for anything else.

Input: "Eavesdropping"

Output: "CONFIDENTIALITY"

Input: "DDoS"

Output: "AVAILABILITY"

Input: "Phishing"

Output: "UNKNOWN"

```
public class CiaTriadClassifier
{
public static String classify(String attackType)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

1.2 Vigenere Cipher Encoder

Simple substitution ciphers like the Caesar cipher are easy to break with frequency analysis, so early cryptographic protocols moved to polyalphabetic ciphers that shift each letter by a different amount. The Vigenere cipher is a classic example and a useful warm-up before studying modern stream ciphers.

Write a program that encrypts a message using the Vigenere cipher, given a repeating keyword. Each letter of the message is shifted forward by the position (0-25) of the corresponding letter of the keyword, wrapping the keyword as needed. Non-alphabetic characters are left unchanged and do not consume a keyword letter. Case is preserved.

Input: message = "attack at dawn", key = "lemon"

Output: "lxfopv ef rgys"

Input: message = "HELLO", key = "key"

Output: "RIJVS"

```
public class VigenereCipher
{
public static String encrypt(String message, String key)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

b) Hands-on with basic encryption and decryption techniques

1.3 Repeating-Key XOR Cipher

A repeating-key XOR cipher is a simple stream cipher and a good illustration of why key reuse is dangerous in real protocols: because XOR is its own inverse, the same function both encrypts and decrypts once the key stream is known.

Write a function that accepts a byte array and a key byte array, and returns a new byte array where each byte of the data has been XOR-ed with the corresponding byte of the key, repeating the key as needed.

Input: data = bytes of "HELLO", key = bytes of "AB"

Output: bytes = [9, 4, 13, 6, 15]

Input: data = [9, 4, 13, 6, 15], key = bytes of "AB"

Output: bytes of "HELLO"

```
public class RepeatingKeyXor
{
public static byte[] apply(byte[] data, byte[] key)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

1.4 Base64 Ciphertext Decoder

Ciphertext and cryptographic key material are frequently transmitted as Base64 text so they survive text-only channels such as email or JSON payloads. Before the underlying bytes can be decrypted, the Base64 wrapper must be removed.

Write a function that decodes a Base64-encoded string into the original plain-text string it represents.

Note: use a standard library decoder (e.g. java.util.Base64) rather than implementing the encoding table by hand.

Input: "U2VjcmV0IE1lc3NhZ2U="

Output: "Secret Message"

Input: "RXZpZGVuY2U="

Output: "Evidence"

```
import java.util.Base64;

public class Base64CiphertextDecoder
{
public static String decode(String encoded)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

2. Symmetric and Asymmetric Cryptography Implementation

a) Implementation of AES, DES, RSA algorithms

2.1 AES Key Size Validator

AES supports three standard key sizes, and a security architecture must reject any key that does not match one of them before it is used for encryption.

Write a function that accepts a key as a byte array and returns "AES-128", "AES-192", or "AES-256" based on its length (16, 24, or 32 bytes respectively). Return "INVALID_KEY_SIZE" for any other length.

Input: key = new byte[16]

Output: "AES-128"

Input: key = new byte[32]

Output: "AES-256"

Input: key = new byte[20]

Output: "INVALID_KEY_SIZE"

```
public class AesKeySizeValidator
{
public static String validate(byte[] key)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

2.2 RSA Encryptor / Decryptor (Small Numbers)

RSA encryption and decryption are both modular exponentiation operations, differing only in which exponent is used. Working through the arithmetic with small numbers builds the intuition needed before using a real library implementation.

Write two functions: encrypt(m, e, n) returns the ciphertext c = m^e mod n, and decrypt(c, d, n) returns the plaintext m = c^d mod n. Use modular exponentiation rather than computing the full power directly.

Input: encrypt: m = 7, e = 5, n = 143

Output: 106

Input: decrypt: c = 106, d = 77, n = 143

Output: 7

```
public class RsaCalculator
{
public static long modPow(long base, long exponent, long modulus)
{
# Write code here
}

public static long encrypt(long m, long e, long n)
{
# Write code here
}

public static long decrypt(long c, long d, long n)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

b) Key generation, encryption, and decryption processes

2.3 PKCS7 Block Padder / Unpadder

Block ciphers such as AES and DES operate on fixed-size blocks, so plaintext whose length is not a multiple of the block size must be padded before encryption, and the padding must be verified and stripped after decryption.

Write a pad(data, blockSize) function that appends PKCS7 padding so the result length is a multiple of blockSize (padding bytes each equal the number of padding bytes added; if data is already a multiple of blockSize, a full block of padding is added). Write a matching unpad(data) function that validates and removes the padding, throwing an IllegalArgumentException if the padding is invalid.

Input: pad(bytes of length 5, blockSize = 8)

Output: bytes of length 8, last 3 bytes = [3, 3, 3]

Input: unpad(bytes of length 8 ending in [3, 3, 3])

Output: bytes of length 5

```
public class Pkcs7Padder
{
public static byte[] pad(byte[] data, int blockSize)
{
# Write code here
}

public static byte[] unpad(byte[] data)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

2.4 Deterministic Symmetric Key Generator (Test Harness)

Automated test suites for encryption pipelines need reproducible keys, so a seeded key generator is used in place of a true random source during testing only; production systems must always use a cryptographically secure random generator.

Write a function that accepts a key length in bits and a long seed, and returns a hex-encoded string of (bits / 8) deterministic key bytes generated from a seeded java.util.Random.

Note: this generator is for reproducible testing only and must never be used to derive keys for real encryption.

Input: bits = 128, seed = 42

Output: a 32-character lowercase hex string, reproducible for the same seed

Input: bits = 256, seed = 42

Output: a 64-character lowercase hex string, reproducible for the same seed

```
import java.util.Random;

public class SeededKeyGenerator
{
public static String generateKeyHex(int bits, long seed)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

3. Message Authentication and Hashing Techniques

a) **Implementing SHA-256, HMAC, and digital signatures**

3.1 Message Digest Calculator

A cryptographic hash function condenses an arbitrary-length message into a fixed-length digest that is used throughout security protocols for integrity checks, digital signatures, and password storage.

