# LAB PROGRAMS

## ⭐ Q1: TCP Client/Server for Sending Encrypted Message

**Objective:**

To implement a TCP client–server application where the client sends a plaintext message, encrypts it (Caesar/substitution), and the server receives and decrypts it.

**SERVER PSEUDOCODE**

1. START
2. CREATE a TCP socket using AF_INET, SOCK_STREAM # Create server socket
3. BIND the socket to SERVER_IP and PORT # Attach address
4. LISTEN for incoming TCP connections # Wait for client

5. ACCEPT a connection from client # Client connected
6. RECEIVE encrypted data from client # Get ciphertext

7. DECRYPT received data using known key/algorithm # Convert to plaintext
8. PRINT decrypted message # Show actual message

9. PREPARE response message # Example: "Hello Client"
10. ENCRYPT response message using same key/algorithm # Convert to ciphertext
11. SEND encrypted response to client # Send encrypted reply

12. CLOSE client connection # End session
13. STOP

**CLIENT PSEUDOCODE**

1. START
2. CREATE a TCP socket using AF_INET, SOCK_STREAM # Create client socket
3. CONNECT socket to SERVER_IP and PORT # Establish TCP connection

4. READ plaintext message from user # Input data
5. ENCRYPT plaintext using agreed key/algorithm # Get ciphertext

6. SEND encrypted message to server # Transmit data
7. RECEIVE encrypted reply from server # Get response

8. DECRYPT encrypted reply using same key/algorithm # Convert to plaintext

9. PRINT decrypted reply # Show server response

10. CLOSE the connection # End session
11. STOP

**Expected Output (Example):**
**Server side:**
Server listening on port 5000...
Connected from ('127.0.0.1', 54512)
Encrypted message: khoor
Decrypted message: hello

**Client side:**
Enter message: hello

**Viva Questions (One-word answers):**
1. Protocol used here? -> TCP
2. Socket type used? -> STREAM
3. Encryption type used?-> SYMMETRIC
4. TCP connection style?-> CONNECTION-ORIENTED
5. Default transport port range type? -> EPHEMERAL

## ⭐ Q2: UDP Client/Server Application

### Objective:

To implement a connectionless UDP client–server program that sends and receives simple text messages.

**SERVER PSEUDOCODE**

1. START
2. CREATE a UDP socket using AF_INET, SOCK_DGRAM # Create UDP socket
3. BIND socket to SERVER_IP and PORT # Attach address

4. LOOP forever # Continuous service
5. RECEIVE datagram and CLIENT_ADDRESS using recvfrom # Get data + address
6. DECODE received data # Convert to string
7. PRINT client address and message # Display

8. PREPARE reply message (e.g., "ACK: " + message) # Build response
9. SEND reply datagram to CLIENT_ADDRESS using sendto # Echo back
10. END LOOP

11. CLOSE UDP socket # Stop server (if needed)
12. STOP

**CLIENT PSEUDOCODE**

1. START
2. CREATE a UDP socket using AF_INET, SOCK_DGRAM # Create UDP socket

3. READ message from user # Input data
4. ENCODE message # Convert to bytes

5. SEND datagram to SERVER_IP and PORT using sendto # Transmit message
6. RECEIVE reply datagram and SERVER_ADDRESS using recvfrom # Get response

7. DECODE reply message # Convert to string
8. PRINT reply message # Show ACK

9. CLOSE UDP socket # End session
10. STOP

**Expected Output (Example):**

**Server:**

UDP server on port 6000...
From ('127.0.0.1', 54321) : test
From ('127.0.0.1', 54321) : hello

**Client:**

Enter message: hello
Reply from server: ACK: hello

**Viva Questions (One-word answers):**

1. Protocol used? -> UDP
2. Connection type? -> CONNECTIONLESS
3. Reliability in UDP? -> NO
4. Basic server call to receive?-> RECVFROM
5. Field for port in UDP header?-> PORT

## ⭐ Q3: Multi-Protocol Server (TCP + UDP) with Two Clients

**Objective:**

To design a single server that handles both TCP and UDP sockets, serving
TCP client requests and UDP client messages concurrently (sequentially in demo).

