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

COURSE CONTENT

DISTRIBUTED SYSTEMS AND SECURITY LABORATORY								
VII Semester: CSE(DS)								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
		L	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
ACCD30	Core	0	0	2	1	40	60	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisites: Nil.								

I. COURSE OVERVIEW:

This laboratory course provides hands-on experience in distributed systems and network security. Students learn to develop and test programs involving socket programming, remote method invocation, distributed algorithms, and synchronization. The course also covers practical cryptography, authentication mechanisms, network security protocols, and system security techniques. Through this lab, students gain exposure to real-world challenges in distributed computing and secure communications.

II. COURSE OBJECTIVES:

The students will try to learn:

I. Make use of distributed algorithms for mutual exclusion, synchronization, and leader election.
II. Analyze, detect, and prevent security threats in distributed systems.
III. Explore cryptographic methods to secure data and communications.

III. COURSE OUTCOMES:

CO1 Develop distributed file system applications with file sharing and access control
CO2 Apply cryptographic techniques for securing data in transit.
CO3 Implement client-server communication using TCP/UDP sockets and remote method invocation.
CO4 Apply distributed algorithms for coordination, synchronization, and mutual exclusion.
CO5 Implement network and system security protocols, including authentication, digital signatures, and SSL/TLS.
CO6 Analyze and simulate security threats and countermeasures in distributed systems

IV. COURSE CONTENT:

DISTRIBUTED SYSTEMS AND SECURITY LABORATORY

Note: All experiments are implemented in Python. Students are encouraged to run each program in a local Python 3 environment (using two terminals for client/server programs) and to extend the base solution using the 'Try' exercise at the end of each experiment.

Getting Started

The main emphasis of this course is on distributed computing and network/system security. Students will implement socket-based communication, remote invocation, classical distributed algorithms for coordination and mutual exclusion, symmetric and asymmetric cryptography, hashing and digital signatures, authentication and access-control mechanisms, common web/system attacks and their defenses, and finally a simple blockchain to tie together hashing, integrity, and consensus. Install Python 3.10+ and the packages pycryptodome, and optionally pyOpenSSL, before starting the lab.

1: Socket Programming

1.1 Client–Server Communication using TCP Sockets

Aim:

Write a program to implement Client–Server communication using TCP sockets.

Hint:

1. Create a TCP socket on the server using socket.AF_INET and socket.SOCK_STREAM.
2. Bind the server socket to an IP address and port, then call listen().
3. Accept an incoming connection using accept(), which returns a connection object.
4. Create a client-side TCP socket and connect() to the server's IP and port.
5. Exchange messages using send()/sendall() and recv(), then close both sockets.

Program:

```
# tcp_server.py
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('127.0.0.1', 5000))
server.listen(1)
print('TCP Server listening on port 5000...')

conn, addr = server.accept()
print('Connected by', addr)
data = conn.recv(1024).decode()
print('Received from client:', data)
conn.sendall(('Echo: ' + data).encode())
conn.close()
server.close()
```

```
# tcp_client.py
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('127.0.0.1', 5000))
client.sendall(b'Hello TCP Server')
reply = client.recv(1024).decode()
print('Server reply:', reply)
client.close()
```

Try: *Modify the program to allow the server to handle multiple messages from the client in a loop until the client sends '*exit*'.*

Sample Output:

```
TCP Server listening on port 5000...
Connected by ('127.0.0.1', 51322)
Received from client: Hello TCP Server
Server reply: Echo: Hello TCP Server
```

1.2 Client–Server Communication using UDP Sockets

Aim:

Write a program to implement Client–Server communication using UDP sockets.

Hint:

1. Create a UDP socket using socket.SOCK_DGRAM.
2. Bind the server socket to an IP address and port (no listen()/accept() needed).
3. Use recvfrom() at the server to receive a datagram along with the sender's address.
4. Use sendto() at the client to send a datagram directly to the server address.
5. Send the reply back using sendto() from the server to the client's address.

Program:

```
# udp_server.py
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(('127.0.0.1', 6000))
print('UDP Server ready on port 6000...')

data, addr = server.recvfrom(1024)
print('Received from', addr, ':', data.decode())
server.sendto(('Ack: ' + data.decode()).encode(), addr)
server.close()

# udp_client.py
import socket
```

```
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.sendto(b'Hello UDP Server', ('127.0.0.1', 6000))
reply, _ = client.recvfrom(1024)
print('Server reply:', reply.decode())
client.close()
```

Try: *Add a timeout using settimeout() on the client socket and handle socket.timeout when the server is unreachable.*

Sample Output:

```
UDP Server ready on port 6000...
Received from ('127.0.0.1', 54810) : Hello UDP Server
Server reply: Ack: Hello UDP Server
```

1.3 Chat Application using Socket Programming

Aim:

Develop a chat application using socket programming.

Hint:

1. Use TCP sockets and Python's threading module so send and receive can happen concurrently.
2. Start one thread that continuously listens for incoming messages using recv().
3. Use the main thread to read user input and send() it to the peer.
4. Agree on a simple 'quit' keyword to gracefully close the connection on both sides.
5. Run one instance in server mode and one in client mode to chat between two terminals.

Program:

```
# chat_app.py
import socket, threading, sys

def receive_messages(sock):
    while True:
        try:
            msg = sock.recv(1024).decode()
            if not msg or msg == 'quit':
                print('Peer disconnected.')
                break
            print('\nPeer:', msg)
        except OSError:
            break

def start_server(port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('0.0.0.0', port)); s.listen(1)
    print('Waiting for a peer to connect...')
    conn, addr = s.accept()
```

```
    print('Connected to', addr)
    return conn

def start_client(host, port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((host, port))
    return s

if __name__ == '__main__':
    mode = sys.argv[1]  # 'server' or 'client'
    conn = start_server(7000) if mode == 'server' else start_client('127.0.0.1', 7000)
    threading.Thread(target=receive_messages, args=(conn,), daemon=True).start()
    while True:
        text = input('You: ')
        conn.send(text.encode())
        if text == 'quit':
            break
    conn.close()
```

Try: *Extend the application to support broadcast group chat using a server that relays messages to all connected clients.*

Sample Output:

```
Waiting for a peer to connect...
Connected to ('127.0.0.1', 55210)
You: hello there
Peer: hi, how are you?
You: quit
```

2: Remote Method Invocation (RMI)

Note: Python has no native RMI. xmlrpc.server / xmlrpc.client provide the equivalent remote-procedure-call mechanism used to demonstrate RMI concepts below.

2.1 RMI-based Distributed Calculator

Aim:

Write a Python program to implement an RMI-based (XML-RPC) distributed calculator.

Hint:

1. Import SimpleXMLRPCServer and register calculator functions (add, sub, mul, div) as remote methods.
2. Start the server with register_function() and serve_forever() so it listens for remote calls.
3. On the client, create an xmlrpc.client.ServerProxy pointing to the server's URL.
4. Invoke the remote methods on the proxy object exactly like local function calls.
5. Handle division by zero on the server and let the exception propagate as a Fault to the client.

Program:

```
# calc_server.py
from xmlrpc.server import SimpleXMLRPCServer

def add(a, b): return a + b
def sub(a, b): return a - b
def mul(a, b): return a * b
def div(a, b): return a / b

server = SimpleXMLRPCServer(('localhost', 8001))
print('RMI Calculator server listening on port 8001...')
for f in (add, sub, mul, div):
    server.register_function(f, f.__name__)
server.serve_forever()

# calc_client.py
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:8001/')
print('add(10, 5) =', proxy.add(10, 5))
print('sub(10, 5) =', proxy.sub(10, 5))
print('mul(10, 5) =', proxy.mul(10, 5))
print('div(10, 5) =', proxy.div(10, 5))
```

Try: *Add a remote method for power(base, exp) and call it from the client.*

Sample Output:

```
RMI Calculator server listening on port 8001...
add(10, 5) = 15
sub(10, 5) = 5
mul(10, 5) = 50
div(10, 5) = 2.0
```

2.2 RMI Application for File Transfer

Aim:

Develop an RMI (XML-RPC) application for file transfer between client and server.

Hint:

1. On the server, expose a remote method that reads a file and returns its contents as Binary data.
2. Use xmlrpc.client.Binary to safely transport binary/text file data over XML-RPC.
3. On the client, call the remote method and write the returned bytes to a local file.
4. Add a remote upload_file() method that accepts Binary data and writes it on the server.
5. Test both download and upload directions between client and server.

