Mid Sem 2024 (old) Netsec

what is end to end encryption. explain how can it be used to secure app based chat

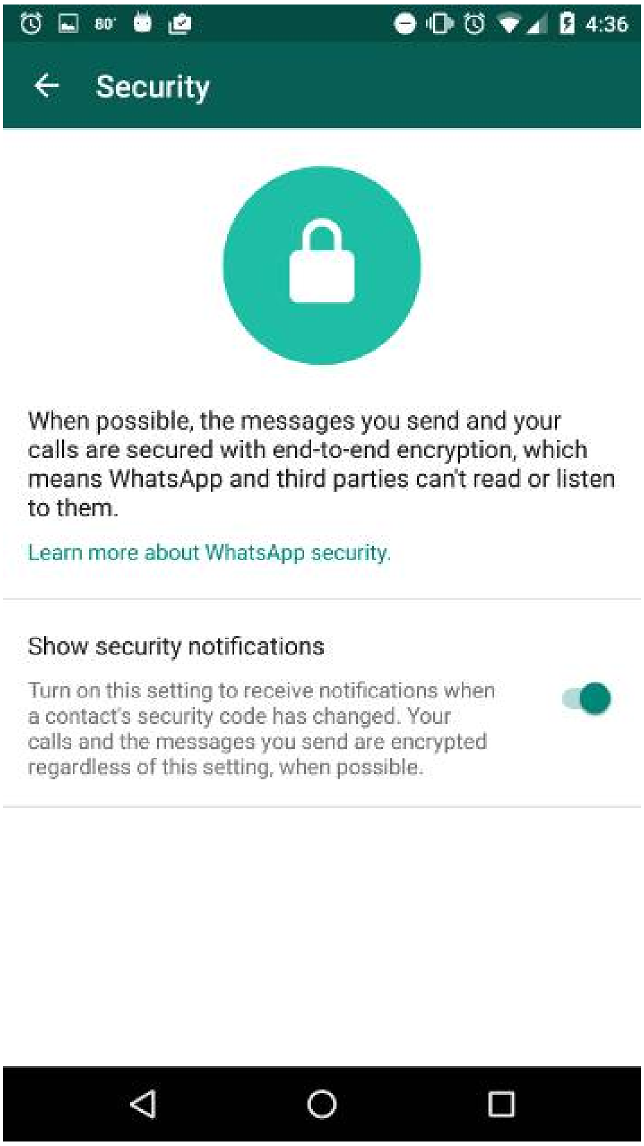
🔐 What is End-to-End Encryption (E2EE)?

End-to-End Encryption (E2EE) is a security method where:

- Data is encrypted **on the sender's device**
- It remains encrypted while traveling through servers/network
- It is decrypted **only on the recipient's device**

👉 No one in between — not ISPs, hackers, or even the service provider — can read the message.