**SERVER PSEUDOCODE**

1. START
2. CREATE TCP socket using AF_INET, SOCK_STREAM # For TCP clients
3. SET TCP socket options if needed # Optional
4. BIND TCP socket to SERVER_IP and TCP_PORT
5. LISTEN for incoming TCP connections

6. CREATE UDP socket using AF_INET, SOCK_DGRAM # For UDP clients
7. BIND UDP socket to SERVER_IP and UDP_PORT # Same or different port

8. ADD TCP and UDP sockets to SOCKET_SET # For multiplexing

9. LOOP forever
10. WAIT on SOCKET_SET using select() # Multiplex I/O

11. IF TCP socket is ready for read # New TCP client
12. ACCEPT TCP connection from client # Get TCP client socket
13. RECEIVE data from TCP client # Read message
14. PREPARE "TCP-ACK: " + message # Build response
15. SEND response to TCP client # Send ACK
16. CLOSE TCP client socket # Close connection
17. ENDIF

18. IF UDP socket is ready for read # UDP datagram arrived
19. RECEIVE datagram and CLIENT_ADDRESS using recvfrom
20. PREPARE "UDP-ACK: " + message # Build response
21. SEND reply datagram to CLIENT_ADDRESS using sendto
22. ENDIF
23. END LOOP

24. CLOSE TCP and UDP sockets # Shutdown server
25. STOP

**TCP CLIENT PSEUDOCODE**

1. START
2. CREATE TCP socket using AF_INET, SOCK_STREAM
3. CONNECT to SERVER_IP and TCP_PORT

4. SEND message "hello tcp" to server
5. RECEIVE reply from server
6. PRINT reply

7. CLOSE TCP socket
8. STOP

**UDP CLIENT PSEUDOCODE**

1. START
2. CREATE UDP socket using AF_INET, SOCK_DGRAM
3. PREPARE message "hello udp"

4. SEND datagram to SERVER_IP and UDP_PORT using sendto
5. RECEIVE reply datagram from server
6. PRINT reply

7. CLOSE UDP socket
8. STOP

**Expected Output (Example):**

**Server:**

Multi-protocol server on TCP/UDP port 7000...
TCP from ('127.0.0.1', 50010) : hello tcp
UDP from ('127.0.0.1', 50011) : hello udp

**Viva Questions (One-word answers):**

1. API used to monitor sockets? -> SELECT
2. No. of protocols supported? -> TWO
3. TCP socket type? -> STREAM
4. UDP socket type? -> DGRAM
5. Multiplexing mechanism name? -> I/O

## ⭐ Q4: MAC-Based Authentication

**Objective:**

To implement message authentication using a Message Authentication Code (MAC) so the receiver verifies integrity and authenticity of a message.

**SENDER PSEUDOCODE**

1. START
2. SHARE secret key K with receiver # Pre-shared key
3. READ plaintext message from user # Input
4. COMPUTE MAC = HMAC(K, message) # Message auth code
5. SEND (message, MAC) to receiver # Transmit both
6. STOP

**RECEIVER PSEUDOCODE**

1. START
2. RECEIVE (message, MAC_recv) from sender # Get data
3. COMPUTE MAC_calc = HMAC(K, message) using same key # Recompute MAC
4. IF MAC_calc == MAC_recv # Compare
5. PRINT "Authentication SUCCESS" # Valid data
6. ACCEPT message # Use message
7. ELSE
8. PRINT "Authentication FAILURE" # Tampered or wrong
9. REJECT message # Discard
10. ENDIF
11. STOP

**Expected Output (Example):**

Enter message: hello
Send this to receiver:
Message: hello
MAC : 1a2b3c...
Authentication: SUCCESS

**Viva Questions (One-word answers):**

1. MAC expansion? -> AUTHENTICATION
2. Algorithm type? -> SYMMETRIC
3. Function used here? -> HMAC
4. Property verified? -> INTEGRITY
5. Key shared between? -> PARTIES

## ⭐ Q5: Diffie–Hellman Key Exchange

**Objective:**

To demonstrate Diffie–Hellman key exchange between client and server to derive a shared secret key over an insecure channel.

**PSEUDOCODE**

1. START
2. AGREE on large prime p and generator g (public)  # System parameters

3. SERVER:
4. CHOOSE private value a randomly  # Server secret
5. COMPUTE A = g^a mod p  # Server public value
6. SEND A to client  # Public share

7. CLIENT:
8. CHOOSE private value b randomly  # Client secret
9. COMPUTE B = g^b mod p  # Client public value
10. SEND B to server  # Public share

11. SERVER:
12. RECEIVE B from client
13. COMPUTE shared_key_server = B^a mod p  # Shared secret

14. CLIENT:
15. RECEIVE A from server
16. COMPUTE shared_key_client = A^b mod p  # Shared secret

17. VERIFY shared_key_server == shared_key_client  # Must match
18. USE shared_key as session key for encryption  # Secure channel
19. STOP

**Expected Output (Example):**

Public prime p: 23
Generator g : 5
Server public A: 8
Client public B: 19
Server shared key: 2
Client shared key: 2

**Viva Questions (One-word answers):**

1. DH is used to share? -> KEY
2. Channel security during DH? -> INSECURE
3. Main attacker type? -> MITM
4. Values p, g are? -> PUBLIC
5. Values a, b are? -> PRIVATE

## ⭐ Q6: TCP Server with Socket Options (SO_REUSEADDR, SO_RCVTIMEO)

**Objective:**

To write a TCP client–server program where the server sets important socket options like SO_REUSEADDR and SO_RCVTIMEO.