Write a function that accepts a message string and returns its SHA-256 hash as a lowercase hexadecimal string.

Note: use a standard library hashing implementation (e.g. java.security.MessageDigest) rather than writing your own hash function.

Input: message = "hello"

Output: "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"

Input: message = ""

Output: "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"

```
import java.security.MessageDigest;

public class MessageDigestCalculator
{
public static String sha256(String message)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

3.2 HMAC-SHA256 Generator

A plain hash alone cannot prove who sent a message, since anyone can compute it. An HMAC combines the message with a shared secret key, so only someone holding the key can produce a valid tag, making it the basis of message authentication in protocols such as TLS.

Write a function that accepts a message and a shared secret key (both strings) and returns the HMAC-SHA256 tag as a lowercase hexadecimal string.

Note: use a standard library MAC implementation (e.g. javax.crypto.Mac with "HmacSHA256") rather than implementing HMAC by hand.

Input: message = "transfer $100", key = "shared-secret"

Output: a 64-character lowercase hex HMAC tag

```
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

public class HmacGenerator
{
```

```
public static String generateHmac(String message, String key)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

b) Integrity verification of transmitted messages

3.3 Message Integrity Checker

When a message is received over an untrusted channel, the receiver recomputes the digest of the received content and compares it against the digest sent alongside (or out-of-band) by the sender to detect any modification in transit.

Write a function that accepts the received message text and an expected SHA-256 digest (lowercase hex string), and returns "INTACT" if the message's own digest matches the expected digest, or "TAMPERED" otherwise.

Input: message = "hello", expectedDigest = "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"

Output: "INTACT"

Input: message = "hellO", expectedDigest = "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"

Output: "TAMPERED"

```
public class MessageIntegrityChecker
{
public static String verify(String message, String expectedDigest)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

3.4 Simplified Signature Verifier

A full digital-signature scheme relies on asymmetric keys, but the verification logic can be illustrated with a shared-key HMAC standing in for a signature: the verifier recomputes the expected tag over the message and checks it against the tag that accompanied the message.

Write a function that accepts a message, a shared secret key, and a received signature (hex string), recomputes the HMAC-SHA256 of the message under the key, and returns "VALID" if it matches the received signature, or "INVALID" otherwise.

Input: message = "approve payment", key = "sig-key", signature = "<matching HMAC hex>"

Output: "VALID"

Input: message = "approve payment", key = "sig-key", signature = "0000...0000"

Output: "INVALID"

```
public class SignatureVerifier
{
public static String verify(String message, String key, String signature)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

4. Secure Key Exchange Protocols

a) Implementing Diffie-Hellman and ECDH key exchange

4.1 Diffie-Hellman Shared Secret Calculator

The Diffie-Hellman key exchange lets two parties agree on a shared secret over an insecure channel without ever transmitting the secret itself, and it underlies key exchange in protocols such as TLS and IPsec.

Write a function that accepts a public base g, a prime modulus p, and a private exponent x, and returns the value g^x mod p that would be published to (or received from) the other party. Use modular exponentiation rather than computing the full power directly.

Input: g = 5, p = 23, x = 6

Output: 8

Input: g = 5, p = 23, x = 15

Output: 19

```
public class DiffieHellmanCalculator
{
public static long modPow(long g, long x, long p)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

4.2 Shared Secret Agreement Verifier

After each party independently combines the other party's public value with their own private exponent, both sides should arrive at the same shared secret. This exercise mirrors that final agreement step of Diffie-Hellman and ECDH.

Write a function that accepts Alice's public value A, Bob's private exponent b, and the prime modulus p, computes Bob's view of the shared secret as A^b mod p, and returns whether it equals a given expectedSecret ("AGREED" or "MISMATCH").

Input: A = 8, b = 15, p = 23, expectedSecret = 2

Output: "AGREED"

Input: A = 8, b = 15, p = 23, expectedSecret = 99

Output: "MISMATCH"

```
public class SharedSecretVerifier
{
public static String verify(long A, long b, long p, long expectedSecret)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

b) Analysis of Man-in-the-Middle (MITM) attacks

4.3 Key Exchange Tamper Detector

An unauthenticated Diffie-Hellman exchange is vulnerable to a man-in-the-middle attack, in which an interceptor substitutes their own public value for the one each party actually sent. A monitoring tool can flag this by comparing the value a party received against the value that was genuinely transmitted (obtained out-of-band, e.g. over a separately authenticated channel).

Write a function that accepts the public value Bob actually sent (sentValue) and the public value Alice received (receivedValue), and returns "TAMPERED" if they differ (indicating possible MITM interception) or "CLEAN" if they match.