📌 Example Apps That Use E2EE



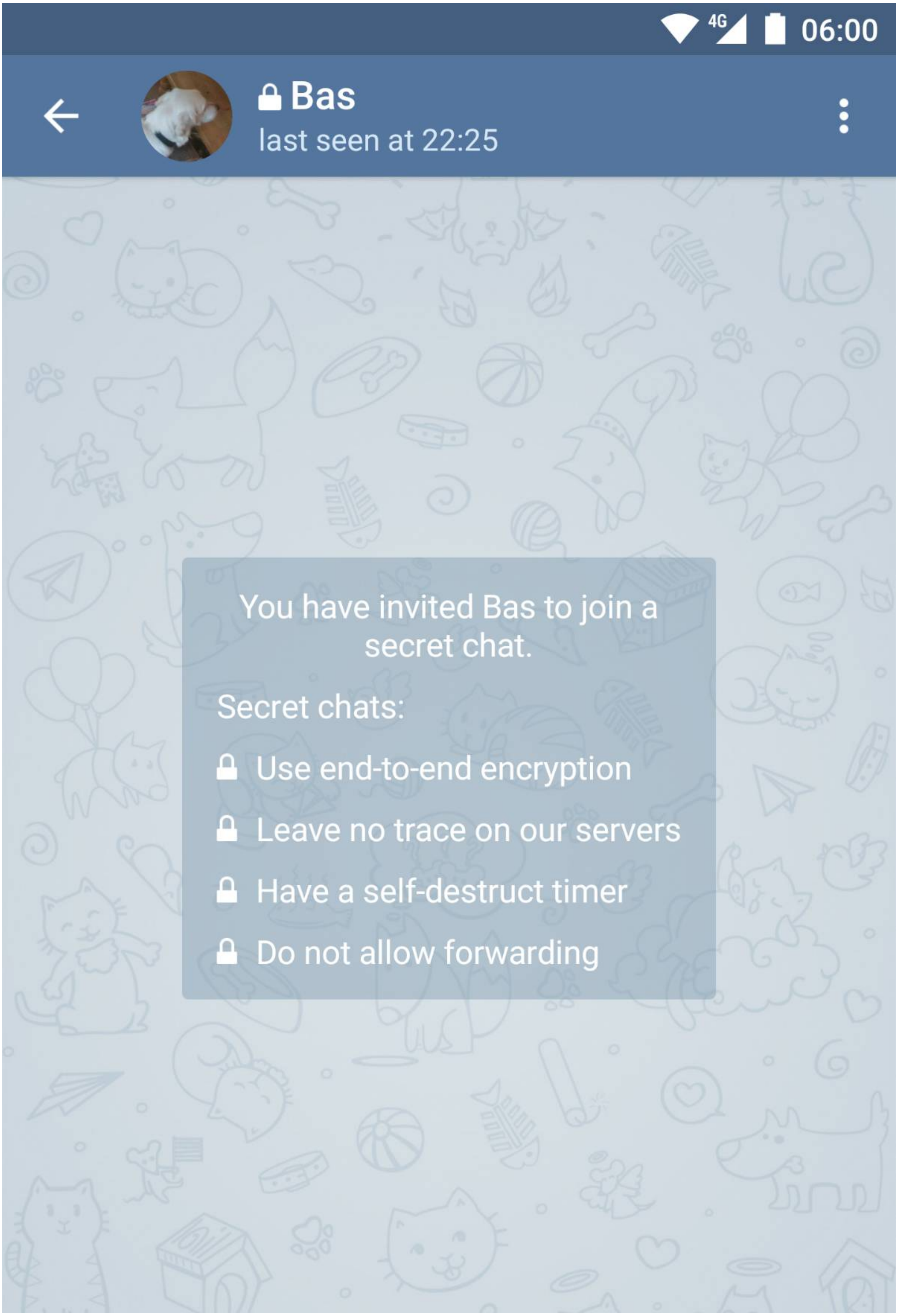
Security

When possible, the messages you send and your calls are secured with end-to-end encryption, which means WhatsApp and third parties can't read or listen to them.

Learn more about WhatsApp security.

Show security notifications

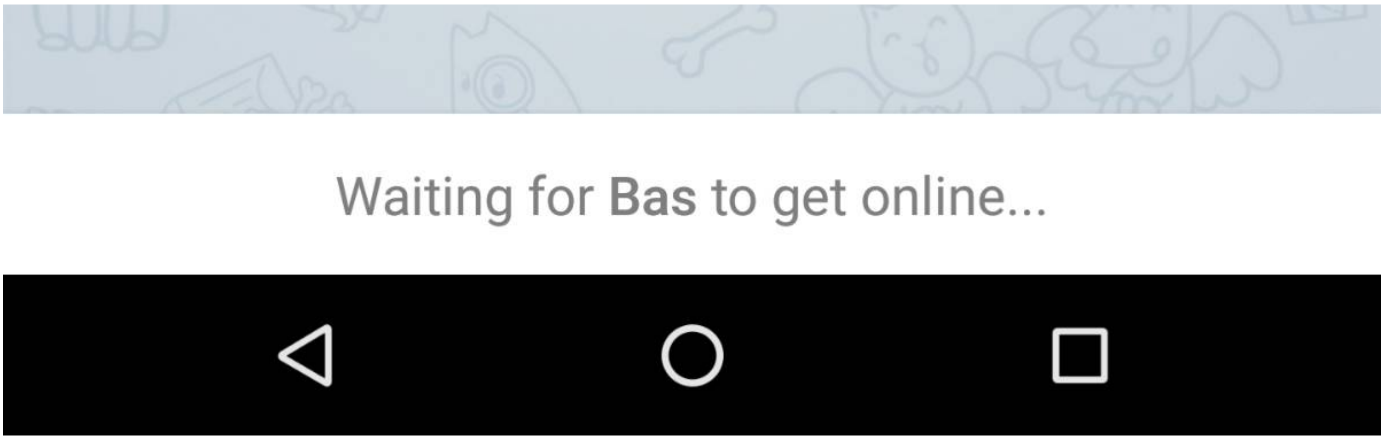
Turn on this setting to receive notifications when a contact's security code has changed. Your calls and the messages you send are encrypted regardless of this setting, when possible.



You have invited Bas to join a secret chat.

Secret chats:

- Use end-to-end encryption
- Leave no trace on our servers
- Have a self-destruct timer
- Do not allow forwarding



Waiting for Bas to get online...