**SERVER PSEUDOCODE**

1. START
2. CREATE TCP socket using AF_INET, SOCK_STREAM # Create server socket
3. SET socket option SO_REUSEADDR = 1 # Allow port reuse
4. SET socket option SO_RCVTIMEO = TIMEOUT_VALUE # Set recv timeout
5. BIND socket to SERVER_IP and PORT # Attach address
6. LISTEN for incoming TCP connections # Wait for client
7. ACCEPT connection from client # Client connected
8. TRY
9. RECEIVE data from client using recv # Wait for data
10. IF data received # Not empty
11. PRINT received data # Show message
12. ELSE
13. PRINT "No data received"
14. ENDIF
15. CATCH timeout event # recv timeout
16. PRINT "Receive timed out" # No data in time
17. END TRY
18. CLOSE client connection # Close socket
19. CLOSE server socket # Shutdown server
20. STOP

**CLIENT PSEUDOCODE**

1. START
2. CREATE TCP socket using AF_INET, SOCK_STREAM
3. CONNECT to SERVER_IP and PORT
4. SEND message "hello with options" to server

5. CLOSE socket # End session
6. STOP

**Expected Output (Example):**

**Server:**

Server on port 8000...
Connected from ('127.0.0.1', 50100)
Received: hello with options

**Viva Questions (One-word answers):**

1. SO_REUSEADDR purpose? -> REUSE
2. SO_RCVTIMEO purpose? -> TIMEOUT
3. Level for these options? -> SOL_SOCKET
4. Timeout unit? -> SECONDS
5. TCP connection call? -> ACCEPT

## ⭐ Q7: Client–Server Using sendmsg() with Ancillary Data

**Objective:**

To demonstrate use of sendmsg() and ancillary data (control messages) in a client–server setting (Unix-only feature; demo focuses on structure).

**SERVER PSEUDOCODE**

1. START
2. CREATE TCP socket using AF_INET, SOCK_STREAM # Server socket
3. BIND to SERVER_IP and PORT
4. LISTEN for incoming TCP connections

5. ACCEPT connection from client # Get client socket
6. CALL recvmsg on client socket # Receive data + control
7. EXTRACT main data buffer from recvmsg # Payload
8. EXTRACT ancillary data (if any) # Control info

9. PRINT received data # Show message
10. PRINT ancillary data list # Debug info

11. CLOSE client socket
12. CLOSE server socket
13. STOP

**CLIENT PSEUDOCODE**

1. START
2. CREATE TCP socket using AF_INET, SOCK_STREAM # Client socket
3. CONNECT to SERVER_IP and PORT # Establish connection

4. PREPARE data buffer with message "hello via sendmsg" # Payload
5. OPTIONALLY PREPARE ancillary data structures # Control info

6. CALL sendmsg with data buffer (+ ancillary if any) # Send message
7. CLOSE client socket # End session
8. STOP

**Expected Output (Example):**

**Server:**

Server on 9000...
Connected from ('127.0.0.1', 50200)
Data: hello via sendmsg
Ancillary: []

**Viva Questions (One-word answers):**

1. System call used? -> SENDMSG
2. Extra metadata name? -> ANCILLARY
3. Descriptor passing via? -> CONTROL
4. Usual OS family? -> UNIX
5. Data type for extra info? -> CMSG

## ⭐ Q8: Simplified Ping Using Raw Sockets (ICMP)

**Objective:**

To implement a basic ping-like program using raw sockets to send ICMP echo request and receive ICMP echo reply.

**SERVER PSEUDOCODE**

1. START
2. SET DESTINATION_IP (e.g., 8.8.8.8) # Target host
3. CREATE raw socket using AF_INET, SOCK_RAW, IPPROTO_ICMP # ICMP raw socket
4. GENERATE unique IDENTIFIER (e.g., process ID) # For matching reply
5. BUILD ICMP echo request header (type 8, code 0) # ICMP fields
6. APPEND payload data (e.g., "pingdata") # ICMP body
7. COMPUTE checksum over header + data # ICMP checksum
8. SET checksum field in header # Final packet
9. RECORD current time as START_TIME # For RTT
10. SEND ICMP packet to DESTINATION_IP using sendto # Echo request
11. WAIT for reply packet using recvfrom # Block until reply
12. RECORD current time as END_TIME # After reply
13. PARSE received packet to locate ICMP header # Skip IP header
14. VERIFY type == 0 and identifier matches # Echo reply check
15. COMPUTE RTT = END_TIME - START_TIME # Round-trip time
16. PRINT DESTINATION_IP and RTT # Show statistics
17. CLOSE raw socket
18. STOP