Input: sentValue = 19, receivedValue = 19

Output: "CLEAN"

Input: sentValue = 19, receivedValue = 7

Output: "TAMPERED"

```
public class KeyExchangeTamperDetector
{
public static String check(long sentValue, long receivedValue)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

4.4 Certificate Fingerprint Comparator

One practical defence against MITM key substitution is to verify the other party's certificate fingerprint through a second, independently trusted channel (e.g. reading it aloud on a phone call) before trusting a key exchange.

Write a function that accepts two SHA-256 certificate fingerprints, obtained via two different channels, as lowercase hex strings, and returns "MATCH" if they are identical, or "MISMATCH - POSSIBLE MITM" if they differ.

Input: fingerprintA = "3a7bd3e2...", fingerprintB = "3a7bd3e2..." (identical)

Output: "MATCH"

Input: fingerprintA = "3a7bd3e2...", fingerprintB = "9f86d081..." (different)

Output: "MISMATCH - POSSIBLE MITM"

```
public class CertificateFingerprintComparator
{
public static String compare(String fingerprintA, String fingerprintB)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

5. Public Key Infrastructure (PKI) and Digital Certificates

a) Generating and managing X.509 certificates

5.1 Generate a Self-Signed X.509 Certificate with OpenSSL

Before a service can offer TLS, it needs a key pair and a certificate binding that key to an identity. A self-signed certificate is the simplest starting point for lab work, since it does not require a separate CA.

1. Generate a 2048-bit RSA private key.

```
openssl genrsa -out server.key 2048
```

2. Create a self-signed certificate valid for 365 days from the key, supplying a subject (Common Name, Organisation, etc.) when prompted.

```
openssl req -new -x509 -key server.key -out server.crt -days 365
```

3. View the certificate's fields in human-readable form to confirm the subject, validity dates, and public key algorithm.

```
openssl x509 -in server.crt -noout -text
```

Note: record the "Not Before" and "Not After" dates from the output; you will use them in exercise 5.2.

5.2 Certificate Expiry Checker

An expired certificate causes clients to reject a TLS connection outright, so monitoring tools continuously check certificate validity windows ahead of time so operators can renew before an outage occurs.

Write a function that accepts a certificate's not After date and the current date, both as "yyyy-MM-dd" strings, and returns "VALID" if the current date is on or before notAfter, or "EXPIRED" otherwise.

Input: notAfter = "2027-01-15", currentDate = "2026-08-01"

Output: "VALID"

Input: notAfter = "2025-12-01", currentDate = "2026-08-01"

Output: "EXPIRED"

```
import java.time.LocalDate;
 public class CertificateExpiryChecker
{
public static String checkStatus(String notAfter, String currentDate)
{
# Write code here
}
 public static void main(String[] args)
{
# Write code here
}
}
```

b) Configuring Certificate Authority (CA) and certificate validation

5.3 Build a Mini Certificate Authority and Sign a Server Certificate

In a real PKI, a Certificate Authority signs certificates so that any relying party who trusts the CA can trust every certificate it issues, without a separate out-of-band check for each one. Setting up a miniature CA locally demonstrates the full issue-and-sign workflow.

1. Create the CA's own private key and self-signed root certificate.

```
openssl genrsa -out ca.key 4096
openssl req -new -x509 -key ca.key -out ca.crt -days 3650
```

2. Create a private key and a Certificate Signing Request (CSR) for the server that will be issued a certificate.

```
openssl genrsa -out server2.key 2048
openssl req -new -key server2.key -out server2.csr
```

3. Use the CA's key and certificate to sign the CSR, producing a certificate the CA vouches for.

```
openssl x509 -req -in server2.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out server2.crt -days 365
```

4. Verify that the issued certificate is valid against the CA's root certificate.

```
openssl verify -CAfile ca.crt server2.crt
```

5.4 Certificate Chain Order Validator

A relying party validates a certificate chain by confirming each certificate was issued by the next certificate in the chain, ending in a self-signed root that the party already trusts.

Write a function that accepts an array of certificates, each represented as a "Subject|Issuer" string, ordered from the leaf certificate to the root. Return true if, for every certificate except the last, its Issuer equals the Subject of the next certificate in the array, and the last certificate's Issuer equals its own Subject (self-signed root). Return false otherwise.

Input: chain = ["server.example.com|LabCA", "LabCA|LabCA"]

Output: true

Input: chain = ["server.example.com|LabCA", "OtherCA|OtherCA"]

Output: false

```
public class CertificateChainValidator
{
public static boolean isValidChain(String[] chain)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

6. Secure Socket Layer (SSL) and Transport Layer Security (TLS)

a) Hands-on SSL/TLS encryption using OpenSSL

6.1 Run a Local TLS Server and Client with OpenSSL

OpenSSL ships with a minimal TLS server and client that are useful for observing a handshake end-to-end without deploying a full web server.

1. Start a local TLS test server on port 4433 using the certificate and key generated in Experiment 5.

```
openssl s_server -accept 4433 -cert server.crt -key server.key
```

2. In a second terminal, connect to the server as a TLS client.

```
openssl s_client -connect localhost:4433
```

3. Type a short message into the client terminal and confirm it appears on the server side, demonstrating an encrypted channel over the loopback interface.

6.2 Inspect a Certificate Presented Over TLS

A client should always be able to extract and inspect the certificate a server presents during the handshake, independent of any browser UI.

1. Connect to the local test server and save the raw certificate it presents.

```
openssl s_client -connect localhost:4433 -showcerts </dev/null | openssl x509 -outform PEM > presented.crt
```

2. Print the certificate's subject, issuer, and validity period.

```
openssl x509 -in presented.crt -noout -subject -issuer -dates
```

b) Analyzing SSL/TLS handshake and vulnerabilities

6.3 Capture and Read a TLS Handshake

Understanding what is negotiated in a handshake -- protocol version, cipher suite, and key exchange method -- is essential before assessing a TLS configuration for weaknesses.

1. Connect with verbose handshake tracing enabled and inspect the negotiated protocol version and cipher suite in the output.