Program:

```
# file_server.py
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def download_file(filename):
    with open(filename, 'rb') as f:
        return xmlrpc.client.Binary(f.read())

def upload_file(filename, data):
    with open(filename, 'wb') as f:
        f.write(data.data)
    return f'{filename} saved on server'

server = SimpleXMLRPCServer(('localhost', 8002))
print('RMI File-Transfer server listening on port 8002...')
server.register_function(download_file, 'download_file')
server.register_function(upload_file, 'upload_file')
server.serve_forever()

# file_client.py
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:8002/')
data = proxy.download_file('report.txt')
with open('report_copy.txt', 'wb') as f:
    f.write(data.data)
print('File downloaded as report_copy.txt')

with open('report_copy.txt', 'rb') as f:
    result = proxy.upload_file('report_uploaded.txt', xmlrpc.client.Binary(f.read()))
print(result)
```

Try: *Add progress reporting by splitting large files into chunks transferred over multiple RPC calls.*

Sample Output:

```
RMI File-Transfer server listening on port 8002...
File downloaded as report_copy.txt
report_uploaded.txt saved on server
```

2.3 Multi-Client RMI Server

Aim:

Implement a multi-client RMI (XML-RPC) server that can serve several clients concurrently.

Hint:

1. Subclass socketserver.ThreadingMixIn together with SimpleXMLRPCServer to allow one thread per client.

2. Register a shared remote object (e.g., a counter or key-value store) so state is visible to all clients.
3. Use a threading.Lock to protect shared state from race conditions between concurrent clients.
4. Launch several client processes/threads simultaneously to verify concurrent access works.
5. Log each client's requests with a timestamp to observe interleaved execution.

Program:

```
# multi_client_server.py
from xmlrpc.server import SimpleXMLRPCServer
from socketserver import ThreadingMixIn
import threading, datetime

class ThreadedXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
    daemon_threads = True

counter = 0
lock = threading.Lock()

def increment(client_id):
    global counter
    with lock:
        counter += 1
        print(f'[{datetime.datetime.now()}] client {client_id} -> counter={counter}')
        return counter

server = ThreadedXMLRPCServer(('localhost', 8003))
print('Multi-client RMI server listening on port 8003...')
server.register_function(increment, 'increment')
server.serve_forever()

# multi_client_client.py
import xmlrpc.client, sys

proxy = xmlrpc.client.ServerProxy('http://localhost:8003/')
client_id = sys.argv[1]
for _ in range(5):
    print('New counter value:', proxy.increment(client_id))
```

Try: *Run three client instances at the same time and verify the counter increments correctly without lost updates.*

Sample Output:

```
Multi-client RMI server listening on port 8003...
[2026-08-01 10:12:01] client A -> counter=1
[2026-08-01 10:12:01] client B -> counter=2
[2026-08-01 10:12:02] client A -> counter=3
```

3: Distributed Algorithms – I

3.1 Lamport's Logical Clock Algorithm

Aim:

Implement Lamport's Logical Clock algorithm to order events in a distributed system.

Hint:

1. Represent each process with its own logical clock counter, initialized to 0.
2. On an internal or send event, increment the process's own clock by 1.
3. On a receive event, set clock = max(local_clock, received_clock) + 1.
4. Simulate message exchange between processes using a list of events with timestamps.
5. Print the logical timestamp of every event to show a consistent causal ordering.

Program:

```
# lamport_clock.py
class Process:
    def __init__(self, name):
        self.name = name
        self.clock = 0

    def internal_event(self):
        self.clock += 1
        print(f'{self.name}: internal event, clock={self.clock}')

    def send_event(self, receiver):
        self.clock += 1
        print(f'{self.name}: send to {receiver.name}, clock={self.clock}')
        receiver.receive_event(self.clock)

    def receive_event(self, msg_clock):
        self.clock = max(self.clock, msg_clock) + 1
        print(f'{self.name}: receive event, clock={self.clock}')

if __name__ == '__main__':
    p1, p2, p3 = Process('P1'), Process('P2'), Process('P3')
    p1.internal_event()
    p1.send_event(p2)
    p2.internal_event()
    p2.send_event(p3)
    p3.internal_event()
```

Try: *Add a fourth process and simulate a scenario with two independent send events that later synchronize at a common receiver.*

Sample Output:

```
P1: internal event, clock=1
P1: send to P2, clock=2
P2: receive event, clock=3
```

```
P2: internal event, clock=4
P2: send to P3, clock=5
P3: receive event, clock=6
P3: internal event, clock=7
```

3.2 Vector Clock Algorithm for Causal Ordering

Aim:

Implement the Vector Clock algorithm to capture causal ordering of events among processes.

Hint:

1. Give each process a vector of size N (N = number of processes), initialized to all zeros.
2. On any local event, increment only the process's own index in its vector.
3. When sending a message, attach the current vector clock to the message.
4. On receive, take the element-wise maximum of the local and received vectors, then increment the receiver's own index.
5. Compare two vector clocks to determine causal precedence, concurrency, or equality.

Program:

```
# vector_clock.py
class VectorClockProcess:
    def __init__(self, pid, n):
        self.pid = pid
        self.clock = [0] * n

    def local_event(self):
        self.clock[self.pid] += 1
        print(f'P{self.pid} local event -> {self.clock}')

    def send_event(self, receiver):
        self.clock[self.pid] += 1
        print(f'P{self.pid} send -> {self.clock}')
        receiver.receive_event(list(self.clock))

    def receive_event(self, msg_clock):
        self.clock = [max(a, b) for a, b in zip(self.clock, msg_clock)]
        self.clock[self.pid] += 1
        print(f'P{self.pid} receive -> {self.clock}')

if __name__ == '__main__':
    n = 3
    procs = [VectorClockProcess(i, n) for i in range(n)]
    procs[0].local_event()
    procs[0].send_event(procs[1])
    procs[1].local_event()
    procs[1].send_event(procs[2])
    procs[2].local_event()
```

Try: *Write a function is_concurrent(v1, v2) that determines whether two vector-clock timestamps represent concurrent events.*

Sample Output:

```
P0 local event -> [1, 0, 0]
P0 send -> [2, 0, 0]
P1 receive -> [2, 1, 0]
P1 local event -> [2, 2, 0]
P1 send -> [2, 3, 0]
P2 receive -> [2, 3, 1]
P2 local event -> [2, 3, 2]
```

3.3 Bully Algorithm for Leader Election

Aim:

Implement the Bully algorithm to elect a coordinator (leader) among distributed processes.

Hint:

1. Assign each process a unique numeric ID; the process with the highest ID becomes coordinator.
2. When a process notices the coordinator has failed, it sends an ELECTION message to all processes with a higher ID.
3. If no higher process responds within a timeout, the process declares itself coordinator and broadcasts COORDINATOR.
4. If a higher process responds, it takes over the election and the original process waits for the result.
5. Simulate process failure by removing a process id from the active set before running the election.

Program:

```
# bully_algorithm.py
class BullyElection:
    def __init__(self, ids):
        self.active = sorted(ids)

    def elect(self, initiator):
        print(f'Process {initiator} starts election')
        higher = [p for p in self.active if p > initiator]
        if not higher:
            return self.announce(initiator)
        print(f'Process {initiator} sends ELECTION to {higher}')
        return self.elect(max(higher))

    def announce(self, leader):
        print(f'Process {leader} is the new COORDINATOR')
        for p in self.active:
            if p != leader:
                print(f'  -> COORDINATOR({leader}) sent to process {p}')
```

```
        return leader

if __name__ == '__main__':
    ids = [1, 2, 3, 4, 5, 6, 7]
    ids.remove(7)  # simulate coordinator (highest id) failure
    election = BullyElection(ids)
    leader = election.elect(initiator=2)
    print('Elected leader:', leader)
```

Try: *Simulate simultaneous election initiation by two different processes and verify both converge to the same leader.*

Sample Output:

```
Process 2 starts election
Process 2 sends ELECTION to [3, 4, 5, 6]
Process 6 starts election
Process 6 is the new COORDINATOR
  -> COORDINATOR(6) sent to process 1
  -> COORDINATOR(6) sent to process 2
Elected leader: 6
```

4: Distributed Algorithms – II

4.1 Ring Algorithm for Leader Election

Aim:

Implement the Ring algorithm for leader election in a logical ring of processes.

Hint:

1. Arrange all process IDs in a logical ring using a Python list (or collections.deque).
2. The initiating process sends an ELECTION message containing its own ID to its neighbour.
3. Each process that receives it appends its own ID (or keeps the max) and forwards it to the next neighbour.
4. When the message returns to the initiator, it picks the highest ID as the coordinator.
5. The initiator circulates a COORDINATOR message around the ring to inform every process.

Program:

```
# ring_algorithm.py
from collections import deque

def ring_election(ring, initiator):
    n = len(ring)
    start = ring.index(initiator)
    order = deque(ring)
    order.rotate(-start)
    collected = [initiator]
    print(f'Process {initiator} starts ELECTION message around the ring')
```

```
    for i in range(1, n):
        pid = order[i]
        collected.append(pid)
        print(f'  ELECTION message passes through process {pid}, collected={collected}')
    leader = max(collected)
    print(f'Process {initiator} determines leader = {leader}')
    for pid in ring:
        print(f'  COORDINATOR({leader}) delivered to process {pid}')
    return leader

if __name__ == '__main__':
    ring = [3, 1, 4, 5, 2]
    ring_election(ring, initiator=1)
```

Try: *Modify the algorithm so a process that has already failed is skipped when the ELECTION message is forwarded.*

Sample Output:

```
Process 1 starts ELECTION message around the ring
  ELECTION message passes through process 4, collected=[1, 4]
  ELECTION message passes through process 5, collected=[1, 4, 5]
  ELECTION message passes through process 2, collected=[1, 4, 5, 2]
  ELECTION message passes through process 3, collected=[1, 4, 5, 2, 3]
Process 1 determines leader = 5
  COORDINATOR(5) delivered to process 3
  COORDINATOR(5) delivered to process 1
```

4.2 Ricart–Agrawala Algorithm for Distributed Mutual Exclusion

Aim:

Implement the Ricart–Agrawala algorithm for achieving mutual exclusion in a distributed system.