Apps like:

- **WhatsApp**
- **Signal**
- **Telegram** (only Secret Chats)
- **iMessage**

use E2EE to protect conversations.

🔎 How E2EE Works (Conceptually)

Let’s say Alice sends a message to Bob.

Step 1: Key Generation

- Bob generates:
    - **Public key** → shared with everyone
    - **Private key** → kept secret

Step 2: Encrypting the Message

- Alice encrypts the message using Bob’s **public key**
- Ciphertext is sent over the internet

Step 3: Decryption

- Bob uses his **private key** to decrypt the message

Even if someone intercepts the data:

```pgsql
HELLO → (encrypted) → 8F3A9X!QZ
```

They cannot read it without Bob’s private key.

🛡️ How To Use E2EE in an App-Based Chat System

Now let's design it like a Computer Science student would.

1️⃣ Use Public-Key Cryptography for Key Exchange

Example:

- **Diffie-Hellman**
- **Elliptic Curve Diffie-Hellman**

Purpose:

- Securely establish a shared symmetric key between users

2️⃣ Use Symmetric Encryption for Messages

Once shared key is created:

- Use **AES**
- Or secure modes like:
  - AES-GCM
  - ChaCha20-Poly1305

Symmetric encryption is:
✔️ Fast
✔️ Efficient
✔️ Suitable for real-time chat

3️⃣ Use Authentication (Integrity + No Tampering)

Add:

- **Message Authentication Code**
- Or authenticated encryption (AES-GCM)

This ensures:

- Message is not modified
- Sender is authentic

4️⃣ Implement Perfect Forward Secrecy

Use:

- Ephemeral keys (new keys per session/message)
- Like in Signal Protocol

Even if long-term key is compromised:

- Old chats remain secure

🔐 Architecture of Secure Chat with E2EE

```
pgsql

User A Device
   |
Encrypt locally
   |
---- Encrypted Message ----> Server (stores ciphertext only)
                                   |
                             Cannot read data
                                   |
Decrypt locally
   |
User B Device
```

The server only stores encrypted blobs.

⚠️ Without End-to-End Encryption

If encryption is only between user ↔ server:

```
pgsql

User → Encrypted → Server → Decrypt → Re-encrypt → User
```

💀 Server can read messages
💀 Insider attacks possible
💀 Government or breach can expose chats

That is NOT true E2EE.

🧠 How Apps Like Signal Secure Chat

Signal Protocol uses:

- Double Ratchet Algorithm
- Prekeys
- X3DH key agreement
- Forward secrecy
- Post-compromise security

Used by:

- **Signal**
- **WhatsApp**

🔒 Security Properties Achieved

Property	How E2EE Provides It
Confidentiality	Only sender & receiver decrypt
Integrity	MAC / AEAD modes
Authentication	Public key identity
Forward Secrecy	Ephemeral keys
Post-compromise security	Key ratcheting

🚀 If You Were Building One

Since you're into backend + crypto systems:

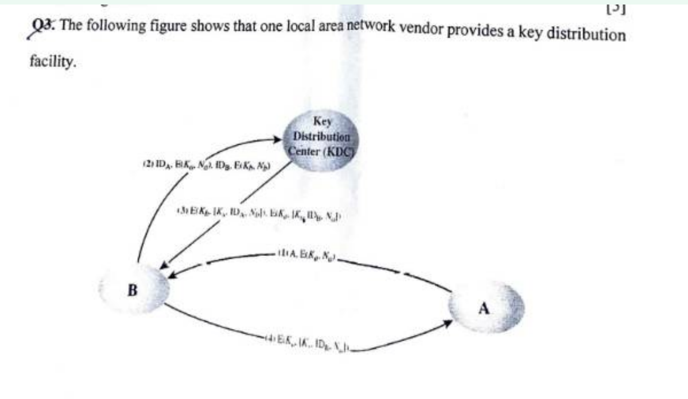
You'd need:

- Client-side key generation
- Secure key storage
- Secure random number generator
- Use battle-tested libraries (libsodium, OpenSSL)
- Never write custom crypto
- End-to-end key verification (QR code fingerprint)

If you want, I can next explain:

- 🔁 How Signal's Double Ratchet works (step-by-step)
- 🛠 How to implement E2EE in your own Node.js app
- 🧨 Common E2EE implementation mistakes
- 🔐 How E2EE differs from TLS

Tell me which direction you want 👌



[5]

Q3. The following figure shows that one local area network vendor provides a key distribution facility.