```
openssl s_client -connect localhost:4433 -tls1_2 -msg </dev/null
```

2. Repeat the connection while explicitly restricting the client to an outdated protocol version, and observe whether the server accepts or rejects it.

```
openssl s_client -connect localhost:4433 -tls1 </dev/null
```

Note: if the server rejects the outdated protocol, that is the expected, secure behaviour -- modern OpenSSL builds disable TLS 1.0/1.1 by default.

6.4 Cipher Suite Strength Classifier

Not every cipher suite a server could theoretically negotiate is safe to allow; a security architecture review typically starts by classifying each configured cipher suite as acceptable or not before writing the server's allow-list.

Write a function that accepts a cipher suite name and returns "STRONG" if it contains "GCM" or "CHACHA20", "WEAK" if it contains "RC4" or "DES", or "REVIEW" for anything else.

Input: "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256"

Output: "STRONG"

Input: "TLS_RSA_WITH_RC4_128_SHA"

Output: "WEAK"

Input: "TLS_RSA_WITH_AES_128_CBC_SHA"

Output: "REVIEW"

```
public class CipherSuiteClassifier
{
public static String classify(String cipherSuiteName)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

7: Virtual Private Networks (VPN) and Secure Communication

a) Setting up VPN using OpenVPN and WireGuard

7.1 Generate a WireGuard Key Pair and Minimal Interface Config

WireGuard authenticates peers using Curve25519 public keys rather than certificates, so the first step in any WireGuard setup is generating a key pair for each peer.

1. Generate a private key and derive the matching public key.

```
wg genkey | tee wg-private.key | wg pubkey > wg-public.key
```

2. Create a minimal interface configuration file referencing the private key, an internal tunnel address, and a listening port.

```
# wg0.conf
[Interface]
PrivateKey = <contents of wg-private.key>
Address = 10.8.0.1/24
ListenPort = 51820
```

3. Add a [Peer] section with the remote peer's public key, allowed IP range, and endpoint once a second peer's keys are available.

7.2 Generate an OpenVPN Static Key and Server Config Skeleton

OpenVPN can be bootstrapped quickly for lab purposes with a pre-shared static key instead of a full PKI, which is convenient for understanding the tunnel mechanics before layering certificate-based authentication on top.

1. Generate a static pre-shared key used to secure the control and data channel.

```
openvpn --genkey secret static.key
```

2. Create a minimal server configuration referencing the static key, a virtual subnet, and the listening port.

```
# server.conf
dev tun
ifconfig 10.9.0.1 10.9.0.2
secret static.key
port 1194
proto udp
```

Note: static-key mode is shown here for simplicity; production OpenVPN deployments should use TLS mode with a proper PKI, as covered in Experiment 5.

b) Secure tunneling and traffic encryption

7.3 VPN Tunnel Overhead Calculator

Every VPN protocol adds header and encryption overhead to each packet it tunnels, which reduces the usable payload size (MSS) and matters when sizing MTUs for a link.

Write a function that accepts a payload size in bytes and a protocol name, and returns the total encapsulated packet size. Use these fixed per-packet overheads: "WireGuard" adds 60 bytes; "OpenVPN-UDP" adds 50

bytes; "IPsec-ESP" adds 73 bytes. Throw an IllegalArgumentException for an unrecognised protocol name.

Input: payloadSize = 1400, protocol = "WireGuard"

Output: 1460

Input: payloadSize = 1400, protocol = "OpenVPN-UDP"

Output: 1450

```
public class VpnOverheadCalculator
{
public static int encapsulatedSize(int payloadSize, String protocol)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

7.4 Verify Tunnel Traffic Is Encrypted with a Packet Capture

The final proof that a VPN tunnel is doing its job is that an observer on the outer network cannot read the inner traffic -- this is confirmed by capturing packets on the tunnel's underlying (outer) interface and checking that no plaintext is visible.

1. Start a capture on the network interface carrying the tunnel's encrypted outer packets while the VPN is active.

```
tcpdump -i eth0 -w tunnel_capture.pcap
```

2. Generate some traffic across the tunnel (e.g. ping the tunnel's internal peer address) so it appears in the capture.

3. Open the capture and confirm that the payload of the outer UDP/ESP packets is opaque ciphertext rather than the plaintext of the inner traffic.

```
tcpdump -r tunnel_capture.pcap -A | less
```

8. Authentication Protocols: Kerberos and Oauth

a) Implementing Kerberos authentication in a network

8.1 Obtain and Inspect a Kerberos Ticket

Kerberos authenticates a user once against a trusted Key Distribution Centre (KDC) and issues a ticket-granting ticket (TGT), which is then used to obtain service tickets without the user re-entering credentials for every service.

1. Request a TGT from the KDC for a configured principal (requires a working krb5.conf pointing at a KDC/realm, e.g. from a lab MIT Kerberos or a container-based test realm).

```
kinit alice@LAB.EXAMPLE
```

2. List the tickets currently cached for the session, noting the ticket's issue time and expiry time.

```
klist
```

3. Destroy the cached tickets once finished, to avoid leaving a live session on a shared machine.