Hint:

1. Maintain a logical (Lamport) clock and a request timestamp for every process wanting the critical section.
2. To enter the critical section, a process broadcasts a REQUEST(timestamp, pid) to all other processes.
3. A process replies immediately unless it is in the critical section, or it also wants it with a smaller timestamp — in which case it defers the reply.
4. A process enters the critical section only after receiving replies from all other processes.
5. On exiting the critical section, the process sends any deferred replies it was holding.

Program:

```
# ricart_agrawala.py
import heapq

class RAProcess:
    def __init__(self, pid, n):
        self.pid = pid; self.n = n
```

```
        self.clock = 0
        self.replies_needed = 0
        self.deferred = []
        self.requesting = False
        self.request_ts = None

    def request_cs(self, others):
        self.clock += 1
        self.request_ts = (self.clock, self.pid)
        self.requesting = True
        self.replies_needed = len(others)
        print(f'P{self.pid} REQUEST ts={self.request_ts}')
        for o in others:
            if o.on_request(self.request_ts, self):
                self.replies_needed -= 1

    def on_request(self, ts, requester):
        defer = self.requesting and (self.request_ts < ts)
        if defer:
            self.deferred.append(requester)
            print(f'P{self.pid} defers reply to P{requester.pid}')
            return False
        print(f'P{self.pid} replies immediately to P{requester.pid}')
        return True

    def enter_cs(self):
        print(f'P{self.pid} ENTERS critical section' if self.replies_needed == 0 else 'waiting...')

    def exit_cs(self, all_procs):
        self.requesting = False
        print(f'P{self.pid} EXITS critical section')
        for p in self.deferred:
            print(f'P{self.pid} sends deferred reply to P{p.pid}')
        self.deferred = []

if __name__ == '__main__':
    p = [RAProcess(i, 3) for i in range(3)]
    p[0].request_cs([p[1], p[2]])
    p[0].enter_cs()
    p[0].exit_cs(p)
```

Try: *Extend the simulation to two processes requesting the critical section simultaneously and show how timestamps break the tie.*

Sample Output:

```
P0 REQUEST ts=(1, 0)
P1 replies immediately to P0
P2 replies immediately to P0
P0 ENTERS critical section
P0 EXITS critical section
```

WEEK 5: Process Coordination & Synchronization

5.1 Token Ring Algorithm for Mutual Exclusion

Aim:

Implement the Token Ring algorithm for achieving mutual exclusion among distributed processes.

Hint:

1. Arrange processes in a logical ring; only the process holding the token may enter the critical section.
2. Circulate a single token continuously around the ring using a simple list rotation or a queue.
3. A process wanting the critical section waits until the token arrives, then uses it and passes it on.
4. A process that does not need the critical section simply forwards the token immediately.
5. Ensure exactly one token exists in the system at any time to guarantee mutual exclusion.

Program:

```
# token_ring.py
import time

def token_ring(processes, wants_cs, rounds=1):
    n = len(processes)
    holder = 0
    for _ in range(rounds * n):
        pid = processes[holder]
        if pid in wants_cs:
            print(f'Process {pid} holds token -> ENTER critical section')
            time.sleep(0.01)
            print(f'Process {pid} EXITS critical section, passes token')
        else:
            print(f'Process {pid} holds token, does not need CS, passes it on')
        holder = (holder + 1) % n

if __name__ == '__main__':
    processes = [1, 2, 3, 4]
    wants_cs = {2, 4}
    token_ring(processes, wants_cs)
```

Try: *Modify the simulation so a process can dynamically request the critical section while the token is circulating (using a shared request queue).*

Sample Output:

```
Process 1 holds token, does not need CS, passes it on
Process 2 holds token -> ENTER critical section
Process 2 EXITS critical section, passes token
Process 3 holds token, does not need CS, passes it on
Process 4 holds token -> ENTER critical section
Process 4 EXITS critical section, passes token
```

5.2 Distributed Semaphore using Message Passing

Aim:

Implement a Distributed Semaphore using message passing between processes.

Hint:

1. Implement a semaphore manager process that maintains a counter and a FIFO queue of waiting requests.
2. Processes send an ACQUIRE message; the manager grants it immediately if counter > 0, else queues the request.
3. Processes send a RELEASE message; the manager increments the counter and grants access to the next queued request if any.
4. Use Python's multiprocessing.Queue or a simple in-process message queue to simulate the message passing.
5. Test the semaphore with more requesting processes than available permits to verify correct queuing.

Program:

```
# distributed_semaphore.py
from collections import deque

class SemaphoreManager:
    def __init__(self, permits):
        self.count = permits
        self.queue = deque()

    def acquire(self, pid):
        if self.count > 0:
            self.count -= 1
            print(f'ACQUIRE granted to P{pid} immediately (remaining={self.count})')
        else:
            self.queue.append(pid)
            print(f'P{pid} queued, waiting for a permit')

    def release(self, pid):
        print(f'RELEASE received from P{pid}')
        if self.queue:
            nxt = self.queue.popleft()
            print(f'ACQUIRE granted to P{nxt} from queue')
        else:
            self.count += 1
            print(f'No waiters, permit count now {self.count}')

if __name__ == '__main__':
    mgr = SemaphoreManager(permits=2)
    for pid in [1, 2, 3, 4]:
        mgr.acquire(pid)
```

```
    mgr.release(1)
    mgr.release(2)
```

Try: *Extend the manager to support weighted semaphores where each process can request more than one permit at a time.*

Sample Output:

```
ACQUIRE granted to P1 immediately (remaining=1)
ACQUIRE granted to P2 immediately (remaining=0)
P3 queued, waiting for a permit
P4 queued, waiting for a permit
RELEASE received from P1
ACQUIRE granted to P3 from queue
RELEASE received from P2
ACQUIRE granted to P4 from queue
```

5.3 Deadlock Detection in a Distributed Environment

Aim:

Demonstrate deadlock detection in a distributed environment using a Wait-For Graph (WFG).

Hint:

1. Model resource dependencies as a directed graph where an edge Pi -> Pj means Pi is waiting for a resource held by Pj.
2. Represent the wait-for graph using a Python dictionary of adjacency lists.
3. Detect a deadlock by finding a cycle in the graph using depth-first search (DFS).
4. Track the recursion stack during DFS to correctly identify back-edges that indicate cycles.
5. Report the exact cycle of processes involved when a deadlock is detected.

Program:

```
# deadlock_detection.py
def has_cycle(graph):
    visited, stack, path = set(), set(), []

    def dfs(node):
        visited.add(node); stack.add(node); path.append(node)
        for neighbour in graph.get(node, []):
            if neighbour not in visited:
                if dfs(neighbour):
                    return True
            elif neighbour in stack:
                cycle = path[path.index(neighbour):] + [neighbour]
                print('Deadlock cycle detected:', ' -> '.join(cycle))
                return True
        stack.remove(node); path.pop()
        return False

    for node in list(graph):
```

```
        if node not in visited:
            if dfs(node):
                return True
    return False

if __name__ == '__main__':
    wait_for_graph = {
        'P1': ['P2'],
        'P2': ['P3'],
        'P3': ['P1'],
        'P4': ['P2'],
    }
    if has_cycle(wait_for_graph):
        print('Deadlock DETECTED in the system')
    else:
        print('No deadlock found')
```

Try: *Remove the edge P3 -> P1 and rerun the program to verify the system correctly reports no deadlock.*

Sample Output:

```
Deadlock cycle detected: P1 -> P2 -> P3 -> P1
Deadlock DETECTED in the system
```

WEEK 6: Distributed File Systems

6.1 Simulate a Distributed File Sharing System

Aim:

Write a program to simulate a Distributed File Sharing System.

Hint:

1. Maintain a central directory (dictionary) that maps file names to the node currently holding each file.
2. Implement register_file(node, filename) so a node can announce it is hosting a file.
3. Implement locate_file(filename) so any node can query which peer currently holds a file.
4. Implement a simple access-control check so only authorized nodes can request certain files.
5. Simulate multiple nodes registering and requesting files from the shared directory.