**Expected Output (Example):**

Reply from: ('8.8.8.8', 0)
RTT (ms): 25.4

**Viva Questions (One-word answers):**

1. Protocol used by ping? -> ICMP
2. Socket type used? -> RAW
3. ICMP echo request type?-> EIGHT
4. ICMP echo reply type? -> ZERO
5. Required privilege? -> ROOT

## ⭐ Q9: Role-Based Access Control (RBAC) – Python Program

**Objective:**

To implement Role-Based Access Control so that operations are allowed based on user roles (ADMIN, USER, GUEST, etc.).

**SERVER PSEUDOCODE**

1. START
2. DEFINE ROLE_PERMISSIONS mapping # e.g., ADMIN/USER/GUEST
3. ADMIN -> {read, write, delete}
4. USER -> {read, write}
5. GUEST -> {read}

6. DEFINE USER_ROLES mapping # Users to roles
7. "alice" -> ADMIN
8. "bob" -> USER
9. "eve" -> GUEST

10. FUNCTION CHECK_ACCESS(user, action):
11. LOOKUP role = USER_ROLES[user] # Get role
12. IF role is NOT FOUND
13. RETURN DENY # Unknown user
14. ENDIF

15. LOOKUP allowed_set = ROLE_PERMISSIONS[role] # Permissions
16. IF action is in allowed_set
17. RETURN ALLOW
18. ELSE
19. RETURN DENY
20. ENDIF
21. END FUNCTION

22. FOR each (user, action) test case # Example calls
23. CALL CHECK_ACCESS(user, action)
24. PRINT "CAN" or "CANNOT" based on result

25. STOP

**Expected Output (Example):**
alice CAN delete
bob CANNOT delete
eve CAN read

**Viva Questions (One-word answers):**
1. RBAC is based on? -> ROLE
2. Basic access decision?-> ALLOW
3. Higher privilege role?-> ADMIN
4. Lower privilege role? -> GUEST
5. Type of control? -> DISCRETIONARY

## ⭐ Q10: Attribute-Based Access Control (ABAC) – Python Program

**Objective:**

To implement Attribute-Based Access Control where decisions are made based on attributes of user, resource, and environment.

**SERVER PSEUDOCODE**

```
START
DEFINE USERS with attributes                    # User attributes
alice: dept = HR,  clearance = HIGH
bob:   dept = IT,  clearance = LOW

DEFINE RESOURCES with attributes                # Resource attributes
salary_db: owner_dept = HR, sensitivity = HIGH
log_file:  owner_dept = IT, sensitivity = LOW

DEFINE ENVIRONMENT attributes                   # Dynamic context
time = current system time

FUNCTION ABAC_POLICY(user_attrs, resource_attrs, env):
IF user_attrs.dept != resource_attrs.owner_dept      # Dept mismatch
RETURN DENY
ENDIF

IF resource_attrs.sensitivity == HIGH AND
user_attrs.clearance != HIGH                # Clearance check
RETURN DENY
ENDIF

EXTRACT hour from env.time                  # Current hour
IF hour < 9 OR hour > 18                    # Outside office hours
RETURN DENY
ENDIF

RETURN ALLOW                                # All conditions met
END FUNCTION

FUNCTION CHECK_ABAC(user_name, resource_name):
LOOKUP user_attrs from USERS
```

26. LOOKUP resource_attrs from RESOURCES
27. SET env.time = current system time

28. RESULT = ABAC_POLICY(user_attrs, resource_attrs, env)
29. IF RESULT == ALLOW
30. PRINT user_name "CAN ACCESS" resource_name
31. ELSE
32. PRINT user_name "CANNOT ACCESS" resource_name
33. ENDIF
34. END FUNCTION

35. CALL CHECK_ABAC for sample (alice, salary_db)
36. CALL CHECK_ABAC for sample (bob, salary_db)
37. CALL CHECK_ABAC for sample (bob, log_file)

38. STOP

**Expected Output (Example):**
alice CAN ACCESS salary_db
bob CANNOT ACCESS salary_db
bob CAN ACCESS log_file (only if time within policy window)

**Viva Questions (One-word answers):**
1. ABAC is based on? -> ATTRIBUTES
2. Core decision engine? -> POLICY
3. Example user attribute? -> DEPARTMENT
4. Example resource attribute?-> SENSITIVITY
5. Example environment attr?-> TIME