```
kdestroy
```

Note: Kerberos requires a KDC and realm to be configured beforehand; if no test realm is available, read the klist output format from the documentation and proceed directly to exercise 8.2, which works from the same fields.

8.2 Kerberos Ticket Expiry Checker

A service that receives a Kerberos ticket must check that it has not expired before honouring it, exactly as klist reports a ticket's validity window to a user.

Write a function that accepts a ticket's issue time (epoch seconds), its lifetime in seconds, and the current time (epoch seconds), and returns "VALID" if the current time is within [issueTime, issueTime + lifetime], or "EXPIRED" otherwise.

Input: issueTime = 1000000, lifetime = 36000, currentTime = 1020000

Output: "VALID"

Input: issueTime = 1000000, lifetime = 36000, currentTime = 1050000

Output: "EXPIRED"

```
public class KerberosTicketChecker
{
public static String checkTicket(long issueTime, long lifetimeSeconds, long currentTime)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

b) <u>Configuring OAuth for secure API access</u>

8.3 Request an OAuth2 Access Token (Client Credentials Grant)

The client credentials grant is used for service-to-service API access, where a client authenticates directly with its own client ID and secret rather than on behalf of a user.

1. Request a token from the authorization server's token endpoint using a registered client ID and secret (use a local/test OAuth provider such as a sandboxed Keycloak or a mock server for this exercise).

```
curl -X POST https://auth.lab.local/oauth/token \
  -d grant_type=client_credentials \
  -d client_id=lab-client \
  -d client_secret=lab-secret \
  -d scope="read:data"
```

2. Inspect the JSON response and identify the access_token, token_type, expires_in, and scope fields.

3. Use the returned access token as a Bearer token when calling a protected API endpoint.

```
curl -H "Authorization: Bearer <access_token>" https://api.lab.local/data
```

8.4 OAuth Scope Authorization Checker

An API gateway does not just check that an OAuth access token is valid -- it also checks that the token's granted scopes actually cover the operation being requested, following the principle of least privilege.

Write a function that accepts an array of granted scope strings and a required scope string, and returns "AUTHORIZED" if the required scope is present in the granted scopes, or "DENIED" otherwise.

Input: grantedScopes = ["read:data", "write:data"], requiredScope = "write:data"

Output: "AUTHORIZED"

Input: grantedScopes = ["read:data"], requiredScope = "write:data"

Output: "DENIED"

```
public class OAuthScopeChecker
{
public static String checkAccess(String[] grantedScopes, String requiredScope)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

9: Secure Email Communication with PGP/GPG

a) Encryption and decryption of emails using GPG

9.1 Generate a GPG Key Pair

PGP-based email security relies on each user holding a personal key pair: the public key is shared with correspondents so they can encrypt messages to the user, and the private key is kept secret so only the user can decrypt them.

1. Generate a new key pair, choosing RSA and a key size of at least 3072 bits when prompted, and supplying a lab-only name and email address.

```
gpg --full-generate-key
```

2. List the public keys in the local keyring to confirm the new key was created.

```
gpg --list-keys
```

3. Export the public key in ASCII-armoured form so it could be shared with a correspondent.

```
gpg --armor --export lab-user@example.com > lab-user-public.asc
```

9.2 Encrypt and Decrypt a Message with GPG

Encrypting a message to a recipient's public key ensures that only the holder of the matching private key can read it, even if the ciphertext is intercepted in transit.

1. Encrypt a short text file to your own public key (standing in for a recipient), producing a .gpg ciphertext file.

```
echo 'This is a confidential lab message.' > message.txt
gpg --encrypt --recipient lab-user@example.com --output message.txt.gpg message.txt
```

2. Decrypt the ciphertext file back to plaintext using the matching private key, entering the key's passphrase when prompted.

```
gpg --decrypt --output message_decrypted.txt message.txt.gpg
```

3. Compare message.txt and message_decrypted.txt to confirm the round trip was lossless.

b) Verifying email authenticity using PGP signatures

9.3 Sign and Verify a Message with GPG

A digital signature over an email or attachment lets the recipient confirm both that the message came from the claimed sender and that it was not altered afterward -- the two properties PGP-signed email is designed to provide.

1. Create a detached signature over the message file using your private key.

```
gpg --detach-sign --armor --output message.txt.sig message.txt
```

2. Verify the signature against the original message, and note the "Good signature" (or failure) result GPG reports.

```
gpg --verify message.txt.sig message.txt
```

3. Modify one character in message.txt and re-run the verification to observe how GPG reports a signature that no longer matches.

9.4 PGP Signature Status Interpreter

Automated mail-processing pipelines parse GPG's machine-readable verification status rather than its human-readable text, so they need a mapping from GPG's status codes to a simple authenticity verdict.

Write a function that accepts a GPG verification status code string and returns a plain-language verdict, using this mapping: "GOODSIG" -> "TRUSTED"; "BADSIG" -> "INVALID_SIGNATURE"; "EXPKEYSIG" -> "SIGNED_WITH_EXPIRED_KEY"; "REVKEYSIG" -> "SIGNED_WITH_REVOKED_KEY". Return "UNKNOWN_STATUS" for any other code.

Input: "GOODSIG"

Output: "TRUSTED"

Input: "BADSIG"

Output: "INVALID_SIGNATURE"

Input: "NODATA"

Output: "UNKNOWN_STATUS"

```
public class PgpStatusInterpreter
{
public static String interpret(String statusCode)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

10. Wireless Security and WPA2 Encryption

Students are encouraged to bring their own laptops for laboratory practice sessions. All wireless, web-attack, intrusion-detection, and penetration-testing exercises must be performed only against equipment, access points, virtual machines, and services that the student owns or that the institution has explicitly set up and authorized for this course (e.g. an isolated lab access point, a local web server, or a deliberately vulnerable practice VM such as Metasploitable/DVWA). No exercise may be directed at any third-party, production, or public system.

a) Implementing WPA2 security for Wi-Fi networks

10.1 Configure WPA2-PSK on a Lab Access Point with hostapd

hostapd turns a Linux host with a suitable wireless adapter into an access point, which makes it a convenient way to configure and study WPA2 settings in a fully controlled lab environment.