Program:

```
# dfs_simulation.py
class DirectoryServer:
    def __init__(self):
        self.registry = {}
        self.permissions = {}

    def register_file(self, node, filename, allowed=None):
```

```
        self.registry[filename] = node
        self.permissions[filename] = allowed or set()
        print(f'{node} registered file "{filename}"')

    def locate_file(self, requester, filename):
        if filename not in self.registry:
            print(f'{requester}: file "{filename}" not found')
            return None
        owner = self.registry[filename]
        allowed = self.permissions[filename]
        if allowed and requester not in allowed and requester != owner:
            print(f'{requester}: ACCESS DENIED for "{filename}"')
            return None
        print(f'{requester}: file "{filename}" is hosted on {owner}')
        return owner

if __name__ == '__main__':
    dirserver = DirectoryServer()
    dirserver.register_file('NodeA', 'notes.txt', allowed={'NodeB'})
    dirserver.register_file('NodeC', 'public.txt')
    dirserver.locate_file('NodeB', 'notes.txt')
    dirserver.locate_file('NodeD', 'notes.txt')
    dirserver.locate_file('NodeD', 'public.txt')
```

Try: *Add a replicate_file() method so a file can be hosted on more than one node, and locate_file() returns the closest/least-loaded replica.*

Sample Output:

```
NodeA registered file "notes.txt"
NodeC registered file "public.txt"
NodeB: file "notes.txt" is hosted on NodeA
NodeD: ACCESS DENIED for "notes.txt"
NodeD: file "public.txt" is hosted on NodeC
```

6.2 Client–Server File Read/Write Operations

Aim:

Write a program to implement client-server file read/write operations over sockets.

Hint:

1. Design a simple text protocol: commands like 'READ filename', 'WRITE filename', followed by the file content.
2. On the server, parse the command and either send back file contents or write received data to disk.
3. Use recv() in a loop to make sure the full file content is received before saving it.
4. On the client, send the command and either display returned data or transmit the data to be written.
5. Handle the case where the requested file does not exist on the server.

Program:

```
# file_server.py
import socket, os

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('127.0.0.1', 9000)); server.listen(1)
print('File server listening on port 9000...')
conn, addr = server.accept()
cmd, fname = conn.recv(1024).decode().split(' ', 1)

if cmd == 'READ':
    if os.path.exists(fname):
        with open(fname) as f:
            conn.sendall(f.read().encode())
    else:
        conn.sendall(b'ERROR: file not found')
elif cmd == 'WRITE':
    data = conn.recv(4096).decode()
    with open(fname, 'w') as f:
        f.write(data)
    conn.sendall(b'WRITE OK')
conn.close(); server.close()

# file_client.py
import socket

def read_file(fname):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('127.0.0.1', 9000))
    s.sendall(f'READ {fname}'.encode())
    print(s.recv(4096).decode())
    s.close()

def write_file(fname, content):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('127.0.0.1', 9000))
    s.sendall(f'WRITE {fname}'.encode())
    s.sendall(content.encode())
    print(s.recv(4096).decode())
    s.close()
```

Try: *Add file locking so a WRITE from one client is rejected while another client currently holds a lock on the same file.*

Sample Output:

```
File server listening on port 9000...
WRITE OK
Hello from the distributed file system lab!
```

7: Message Passing Systems

7.1 Producer–Consumer Problem using Message Passing

Aim:

Implement the Producer–Consumer problem using message passing (queue-based IPC).

Hint:

1. Use a bounded queue.Queue as the shared message channel between producer and consumer threads/processes.
2. The producer put()s items into the queue; if the queue is full it blocks automatically, avoiding overflow.
3. The consumer get()s items from the queue; if empty it blocks until a new item is produced.
4. Use a sentinel value (e.g., None) to signal the consumer to stop once production is finished.
5. Run producer and consumer as separate threads to demonstrate real concurrent message passing.

Program:

```
# producer_consumer.py
import queue, threading, time, random

buffer = queue.Queue(maxsize=5)

def producer():
    for i in range(10):
        item = f'item-{i}'
        buffer.put(item)
        print('Produced:', item)
        time.sleep(random.uniform(0.01, 0.05))
    buffer.put(None)  # sentinel

def consumer():
    while True:
        item = buffer.get()
        if item is None:
            break
        print('Consumed:', item)
        time.sleep(random.uniform(0.01, 0.05))

if __name__ == '__main__':
    t1 = threading.Thread(target=producer)
    t2 = threading.Thread(target=consumer)
    t1.start(); t2.start()
    t1.join(); t2.join()
    print('Producer-Consumer finished')
```

Try: *Extend the program to use multiple producer and consumer threads sharing the same bounded queue.*

Sample Output:

```
Produced: item-0
Consumed: item-0
Produced: item-1
Consumed: item-1
...
Producer-Consumer finished
```

7.2 Publisher–Subscriber Model

Aim:

Implement the Publisher–Subscriber (Pub/Sub) messaging model.

Hint:

1. Design a broker/topic manager class that maintains a mapping of topic -> list of subscriber callbacks.
2. Implement subscribe(topic, callback) to register a subscriber's interest in a topic.
3. Implement publish(topic, message) to invoke every subscriber callback registered for that topic.
4. Allow subscribers to unsubscribe() from a topic when no longer interested.
5. Demonstrate multiple subscribers receiving the same published message independently.

Program:

```
# pub_sub.py
class Broker:
    def __init__(self):
        self.topics = {}

    def subscribe(self, topic, subscriber, callback):
        self.topics.setdefault(topic, []).append((subscriber, callback))
        print(f'{subscriber} subscribed to "{topic}"')

    def unsubscribe(self, topic, subscriber):
        self.topics[topic] = [s for s in self.topics.get(topic, []) if s[0] != subscriber]
        print(f'{subscriber} unsubscribed from "{topic}"')

    def publish(self, topic, message):
        print(f'Publishing to "{topic}": {message}')
        for subscriber, callback in self.topics.get(topic, []):
            callback(subscriber, topic, message)

def handler(subscriber, topic, message):
    print(f'  {subscriber} received on "{topic}": {message}')

if __name__ == '__main__':
    broker = Broker()
```

```
    broker.subscribe('weather', 'Sub-A', handler)
    broker.subscribe('weather', 'Sub-B', handler)
    broker.subscribe('sports', 'Sub-C', handler)
    broker.publish('weather', 'Rain expected tomorrow')
    broker.unsubscribe('weather', 'Sub-A')
    broker.publish('weather', 'Clear skies now')
```

Try: *Add topic wildcards (e.g., 'weather.*') so a subscriber can receive messages from multiple related sub-topics.*

Sample Output:

```
Sub-A subscribed to "weather"
Sub-B subscribed to "weather"
Sub-C subscribed to "sports"
Publishing to "weather": Rain expected tomorrow
  Sub-A received on "weather": Rain expected tomorrow
  Sub-B received on "weather": Rain expected tomorrow
Sub-A unsubscribed from "weather"
Publishing to "weather": Clear skies now
  Sub-B received on "weather": Clear skies now
```

8: Cryptography – Symmetric Key

8.1 DES Encryption and Decryption

Aim:

Write a program to implement DES encryption and decryption.

Hint:

1. Install pycryptodome (pip install pycryptodome) which provides the Crypto.Cipher.DES module.
2. DES requires an 8-byte key and operates on 8-byte blocks, so use padding (e.g., Crypto.Util.Padding.pad).
3. Create the cipher object with DES.new(key, DES.MODE_ECB) or MODE_CBC with an IV.
4. Encrypt with cipher.encrypt(padded_plaintext) and decrypt with a fresh cipher object using cipher.decrypt().
5. Unpad the decrypted bytes to recover the original plaintext.

Program:

```
# des_cipher.py
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad

key = b'8bytekey'          # exactly 8 bytes
plaintext = b'Hello Distributed Security Lab!'

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

```
ciphertext = cipher.encrypt(pad(plaintext, DES.block_size))
print('Ciphertext (hex):', ciphertext.hex())

decipher = DES.new(key, DES.MODE_ECB)
decrypted = unpad(decipher.decrypt(ciphertext), DES.block_size)
print('Decrypted text:', decrypted.decode())
```

Try: *Switch the mode to DES.MODE_CBC with a random 8-byte IV and observe how the ciphertext changes for identical plaintext blocks.*

Sample Output:

```
Ciphertext (hex): 7c3f9a1e6b0d2c4e5a8f1d9b3c7e0a6f...
Decrypted text: Hello Distributed Security Lab!
```

8.2 AES Encryption and Decryption

Aim:

Implement AES encryption and decryption.

Hint:

1. Use Crypto.Cipher.AES from pycryptodome with a 16/24/32-byte key (AES-128/192/256).
2. Use AES.MODE_CBC or AES.MODE_GCM for a secure mode; generate a random IV/nonce for each encryption.
3. Pad the plaintext to a multiple of AES.block_size (16 bytes) before encrypting in CBC mode.
4. Prepend or store the IV alongside the ciphertext so the receiver can decrypt correctly.
5. Decrypt using the same key and IV, then unpad to recover the original plaintext.

Program:

```
# aes_cipher.py
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

key = get_random_bytes(16)          # AES-128 key
iv = get_random_bytes(16)
plaintext = b'Secure communication using AES'

cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(plaintext, AES.block_size))
print('Ciphertext (hex):', ciphertext.hex())

decipher = AES.new(key, AES.MODE_CBC, iv)
decrypted = unpad(decipher.decrypt(ciphertext), AES.block_size)
print('Decrypted text:', decrypted.decode())
```

Try: *Re-implement the program using AES.MODE_GCM to additionally get a message authentication tag, and verify tampering is detected.*

Sample Output:

```
Ciphertext (hex): 4e91c8a2f0b7... (varies each run)
Decrypted text: Secure communication using AES
```

8.3 Caesar Cipher and Playfair Cipher

Aim:

Implement the Caesar Cipher and Playfair Cipher classical encryption techniques.

Hint:

1. For Caesar cipher, shift each alphabetic character by a fixed key using modular arithmetic on ord() values.
2. Preserve case and leave non-alphabetic characters unchanged during Caesar encryption.
3. For Playfair cipher, first build a 5x5 key matrix from a keyword (removing duplicate letters, merging I/J).
4. Split the plaintext into digraphs, inserting a filler letter ('X') for repeated letters or odd length.
5. Apply the Playfair row/column/rectangle rules to encrypt each digraph.

Program:

```
# caesar_playfair.py
def caesar_encrypt(text, key):
    result = []
    for ch in text:
        if ch.isalpha():
            base = ord('A') if ch.isupper() else ord('a')
            result.append(chr((ord(ch) - base + key) % 26 + base))
        else:
            result.append(ch)
    return ''.join(result)

def build_playfair_matrix(keyword):
    keyword = keyword.upper().replace('J', 'I')
    seen, matrix = set(), []
    for ch in keyword + 'ABCDEFGHIKLMNOPQRSTUVWXYZ':
        if ch not in seen and ch.isalpha():
            seen.add(ch); matrix.append(ch)
    return [matrix[i*5:(i+1)*5] for i in range(5)]

def find_pos(matrix, ch):
    for r, row in enumerate(matrix):
        if ch in row:
            return r, row.index(ch)

def playfair_encrypt(text, keyword):
```

```
    matrix = build_playfair_matrix(keyword)
    text = text.upper().replace('J', 'I').replace(' ', '')
    pairs, i = [], 0
    while i < len(text):
        a = text[i]
        b = text[i+1] if i+1 < len(text) else 'X'
        if a == b:
            b = 'X'; i += 1
        else:
            i += 2
        pairs.append((a, b))
    cipher = ''
    for a, b in pairs:
        r1, c1 = find_pos(matrix, a); r2, c2 = find_pos(matrix, b)
        if r1 == r2:
            cipher += matrix[r1][(c1+1)%5] + matrix[r2][(c2+1)%5]
        elif c1 == c2:
            cipher += matrix[(r1+1)%5][c1] + matrix[(r2+1)%5][c2]
        else:
            cipher += matrix[r1][c2] + matrix[r2][c1]
    return cipher

if __name__ == '__main__':
    print('Caesar:', caesar_encrypt('Attack At Dawn', 3))
    print('Playfair:', playfair_encrypt('HELLO WORLD', 'KEYWORD'))
```

Try: *Write the corresponding caesar_decrypt() and playfair_decrypt() functions to recover the original plaintext.*

Sample Output:

```
Caesar: Dwwdfn Dw Gdzq
Playfair: GYIZSDCPDZKG
```

9: Cryptography – Asymmetric Key

9.1 RSA Encryption and Decryption

Aim:

Implement RSA encryption and decryption.

Hint:

1. Choose two prime numbers p and q, then compute n = p*q and phi = (p-1)*(q-1).
2. Pick a public exponent e coprime with phi (commonly 65537), and compute the private exponent d as the modular inverse of e mod phi.
3. Encrypt an integer message m as c = m^e mod n using Python's built-in pow(m, e, n).
4. Decrypt using m = c^d mod n with pow(c, d, n).

5. For real text, convert characters to integers below n, or use pycryptodome's Crypto.PublicKey.RSA for production-grade key sizes.

Program:

```
# rsa_cipher.py
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

key = RSA.generate(2048)
public_key = key.publickey()

message = b'Distributed Systems and Security Lab'
cipher_enc = PKCS1_OAEP.new(public_key)
ciphertext = cipher_enc.encrypt(message)
print('Ciphertext (hex):', ciphertext.hex()[:64], '...')

cipher_dec = PKCS1_OAEP.new(key)
plaintext = cipher_dec.decrypt(ciphertext)
print('Decrypted:', plaintext.decode())
```

Try: *Implement RSA from first principles (small primes p, q) to encrypt and decrypt a single small integer, without using pycryptodome.*

Sample Output:

```
Ciphertext (hex): 3a91f0c8... ...
Decrypted: Distributed Systems and Security Lab
```

9.2 Diffie–Hellman Key Exchange Algorithm

Aim:

Implement the Diffie–Hellman key exchange algorithm.

Hint:

1. Agree publicly on a large prime p and a primitive root (generator) g.
2. Each party picks a private random number and computes its public value as g^private mod p.
3. Exchange the public values over the (insecure) channel.
4. Each party computes the shared secret as other_public^own_private mod p.
5. Verify both parties independently arrive at the same shared secret key.

Program:

```
# diffie_hellman.py
import random

p = 23   # public prime
g = 5    # public primitive root
```

```
a_private = random.randint(1, p - 2)   # Alice's private key
b_private = random.randint(1, p - 2)   # Bob's private key

a_public = pow(g, a_private, p)
b_public = pow(g, b_private, p)
print('Alice public:', a_public, ' Bob public:', b_public)

a_shared = pow(b_public, a_private, p)
b_shared = pow(a_public, b_private, p)

print('Alice computed shared secret:', a_shared)
print('Bob computed shared secret  :', b_shared)
print('Keys match:', a_shared == b_shared)
```

Try: *Replace the toy prime p=23 with a cryptographically strong 2048-bit prime from RFC 3526 and repeat the exchange.*

Sample Output:

```
Alice public: 8  Bob public: 19
Alice computed shared secret: 2
Bob computed shared secret  : 2
Keys match: True
```

9.3 Elliptic Curve Cryptography (ECC) – Basic Simulation

Aim:

Implement a basic simulation of Elliptic Curve Cryptography (ECC).

Hint:

1. Define a small elliptic curve y^2 = x^3 + ax + b (mod p) and a base point G on the curve.
2. Implement point addition and point doubling using the standard elliptic-curve group formulas over a finite field.
3. Implement scalar multiplication k*G via repeated point addition (double-and-add algorithm).
4. Each party chooses a private scalar and computes its public key as private*G.
5. Perform ECDH: each party computes the shared secret as own_private * other_public and confirms both match.

Program:

```
# ecc_simulation.py
p, a, b = 97, 2, 3          # small curve for demonstration: y^2 = x^3 + 2x + 3 (mod 97)
G = (3, 6)                  # base point on the curve

def inverse_mod(k, p):
    return pow(k, p - 2, p)
```

```
def point_add(P, Q):
    if P is None: return Q
    if Q is None: return P
    if P == Q:
        lam = (3 * P[0]**2 + a) * inverse_mod(2 * P[1], p) % p
    else:
        if P[0] == Q[0]:
            return None
        lam = (Q[1] - P[1]) * inverse_mod(Q[0] - P[0], p) % p
    x = (lam**2 - P[0] - Q[0]) % p
    y = (lam * (P[0] - x) - P[1]) % p
    return (x, y)

def scalar_mult(k, P):
    result, addend = None, P
    while k:
        if k & 1:
            result = point_add(result, addend)
        addend = point_add(addend, addend)
        k >>= 1
    return result

if __name__ == '__main__':
    alice_priv, bob_priv = 15, 23
    alice_pub = scalar_mult(alice_priv, G)
    bob_pub = scalar_mult(bob_priv, G)
    print('Alice public key:', alice_pub)
    print('Bob public key  :', bob_pub)

    shared_alice = scalar_mult(alice_priv, bob_pub)
    shared_bob = scalar_mult(bob_priv, alice_pub)
    print('Shared secret (Alice):', shared_alice)
    print('Shared secret (Bob)  :', shared_bob)
```

Try: *Replace the toy curve with the standard secp256k1 parameters (using the 'ecdsa' or 'tinyec' Python library) for realistic key sizes.*

Sample Output:

```
Alice public key: (63, 55)
Bob public key  : (43, 63)
Shared secret (Alice): (28, 17)
Shared secret (Bob)  : (28, 17)
```

10: Hashing and Digital Signatures

10.1 Message Authentication Code (MAC)

Aim:

Implement a Message Authentication Code (MAC) using HMAC to verify data integrity and authenticity.

Hint:

1. Use Python's built-in hmac module together with hashlib (e.g., hashlib.sha256) to build the MAC function.
2. Both sender and receiver must share the same secret key beforehand.
3. The sender computes hmac.new(key, message, hashlib.sha256).hexdigest() and transmits it with the message.
4. The receiver recomputes the HMAC on the received message using the same key.
5. Use hmac.compare_digest() for a constant-time comparison to avoid timing attacks, and detect tampering.

Program:

```
# hmac_mac.py
import hmac, hashlib

secret_key = b'shared-secret-key'

def generate_mac(message: bytes) -> str:
    return hmac.new(secret_key, message, hashlib.sha256).hexdigest()

def verify_mac(message: bytes, mac: str) -> bool:
    expected = generate_mac(message)
    return hmac.compare_digest(expected, mac)

if __name__ == '__main__':
    message = b'Transfer $500 to account 12345'
    mac = generate_mac(message)
    print('Message:', message.decode())
    print('Generated MAC:', mac)

    tampered = b'Transfer $5000 to account 12345'
    print('Original message valid? ', verify_mac(message, mac))
    print('Tampered message valid? ', verify_mac(tampered, mac))
```

Try: *Recompute the MAC after changing a single character in the message and observe how completely the digest changes (avalanche effect).*

Sample Output:

```
Message: Transfer $500 to account 12345
Generated MAC: 8f21ac0e6b93d1... (64 hex chars)
Original message valid?  True
Tampered message valid?  False
```

10.2 Man-in-the-Middle Attack Simulation

Aim:

Demonstrate a Man-in-the-Middle (MITM) attack simulation on an unauthenticated Diffie–Hellman key exchange.

Hint:

1. Simulate three parties: Alice, Bob, and an Attacker (Mallory) sitting between them.
2. Have Mallory intercept Alice's public DH value and substitute her own public value before forwarding it to Bob, and vice versa.
3. Show that Mallory ends up sharing one key with Alice and a different key with Bob, letting her decrypt/re-encrypt all traffic.
4. Contrast this with an authenticated exchange where digital signatures on the public values would let Bob detect the substitution.
5. Print the three parties' derived keys to clearly illustrate the mismatch that MITM produces.

Program:

```
# mitm_simulation.py
import random

p, g = 23, 5

def keypair():
    priv = random.randint(2, p - 2)
    return priv, pow(g, priv, p)

alice_priv, alice_pub = keypair()
bob_priv, bob_pub = keypair()
mallory_priv, mallory_pub = keypair()

print('Without attacker -> Alice and Bob would share the same key.')
print()
print('--- MITM attack in progress ---')
print('Mallory intercepts and substitutes her own public value for both sides.')

alice_shared_with_mallory = pow(mallory_pub, alice_priv, p)   # Alice thinks this is Bob's key
mallory_shared_with_alice = pow(alice_pub, mallory_priv, p)

bob_shared_with_mallory = pow(mallory_pub, bob_priv, p)       # Bob thinks this is Alice's key
```

```
mallory_shared_with_bob = pow(bob_pub, mallory_priv, p)

print('Alice <-> Mallory shared key:', alice_shared_with_mallory, mallory_shared_with_alice)
print('Mallory <-> Bob shared key  :', mallory_shared_with_bob, bob_shared_with_mallory)
print('Alice-Bob direct key would differ, proving Mallory can decrypt and re-encrypt all traffic.')
```

Try: *Add digital signatures (e.g., RSA-signed public values) to the exchange and show how Bob can now detect Mallory's substitution.*

Sample Output:

```
Without attacker -> Alice and Bob would share the same key.

--- MITM attack in progress ---
Mallory intercepts and substitutes her own public value for both sides.
Alice <-> Mallory shared key: 12 12
Mallory <-> Bob shared key  : 8 8
Alice-Bob direct key would differ, proving Mallory can decrypt and re-encrypt all traffic.
```

10.3 Secure Socket Layer (SSL) Communication

Aim:

Implement Secure Socket Layer (SSL/TLS) communication between a client and a server.

Hint:

1. Generate a self-signed certificate and private key for testing, e.g. using the openssl command-line tool.
2. On the server, create an ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER) and load the certificate with load_cert_chain().
3. Wrap a normal TCP socket with context.wrap_socket() on both the server and client sides.
4. On the client, create an ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT) and load the CA certificate to verify the server.
5. Exchange an encrypted message and print the negotiated TLS cipher suite using sslsock.cipher().

Program:

```
# ssl_server.py
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain(certfile='server.crt', keyfile='server.key')

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('127.0.0.1', 8443)); sock.listen(1)
print('SSL server listening on port 8443...')
```

```
with context.wrap_socket(sock, server_side=True) as ssock:
    conn, addr = ssock.accept()
    print('TLS handshake complete with', addr, '| cipher:', conn.cipher())
    print('Received:', conn.recv(1024).decode())
    conn.sendall(b'Hello securely from server!')
    conn.close()

# ssl_client.py
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('server.crt')

with socket.create_connection(('127.0.0.1', 8443)) as sock:
    with context.wrap_socket(sock, server_hostname='127.0.0.1') as ssock:
        ssock.sendall(b'Hello securely from client!')
        print('Server replied:', ssock.recv(1024).decode())
```

Try: *Set context.verify_mode = ssl.CERT_REQUIRED and enable mutual TLS by also having the server verify a client certificate.*

Sample Output:

```
SSL server listening on port 8443...
TLS handshake complete with ('127.0.0.1', 51844) | cipher: ('TLS_AES_256_GCM_SHA384', 'TLSv1.3', 256)
Received: Hello securely from client!
Server replied: Hello securely from server!
```

11: Authentication & Access Control

11.1 Username–Password Authentication System

Aim:

Implement a username–password authentication system.

Hint:

1. Never store plain-text passwords; use a salted hash via hashlib.pbkdf2_hmac() or bcrypt.
2. Store a dictionary/table of username -> (salt, password_hash) pairs to simulate a user database.
3. On registration, generate a random salt with os.urandom() and hash the password with it.
4. On login, recompute the hash of the entered password with the stored salt and compare using a constant-time check.
5. Lock the account or add a delay after several consecutive failed login attempts to resist brute-force attacks.

Program:

```
# auth_system.py
import hashlib, os, hmac
```

```
users = {}

def register(username, password):
    salt = os.urandom(16)
    pw_hash = hashlib.pbkdf2_hmac('sha256', password.encode(), salt, 100_000)
    users[username] = (salt, pw_hash)
    print(f'User "{username}" registered successfully')

def login(username, password):
    if username not in users:
        print('Login failed: user not found')
        return False
    salt, stored_hash = users[username]
    attempt_hash = hashlib.pbkdf2_hmac('sha256', password.encode(), salt, 100_000)
    if hmac.compare_digest(attempt_hash, stored_hash):
        print(f'User "{username}" logged in successfully')
        return True
    print('Login failed: incorrect password')
    return False

if __name__ == '__main__':
    register('alice', 'S3cureP@ss')
    login('alice', 'S3cureP@ss')
    login('alice', 'wrongpassword')
```

Try: *Add account lockout after 3 consecutive failed attempts, unlocking automatically after a 30-second cooldown.*

Sample Output:

```
User "alice" registered successfully
User "alice" logged in successfully
Login failed: incorrect password
```

11.2 Role-Based Access Control (RBAC)

Aim:

Implement a Role-Based Access Control (RBAC) system.

Hint:

1. Define a set of roles (e.g., admin, editor, viewer), each mapped to a set of permitted actions/permissions.
2. Maintain a mapping of users to one or more assigned roles.
3. Implement has_permission(user, action) that checks whether any of the user's roles grants the requested action.
4. Implement an authorize() decorator or function to guard sensitive operations using has_permission().

5. Demonstrate access being granted for an authorized role and denied for an unauthorized one.

Program:

```
# rbac.py
roles_permissions = {
    'admin':  {'create', 'read', 'update', 'delete'},
    'editor': {'create', 'read', 'update'},
    'viewer': {'read'},
}

user_roles = {
    'alice': ['admin'],
    'bob':   ['editor'],
    'carol': ['viewer'],
}

def has_permission(user, action):
    for role in user_roles.get(user, []):
        if action in roles_permissions.get(role, set()):
            return True
    return False

def perform_action(user, action):
    if has_permission(user, action):
        print(f'{user}: ALLOWED to {action}')
    else:
        print(f'{user}: DENIED to {action}')

if __name__ == '__main__':
    perform_action('alice', 'delete')
    perform_action('bob', 'delete')
    perform_action('carol', 'read')
    perform_action('carol', 'update')
```

Try: *Add support for role hierarchies, where 'admin' automatically inherits all permissions of 'editor' and 'viewer'.*

Sample Output:

```
alice: ALLOWED to delete
bob: DENIED to delete
carol: ALLOWED to read
carol: DENIED to update
```

11.3 Kerberos Authentication Protocol Demonstration

Aim:

Demonstrate the working of the Kerberos authentication protocol using a simplified simulation.

Hint:

1. Simulate three entities: the Client, the Authentication Server (AS), and the Ticket Granting Server (TGS).
2. The client requests a Ticket Granting Ticket (TGT) from the AS by supplying its identity.
3. The AS returns a TGT encrypted with the TGS's secret key, plus a session key encrypted with the client's secret key.
4. The client then presents the TGT to the TGS to obtain a Service Ticket for a specific application server.
5. Finally, the client presents the Service Ticket (encrypted with the service's key) to the application server to gain access.

Program:

```
# kerberos_simulation.py
import hashlib, json

def key_for(name):
    return hashlib.sha256(name.encode()).digest()[:8]

client_key, tgs_key, service_key = key_for('client'), key_for('tgs'), key_for('printer-service')

def xor_encrypt(data: bytes, key: bytes) -> bytes:
    return bytes(b ^ key[i % len(key)] for i, b in enumerate(data))

def step1_as_request(client_id):
    session_key_c_tgs = key_for('session-c-tgs')
    tgt = xor_encrypt(json.dumps({'client': client_id, 'session_key': session_key_c_tgs.hex()}).encode(), tgs_key)
    enc_session_key = xor_encrypt(session_key_c_tgs, client_key)
    print('AS -> Client: issued TGT (encrypted with TGS key) + session key (encrypted with client key)')
    return tgt, enc_session_key, session_key_c_tgs

def step2_tgs_request(tgt, service_id):
    tgt_data = json.loads(xor_encrypt(tgt, tgs_key))
    session_key_c_srv = key_for('session-c-service')
    service_ticket = xor_encrypt(json.dumps({'client': tgt_data['client'], 'session_key': session_key_c_srv.hex()}).encode(),
service_key)
    print(f'TGS -> Client: issued Service Ticket for "{service_id}"')
    return service_ticket, session_key_c_srv

def step3_service_access(service_ticket):
    ticket_data = json.loads(xor_encrypt(service_ticket, service_key))
    print(f"Service grants access to client '{ticket_data['client']}'")

if __name__ == '__main__':
    tgt, enc_sk, sk = step1_as_request('alice')
    service_ticket, sk2 = step2_tgs_request(tgt, 'printer-service')
    step3_service_access(service_ticket)
```

Try: *Add a timestamp and lifetime field to each ticket, and reject the service ticket if it has expired.*

Sample Output:

```
AS -> Client: issued TGT (encrypted with TGS key) + session key (encrypted with client key)
TGS -> Client: issued Service Ticket for "printer-service"
Service grants access to client 'alice'
```

12: Web & System Security – I

12.1 SQL Injection Attack Demonstration

Aim:

Implement and demonstrate a SQL Injection attack on a vulnerable query.

Hint:

1. Create a small SQLite database with a users table (id, username, password) using the sqlite3 module.
2. Write a deliberately vulnerable login function that builds a query using plain Python string formatting/concatenation.
3. Show how an attacker input such as "' OR '1'='1" bypasses authentication because it alters the query's logic.
4. Print the raw SQL string being executed so the injection is clearly visible.
5. Keep this experiment isolated in a local SQLite file for safe, offline demonstration.

Program:

```
# sql_injection_attack.py
import sqlite3

conn = sqlite3.connect(':memory:')
cur = conn.cursor()
cur.execute('CREATE TABLE users (id INTEGER PRIMARY KEY, username TEXT, password TEXT)')
cur.execute("INSERT INTO users VALUES (1, 'admin', 'admin123')")
conn.commit()

def vulnerable_login(username, password):
    query = f"SELECT * FROM users WHERE username='{username}' AND password='{password}'"
    print('Executing query:', query)
    cur.execute(query)
    return cur.fetchone()

if __name__ == '__main__':
    print('Normal login attempt:')
    print(vulnerable_login('admin', 'admin123'))

    print()
    print('SQL Injection attempt:')
    result = vulnerable_login("' OR '1'='1", "' OR '1'='1")
    print('Bypassed authentication! Row returned:', result)
```

Try: *Craft an injection payload that also extracts the database schema using UNION SELECT against sqlite_master.*

Sample Output:

```
Normal login attempt:
Executing query: SELECT * FROM users WHERE username='admin' AND password='admin123'
(1, 'admin', 'admin123')

SQL Injection attempt:
Executing query: SELECT * FROM users WHERE username='' OR '1'='1' AND password='' OR '1'='1'
Bypassed authentication! Row returned: (1, 'admin', 'admin123')
```

12.2 SQL Injection Prevention

Aim:

Implement SQL Injection prevention using parameterized queries and input validation.

Hint:

1. Rewrite the login query using parameterized placeholders (?) instead of string interpolation.
2. Pass user-supplied values as a separate tuple to cursor.execute(query, params) so the driver escapes them safely.
3. Add input validation (e.g., reject unexpected characters like quotes) as a defense-in-depth measure.
4. Re-run the same injection payload used in 12.1 and confirm it no longer bypasses authentication.
5. Optionally use an ORM (like SQLAlchemy) which parameterizes queries by default.

Program:

```
# sql_injection_prevention.py
import sqlite3

conn = sqlite3.connect(':memory:')
cur = conn.cursor()
cur.execute('CREATE TABLE users (id INTEGER PRIMARY KEY, username TEXT, password TEXT)')
cur.execute("INSERT INTO users VALUES (1, 'admin', 'admin123')")
conn.commit()

def secure_login(username, password):
    query = 'SELECT * FROM users WHERE username=? AND password=?'
    cur.execute(query, (username, password))   # values are safely parameterized
    return cur.fetchone()

if __name__ == '__main__':
    print('Normal login attempt:')
    print(secure_login('admin', 'admin123'))

    print()
    print('Same SQL Injection payload as before:')
```

```
result = secure_login("' OR '1'='1", "' OR '1'='1")
print('Injection blocked, row returned:', result)
```

Try: *Add a regex-based input validator that rejects usernames containing quotes, semicolons, or SQL keywords before the query even runs.*

Sample Output:

```
Normal login attempt:
(1, 'admin', 'admin123')

Same SQL Injection payload as before:
Injection blocked, row returned: None
```

13: Web & System Security – II

13.1 Firewall Rule Simulation

Aim:

Implement a simulation of firewall rule matching and packet filtering.

Hint:

1. Represent each firewall rule as a dictionary with source IP/port, destination IP/port, protocol, and action (ALLOW/DENY).
2. Store rules in an ordered list, since firewalls typically apply the first matching rule.
3. Support wildcard values ('*' or 'ANY') for fields that should match any value.
4. Write a match_packet(packet, rule) function that checks each field of an incoming packet against a rule.
5. Iterate through the rule list for each simulated packet and print whether it is ALLOWED or DENIED, with a default DENY if no rule matches.

Program:

```
# firewall_simulation.py
rules = [
    {'src': '192.168.1.10', 'dst': 'ANY', 'port': 22,  'proto': 'TCP', 'action': 'DENY'},
    {'src': 'ANY',          'dst': 'ANY', 'port': 80,  'proto': 'TCP', 'action': 'ALLOW'},
    {'src': 'ANY',          'dst': 'ANY', 'port': 443, 'proto': 'TCP', 'action': 'ALLOW'},
]

def match(packet, rule):
    return all(rule[f] in ('ANY', packet[f]) for f in ('src', 'dst', 'port', 'proto'))

def filter_packet(packet):
    for rule in rules:
        if match(packet, rule):
            print(f"Packet {packet} -> {rule['action']} (matched rule)")
```

```
            return rule['action']
    print(f'Packet {packet} -> DENY (default rule)')
    return 'DENY'

if __name__ == '__main__':
    filter_packet({'src': '192.168.1.10', 'dst': '10.0.0.5', 'port': 22, 'proto': 'TCP'})
    filter_packet({'src': '10.0.0.7', 'dst': '10.0.0.5', 'port': 80, 'proto': 'TCP'})
    filter_packet({'src': '10.0.0.7', 'dst': '10.0.0.5', 'port': 3389, 'proto': 'TCP'})
```

Try: *Add support for CIDR-based IP ranges (e.g., using Python's ipaddress module) instead of exact IP matches.*

Sample Output:

```
Packet {'src': '192.168.1.10', 'dst': '10.0.0.5', 'port': 22, 'proto': 'TCP'} -> DENY (matched rule)
Packet {'src': '10.0.0.7', 'dst': '10.0.0.5', 'port': 80, 'proto': 'TCP'} -> ALLOW (matched rule)
Packet {'src': '10.0.0.7', 'dst': '10.0.0.5', 'port': 3389, 'proto': 'TCP'} -> DENY (default rule)
```

13.2 Intrusion Detection System (IDS) Simulation

Aim:

Implement a simulation of a signature-based Intrusion Detection System (IDS).

Hint:

1. Maintain a list of known attack signatures, e.g., patterns like repeated failed logins or suspicious payload strings.
2. Model a stream of simulated log entries or packets as a list of dictionaries with fields like source IP and payload.
3. For signature-based detection, check each log entry's payload against known malicious patterns (e.g., "' OR 1=1", "<script>").
4. For anomaly-based detection, count failed login attempts per source IP within a short simulated time window and flag it if it exceeds a threshold.
5. Log every detected intrusion attempt with the reason for the alert.

Program:

```
# ids_simulation.py
from collections import defaultdict

signatures = ["' OR 1=1", '<script>', 'DROP TABLE', '../../etc/passwd']
failed_login_threshold = 3

traffic_log = [
    {'src': '10.0.0.5', 'event': 'login_failed'},
    {'src': '10.0.0.5', 'event': 'login_failed'},
    {'src': '10.0.0.5', 'event': 'login_failed'},
    {'src': '10.0.0.9', 'event': 'http_request', 'payload': "id=1' OR 1=1 --"},
```

```
    {'src': '10.0.0.9', 'event': 'http_request', 'payload': 'q=hello world'},
]

failed_attempts = defaultdict(int)

def signature_check(payload):
    return [sig for sig in signatures if sig.lower() in payload.lower()]

for entry in traffic_log:
    if entry['event'] == 'login_failed':
        failed_attempts[entry['src']] += 1
        if failed_attempts[entry['src']] >= failed_login_threshold:
            print(f"ALERT: possible brute-force from {entry['src']} "
                  f"({failed_attempts[entry['src']]} failed logins)")
    elif entry['event'] == 'http_request':
        matched = signature_check(entry['payload'])
        if matched:
            print(f"ALERT: signature match {matched} from {entry['src']} -> {entry['payload']}")
        else:
            print(f"OK: normal traffic from {entry['src']}")
```

Try: *Add a sliding time-window (e.g., using timestamps) so old failed-login events no longer count toward the brute-force threshold.*

Sample Output:

```
ALERT: possible brute-force from 10.0.0.5 (3 failed logins)
ALERT: signature match ["' OR 1=1"] from 10.0.0.9 -> id=1' OR 1=1 --
OK: normal traffic from 10.0.0.9
```

14: Blockchain & Advanced Security

14.1 Implement a Simple Blockchain

Aim:

Implement a simple blockchain data structure in Python.

Hint:

1. Define a Block class holding index, timestamp, data, previous_hash, and its own computed hash.
2. Compute each block's hash using SHA-256 over its contents (index + timestamp + data + previous_hash).
3. Create a genesis block (the first block) with a hard-coded previous_hash of '0'.
4. Implement add_block() on a Blockchain class that links a new block to the previous block's hash.
5. Implement is_chain_valid() that recomputes hashes and checks linkage to detect any tampering.

Program:

```
# simple_blockchain.py
import hashlib, time, json

class Block:
    def __init__(self, index, data, previous_hash):
        self.index = index
        self.timestamp = time.time()
        self.data = data
        self.previous_hash = previous_hash
        self.hash = self.compute_hash()

    def compute_hash(self):
        content = json.dumps({
            'index': self.index, 'timestamp': self.timestamp,
            'data': self.data, 'previous_hash': self.previous_hash
        }, sort_keys=True)
        return hashlib.sha256(content.encode()).hexdigest()

class Blockchain:
    def __init__(self):
        self.chain = [Block(0, 'Genesis Block', '0')]

    def add_block(self, data):
        prev = self.chain[-1]
        self.chain.append(Block(len(self.chain), data, prev.hash))

    def is_valid(self):
        for i in range(1, len(self.chain)):
            cur, prev = self.chain[i], self.chain[i - 1]
            if cur.hash != cur.compute_hash() or cur.previous_hash != prev.hash:
                return False
        return True

if __name__ == '__main__':
    bc = Blockchain()
    bc.add_block('Alice pays Bob 10 coins')
    bc.add_block('Bob pays Carol 5 coins')
    for block in bc.chain:
        print(f'Block {block.index}: hash={block.hash[:16]}... prev={block.previous_hash[:16]}...')
    print('Blockchain valid?', bc.is_valid())

    bc.chain[1].data = 'Alice pays Bob 1000 coins'  # tamper
    print('Blockchain valid after tampering?', bc.is_valid())
```

Try: *Add a merkle-root field to each block that summarizes multiple transactions instead of a single data string.*

Sample Output:

```
Block 0: hash=6f2c1a9e0d3b4f8a... prev=0000000000000000...
Block 1: hash=a1e7c8f209b34d5e... prev=6f2c1a9e0d3b4f8a...
```

```
Block 2: hash=de90f3b1c7a24e68... prev=a1e7c8f209b34d5e...
Blockchain valid? True
Blockchain valid after tampering? False
```

14.2 Proof-of-Work Consensus Mechanism

Aim:

Demonstrate the Proof-of-Work (PoW) consensus mechanism used in blockchain mining.

Hint:

1. Extend the Block class with a nonce field and a difficulty target (e.g., hash must start with N leading zeros).
2. Implement mine_block() that repeatedly increments the nonce and recomputes the hash until it satisfies the difficulty target.
3. Measure and print how many attempts (nonce values) were needed to find a valid hash.
4. Increase the difficulty and observe how the mining time grows, illustrating PoW's computational cost.
5. Use the mined block's hash as the previous_hash for the next block, exactly as in real blockchains.

Program:

```
# proof_of_work.py
import hashlib, json, time

def compute_hash(index, data, previous_hash, nonce):
    content = json.dumps({'index': index, 'data': data,
                          'previous_hash': previous_hash, 'nonce': nonce}, sort_keys=True)
    return hashlib.sha256(content.encode()).hexdigest()

def mine_block(index, data, previous_hash, difficulty=4):
    nonce = 0
    target = '0' * difficulty
    start = time.time()
    while True:
        h = compute_hash(index, data, previous_hash, nonce)
        if h.startswith(target):
            elapsed = time.time() - start
            print(f'Block mined! nonce={nonce}, hash={h}, time={elapsed:.3f}s')
            return nonce, h
        nonce += 1

if __name__ == '__main__':
    print('Mining with difficulty 4 (hash must start with "0000")...')
    nonce, block_hash = mine_block(1, 'Alice pays Bob 10 coins', '0' * 64, difficulty=4)
```

Try: *Increase the difficulty to 5 and 6 leading zeros and record how the mining time increases exponentially.*

Sample Output:

```
Mining with difficulty 4 (hash must start with "0000")...
Block mined! nonce=94823, hash=0000a3f1c8e02d4b..., time=0.412s
```

14.3 Secure Data Storage using Hashing and Encryption

Aim:

Implement secure data storage that combines hashing (for integrity) and encryption (for confidentiality).

Hint:

1. Encrypt the sensitive data using AES (Crypto.Cipher.AES) before writing it to storage.
2. Compute a SHA-256 hash of the original plaintext to serve as an integrity checksum.
3. Store the ciphertext, IV, and the integrity hash together (e.g., as a JSON record).
4. On retrieval, decrypt the ciphertext first, then recompute the hash of the decrypted data and compare it with the stored hash.
5. Report a tampering/corruption alert if the recomputed hash does not match the stored one.

Program:

```
# secure_storage.py
import hashlib, json, base64
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

key = get_random_bytes(16)

def secure_store(plaintext: str):
    integrity_hash = hashlib.sha256(plaintext.encode()).hexdigest()
    iv = get_random_bytes(16)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block_size))
    record = {
        'ciphertext': base64.b64encode(ciphertext).decode(),
        'iv': base64.b64encode(iv).decode(),
        'hash': integrity_hash,
    }
    print('Stored record:', record)
    return record

def secure_retrieve(record, tamper=False):
    iv = base64.b64decode(record['iv'])
    ciphertext = base64.b64decode(record['ciphertext'])
    if tamper:
        ciphertext = bytes([ciphertext[0] ^ 0xFF]) + ciphertext[1:]
```

```
    cipher = AES.new(key, AES.MODE_CBC, iv)
    try:
        plaintext = unpad(cipher.decrypt(ciphertext), AES.block_size).decode()
    except Exception:
        print('Decryption failed: data corrupted')
        return None
    if hashlib.sha256(plaintext.encode()).hexdigest() != record['hash']:
        print('INTEGRITY CHECK FAILED: data has been tampered with')
        return None
    print('Integrity verified. Retrieved data:', plaintext)
    return plaintext

if __name__ == '__main__':
    record = secure_store('Top secret distributed systems exam paper')
    secure_retrieve(record)
    secure_retrieve(record, tamper=True)
```

Try: *Replace the SHA-256 checksum with an HMAC keyed with a second secret key, so integrity cannot be forged even if an attacker also knows the encryption key.*

Sample Output:

```
Stored record: {'ciphertext': 'Qk3F...', 'iv': 'Z9e1...', 'hash': '7c1e4f...'}
Integrity verified. Retrieved data: Top secret distributed systems exam paper
Decryption failed: data corrupted
```

V. TEXT BOOKS

1. Distributed Systems: Concepts and Design – George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair, 6th Edition, Pearson, 2019.
2. Cryptography and Network Security: Principles and Practice – William Stallings, 8th Edition, Pearson, 2020.

VI. REFERENCE BOOKS

1. Distributed Systems: Principles and Paradigms – Andrew S. Tanenbaum, Maarten Van Steen, 2nd Edition, Pearson, 2007.
2. Network Security Essentials: Applications and Standards – William Stallings, 6th Edition, Pearson, 2019.

VII. WEB REFERENCES

1. https://www.geeksforgeeks.org/category/distributed-systems/