1. Create a hostapd configuration enabling WPA2-Personal (WPA2-PSK) with AES/CCMP encryption on the lab interface.

```
# hostapd.conf
interface=wlan0
ssid=LabNetwork
wpa=2
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
wpa_passphrase=Lab-Only-Passphrase-2026
```

2. Start the access point using the configuration file.

```
hostapd hostapd.conf
```

3. From a separate client device, connect to LabNetwork and confirm in the client's network details that the connection reports WPA2-PSK (AES/CCMP), not WEP or an open network.

Note: use a lab-only SSID and passphrase, and only broadcast on hardware and a channel the institution has approved for this exercise.

10.2 WPA2 Passphrase Strength Validator

WPA2-Personal's security is only as strong as the pre-shared passphrase configured on the access point, since a short or common passphrase can be recovered from a captured handshake through offline brute-force. Before deploying an access point, an administrator should validate the passphrase against baseline rules.

Write a function that accepts a WPA2 passphrase and returns "INVALID_LENGTH" if it is shorter than 8 or longer than 63 characters (the protocol's hard limits). Otherwise, score one point each for: length >= 12, contains an uppercase letter, contains a digit, contains a special character (any of !@#$%^&*-_). Return "WEAK" for 0-1 points, "MODERATE" for 2-3 points, or "STRONG" for 4 points.

Input: "short"

Output: "INVALID_LENGTH"

Input: "homewifi"

Output: "WEAK"

Input: "Lab-Only-Passphrase-2026"

Output: "STRONG"

```
public class Wpa2PassphraseValidator
{
public static String validate(String passphrase)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

b) Analyzing wireless security threats and defenses

10.3 Capture a WPA2 Four-Way Handshake on the Lab Network

The WPA2 four-way handshake is the exchange that derives the session keys from the pre-shared passphrase, and capturing it (against your own lab AP only) is the standard way to study how offline passphrase-cracking tools operate and why passphrase strength matters.

1. Put the wireless adapter into monitor mode.

```
airmon-ng start wlan0
```

2. Capture frames on the lab access point's channel, writing them to a file.

```
airodump-ng -c <channel> --bssid <lab-AP-MAC> -w handshake wlan0mon
```

3. From an authorized lab client, reconnect to the access point (or trigger a reconnection) so the four-way handshake occurs while the capture is running.

4. Confirm airodump-ng reports "WPA handshake" captured for the lab AP's BSSID, then stop the capture.

Note: perform this only against the access point set up in exercise 10.1, and stop monitor mode when finished: airmon-ng stop wlan0mon.

10.4 Wireless Security Protocol Risk Classifier

A wireless security audit typically starts with an inventory of the encryption protocol each discovered access point advertises, since some legacy protocols are considered broken and should be flagged immediately.

Write a function that accepts a wireless security type string and returns its risk level, using this mapping: "OPEN" and "WEP" -> "CRITICAL"; "WPA-PSK" -> "HIGH"; "WPA2-PSK" -> "MODERATE"; "WPA3-SAE" -> "LOW". Return "UNKNOWN" for any other value.

Input: "WEP"

Output: "CRITICAL"

Input: "WPA2-PSK"

Output: "MODERATE"

Input: "WPA3-SAE"

Output: "LOW"

```
public class WirelessRiskClassifier
{
public static String classify(String securityType)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

11. Web Security and HTTPS Configuration

a) Configuring HTTPS with TLS on web servers

11.1 Configure HTTPS on Nginx with a TLS Certificate

Serving a web application over plain HTTP leaves every request and response readable and modifiable in transit, so enabling HTTPS with the certificate and key from Experiment 5 is one of the first steps in hardening a web server.

1. Add a server block that listens on port 443, references the certificate and key, and restricts the protocol to modern TLS versions.

```
server {
    listen 443 ssl;
    server_name lab.local;
    ssl_certificate     /etc/nginx/ssl/server.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;
    ssl_protocols TLSv1.2 TLSv1.3;
}
```

2. Reload Nginx to apply the configuration.

```
nginx -t
systemctl reload nginx
```

3. Confirm the site is reachable over HTTPS and inspect the served certificate.

```
curl -vI https://lab.local
```

11.2 Enforce an HTTP-to-HTTPS Redirect and Enable HSTS

Even after HTTPS is available, a site remains vulnerable if plain HTTP requests are still served, since a user's first request (or a stripped link) may still go out unencrypted. Redirecting HTTP to HTTPS and sending an HSTS header closes that gap.

1. Add a port-80 server block that redirects every request to the HTTPS equivalent.

```
server {
    listen 80;
    server_name lab.local;
    return 301 https://$host$request_uri;
}
```

2. Add the Strict-Transport-Security header to the HTTPS server block so compliant browsers refuse to fall back to HTTP for this site in the future.

```
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;
```

3. Reload Nginx and confirm a plain HTTP request now returns a 301 redirect, and the HTTPS response includes the HSTS header.

```
curl -I http://lab.local
curl -I https://lab.local
```

b) Preventing web-based attacks (XSS, CSRF, SQL Injection)

11.3 HTML Output Sanitizer (XSS Prevention)

A reflected or stored cross-site scripting (XSS) vulnerability arises when user-supplied text is written into an HTML page without escaping the characters that give it special meaning to the browser's parser.

Write a function that accepts a raw string and returns it with the following characters replaced: & -> &, < -> <, > -> >, " -> ", ' -> '. Replace & first, before any of the other substitutions, so escaped entities are not themselves re-escaped.

Input: "<script>alert(1)</script>"

Output: "<script>alert(1)</script>"

Input: "5 > 3 && 2 < 4"

Output: "5 > 3 && 2 < 4"

```
public class HtmlSanitizer
{
public static String sanitize(String input)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

11.4 SQL Injection Pattern Detector

As a defence-in-depth layer alongside parameterised queries, some web application firewalls flag request fields that contain classic SQL-injection metacharacters or keywords before the query is ever built, so the attempt can be logged and blocked.

Write a function that accepts a raw input string and returns "SUSPICIOUS" if it contains any of these (case-insensitive): "' OR '", "--", ";", "UNION SELECT", "DROP TABLE". Otherwise return "CLEAN".

Note: this pattern-matching check is a defence-in-depth aid only; the primary defence against SQL injection is always parameterised queries / prepared statements, not string filtering.

Input: "admin' OR '1'='1"

Output: "SUSPICIOUS"

Input: "Robert'); DROP TABLE Students;--"

Output: "SUSPICIOUS"

Input: "ordinary search term"

Output: "CLEAN"

```
public class SqlInjectionDetector
{
public static String scan(String input)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

12. Intrusion Detection and Prevention Systems (IDS/IPS)

a) Deploying Snort and Suricata for network intrusion detection

12.1 Run Snort in IDS Mode with a Custom Rule

Snort matches live or captured traffic against a rule set and raises an alert whenever a rule's conditions are met, making it one of the most widely deployed open-source network intrusion detection engines.

1. Add a simple custom rule to the local rules file that alerts on inbound traffic to TCP port 23 (Telnet), a protocol that should not appear on a hardened lab network.

```
alert tcp any any -> any 23 (msg:"Telnet traffic detected"; sid:1000001; rev:1;)
```

2. Run Snort in IDS mode against the lab interface (or a saved capture file) using this rule set.

```
snort -c /etc/snort/snort.conf -i eth0
# or, against a capture file:
snort -c /etc/snort/snort.conf -r sample.pcap
```

3. Generate a small amount of Telnet traffic on the lab network (or replay a capture containing it) and confirm the alert appears in Snort's console output or alert log.

12.2 Deploy Suricata and Analyse a Sample Capture

Suricata is a multi-threaded IDS/IPS engine that is largely rule-compatible with Snort but adds built-in protocol logging (HTTP, DNS, TLS, etc.), which makes its output useful for traffic analysis as well as alerting.

1. Run Suricata against a sample packet capture using a rule set and a log output directory.

```
suricata -r sample.pcap -c /etc/suricata/suricata.yaml -l ./suricata-logs
```

2. Inspect the generated alert log for any triggered signatures.

```
cat ./suricata-logs/fast.log
```

3. Inspect the structured event log and identify the protocols Suricata recognised in the capture.

```
jq '.event_type' ./suricata-logs/eve.json | sort | uniq -c
```

b) Traffic analysis and threat detection

12.3 Snort Rule Field Extractor

Analysts frequently need to programmatically inventory a large rule set -- for example, to list every rule's message and signature ID for a change-management report -- rather than reading each rule by hand.

Write a function that accepts a single Snort rule string and returns its msg and sid values as a two-element string array [msg, sid]. Assume the rule always contains msg:"..."; and sid:<number>; fields inside the parentheses.

Input: rule = "alert tcp any any -> any 23 (msg:\"Telnet traffic detected\"; sid:1000001; rev:1;)"

Output: ["Telnet traffic detected", "1000001"]

```
public class SnortRuleParser
{
public static String[] extractFields(String rule)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

12.4 Threshold-Based Port Scan Detector

A classic reconnaissance pattern is a single source host contacting many distinct destination ports in a short window; both Snort and Suricata include preprocessors for exactly this pattern, and re-implementing a simplified version builds intuition for how the detection threshold works.

Write a function that accepts an array of (port) integers representing the distinct destination ports one source IP contacted within a monitoring window, and a threshold integer, and returns "PORT_SCAN_DETECTED" if the number of distinct ports contacted is greater than or equal to the threshold, or "NORMAL" otherwise.

Input: ports = [22, 23, 80, 443, 3389, 8080], threshold = 5

Output: "PORT_SCAN_DETECTED"

Input: ports = [80, 443], threshold = 5

Output: "NORMAL"

```
import java.util.Set;
import java.util.HashSet;

public class PortScanDetector
{
public static String detect(int[] ports, int threshold)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

13. Penetration Testing and Ethical Hacking Techniques

a) Using Kali Linux tools for security testing

13.1 Network Discovery and Port Scanning with Nmap

The reconnaissance phase of an authorized penetration test starts with discovering live hosts and open ports on the target scope, which is exactly what a client engagement's rules of engagement would define before any testing begins.

1. Discover live hosts on the isolated lab subnet.

```
nmap -sn 192.168.56.0/24
```

2. Scan a specific lab target for open TCP ports.

```
nmap -p- 192.168.56.101
```

3. Re-scan the discovered open ports with service/version detection enabled.

```
nmap -sV -p <open-ports> 192.168.56.101
```

Note: run these scans only against hosts inside the isolated lab network (e.g. a local hypervisor's host-only network) that the institution has provided for this course.

13.2 Enumerate Known Vulnerabilities with Nmap Scripts

Once services and versions are known, the next standard step is checking them against known-vulnerability scripts, which is how a real assessment prioritises which findings to investigate further.

1. Run Nmap's vulnerability-detection script category against the lab target's discovered services.

```
nmap -sV --script vuln 192.168.56.101
```

2. Record each finding's script name, affected service, and any CVE identifiers reported, ready for the report drafted in exercise 13.4.

b) Exploiting and securing vulnerable network services

13.3 Exploit a Deliberately Vulnerable Service in the Lab VM with Metasploit

Purpose-built vulnerable virtual machines such as Metasploitable exist specifically so students can practise the exploitation phase of a penetration test end-to-end, safely and legally, before being trusted anywhere near a real engagement.

1. Start the Metasploit console.

```
msfconsole
```

2. Search for a module matching one of the vulnerable services identified on the lab VM in exercise 13.2.

```
search <service-name>
```

3. Select the module, set the target host, and review its required options before running it.

```
use <selected-module>
set RHOSTS 192.168.56.101
options
```

4. Run the module against the lab VM only, and record what access or information it provided.

```
run
```

5. Immediately afterward, identify the specific misconfiguration or missing patch that allowed the exploit to succeed, and note the remediation (e.g. patch version, disable the service, restrict network access).

Note: this exercise must be performed only against an intentionally vulnerable practice VM (e.g. Metasploitable) running on an isolated lab network segment with no route to production systems or the internet.

13.4 CVSS Score Severity Classifier

After a finding is confirmed, it is triaged by severity so remediation effort is prioritised correctly; the industry-standard way to do this is to map a CVSS base score to a qualitative severity band.

Write a function that accepts a CVSS v3 base score (a double from 0.0 to 10.0) and returns its severity band: "NONE" for 0.0, "LOW" for 0.1-3.9, "MEDIUM" for 4.0-6.9, "HIGH" for 7.0-8.9, or "CRITICAL" for 9.0-10.0.

Input: score = 9.8

Output: "CRITICAL"

Input: score = 5.3

Output: "MEDIUM"

Input: score = 0.0

Output: "NONE"

```
public class CvssSeverityClassifier
{
public static String classify(double score)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

14. Security Evaluation and Final Project Presentation

a) Analyzing security risks in a given system

14.1 Structured Risk Assessment of a Given System

Given a system description provided by the instructor (e.g. a small e-commerce platform with a web front end, an API, a database, and a payment gateway integration), carry out a structured risk assessment and produce a short written report containing:

- An asset inventory listing each component and the data it handles.
- A threat list for each asset, drawing on the topics covered in Experiments 1-13 (e.g. weak key management, missing TLS, injection flaws, unpatched services, weak Wi-Fi).
- A likelihood rating (1-5) and impact rating (1-5) for each threat, with a one-line justification for each rating.
- A resulting risk matrix (likelihood x impact) with each threat plotted, and the top three risks called out explicitly.

14.2 Risk Score Calculator

Turning likelihood and impact ratings into a single comparable number is what lets a risk matrix be sorted and prioritised automatically, rather than compared by eye.

Write a function that accepts a likelihood rating and an impact rating, both integers from 1 to 5, and returns their product as the risk score, together with a category: "LOW" for 1-6, "MEDIUM" for 7-14, "HIGH" for 15-20, or "CRITICAL" for 21-25. Return the result as a string in the form "<score> (<category>)".

Input: likelihood = 5, impact = 5

Output: "25 (CRITICAL)"

Input: likelihood = 2, impact = 3

Output: "6 (LOW)"

Input: likelihood = 4, impact = 4

Output: "16 (HIGH)"

```
public class RiskScoreCalculator
{
public static String calculate(int likelihood, int impact)
{
# Write code here
}

public static void main(String[] args)
{
# Write code here
}
}
```

b) Group projects on designing secure architectures

14.3 Final Group Project: Design a Secure Architecture

In groups, design a secure architecture for a scenario assigned by the instructor (e.g. a telemedicine platform, a mobile banking app, or an IoT-connected smart building system). The design must incorporate protocols and controls covered across the semester, and should be submitted as a report plus diagram containing:

- A high-level architecture diagram showing all major components, trust boundaries, and data flows.
- The cryptographic building blocks used and why (symmetric/asymmetric algorithms, hashing, key exchange -- Experiments 2-4).
- The PKI and TLS approach used to secure communications, including certificate issuance and validation (Experiments 5-6).
- The remote-access and network-segmentation approach, including any VPN usage (Experiment 7).
- The authentication and authorization approach for users and services (Experiment 8).
- The monitoring and detection controls proposed, including IDS/IPS placement (Experiment 12).
- A short threat model identifying the top risks the design mitigates and any residual risks that remain.

14.4 Final Presentation Requirements

Each group presents their design to the class and instructor. The presentation should be prepared as slides and should cover:

- A 2–3-minute overview of the scenario and the system being secured.
- A walkthrough of the architecture diagram and the key security decisions made, with justification for each.
- The top risks identified in the threat model and how the design addresses them.
- One design trade-off the group had to make (e.g. usability vs. security, cost vs. defence-in-depth) and how it was resolved.
- Time for questions from the class and instructor.

Note: presentations are typically 10-15 minutes per group, followed by a short Q&A; check with the instructor for the exact time allotted and grading rubric for this offering.

V. REFERENCE BOOKS:

1. William Stallings, **Cryptography and Network Security: Principles and Practice**, 9th Edition, Pearson, 2024.
2. Charlie Kaufman, Radia Perlman, Mike Speciner, **Network Security: Private Communication in a Public World**, 3rd Edition, Pearson, 2015.
3. **William Stallings,** Network Security Essentials: Applications and Standards**, 7th Edition, Pearson, 2023.**

VI. MATERIALS ONLINE

1. Course Outline Description
2. Lab Manul