Key Distribution Center (KDC)
(2) ID_A, $E(K_a, N_a)$, ID_B, $E(K_b, N_b)$
(3) $E(K_b, [K_s, ID_A, N_b])$, $E(K_a, [K_s, ID_B, N_a])$
(1) A, $E(K_a, N_a)$
B
A
(4) $E(K_a, [K_s, ID_B, N_a])$

(i) Explain the key distribution scheme

The figure represents a **Key Distribution Center (KDC) based symmetric key distribution scheme**, similar to the **Needham–Schroeder Symmetric Key Protocol**.

There are three parties:

- **A** → Initiator
- **B** → Receiver
- **KDC** → Trusted Key Distribution Center

Each user shares a **long-term secret key** with the KDC:

- A shares K_A with KDC
- B shares K_B with KDC

The goal is to establish a **session key** K_s between A and B securely.

🔐 Step-by-Step Explanation of the Scheme

Step (1): A → KDC

A sends a request to the KDC:

$$ID_A, ID_B, N_A$$

- ID_A = Identity of A
- ID_B = Identity of B
- N_A = Nonce generated by A (to prevent replay attacks)

Purpose:

- A tells KDC that it wants to communicate with B.
- The nonce ensures freshness.

Step (2): KDC → A

KDC generates a **session key** K_s and sends:

$$E_{K_A}(K_s, ID_B, N_A,\ E_{K_B}(K_s, ID_A))$$

Encrypted using A's long-term key K_A.

Inside this message:

1. K_s → session key
2. ID_B → confirms intended receiver
3. N_A → proves freshness
4. Ticket for B → $E_{K_B}(K_s, ID_A)$

Purpose:

- Only A can decrypt because it's encrypted with K_A.
- A gets:
  - The session key
  - A "ticket" meant for B.

Step (3): A → B

A forwards B's ticket:

$$E_{K_B}(K_s, ID_A)$$

Since only B knows K_B, only B can decrypt this.

After decryption, B learns:

- Session key K_s
- Identity of A

Now both A and B share K_s.

Step (4): B → A (Mutual Authentication)

B sends:

$$E_{K_s}(N_B)$$

A decrypts it and returns:

$$E_{K_s}(f(N_B))$$

This proves:

- Both parties know the session key
- Mutual authentication is achieved

🔐 What This Scheme Achieves

Security Property	How It Is Achieved
Confidentiality	Session key encrypted with long-term keys
Authentication	Ticket contains identity
Freshness	Nonces N_A, N_B
Secure key delivery	KDC generates and distributes K_s

🧠 Why KDC is Needed

Without KDC:

- Every pair of users must share a key.
- For n users $\rightarrow \frac{n(n-1)}{2}$ keys required.

With KDC:

- Each user only shares **one long-term key** with KDC.
- KDC dynamically creates session keys.

📌 Important Points (Exam-Oriented)

- It is a **symmetric key distribution protocol**
- KDC is a **trusted third party**
- Uses **nonces** to prevent replay attacks
- Establishes a **temporary session key**
- Reduces key management complexity

If you want, I can also explain:

- The replay attack vulnerability in original Needham–Schroeder
- How Kerberos improves this scheme
- Compare KDC vs Diffie-Hellman key exchange

(ii) Highlight the advantages and disadvantages of key distribution scheme

The question refers to the **KDC-based symmetric key distribution scheme** (like Needham–Schroeder / Kerberos style).

Here are the **advantages and disadvantages** clearly for exam writing:

✅ Advantages of KDC-Based Key Distribution

1️⃣ Reduces Key Management Complexity

- Without KDC:
  $\frac{n(n-1)}{2}$ keys required for *n* users.
- With KDC:
  - Each user shares only **one long-term key** with KDC.
  - Total keys required = **n**.

✔️ Scalable for large networks.

2️⃣ Session Keys for Every Communication

- KDC generates fresh **session key** K_s for each session.
- Limits exposure if a session key is compromised.

✔️ Improves confidentiality.

3️⃣ Provides Mutual Authentication

- Use of nonces ensures:
  - Freshness
  - No replay attacks (partially)
- Both A and B prove knowledge of K_s.

4️⃣ Centralized Control

- Easy to:
  - Add new users
  - Revoke compromised users
  - Monitor authentication

✔️ Better administrative control.

5️⃣ Efficient (Uses Symmetric Cryptography)

- Uses symmetric encryption (faster than public key crypto).
- Suitable for LAN environments.

❌ Disadvantages of KDC-Based Key Distribution

1️⃣ Single Point of Failure

- If KDC fails → entire system stops.
- If KDC is compromised → attacker can:
    - Generate fake session keys
    - Decrypt all communications

⚠️ Major security risk.

2️⃣ Key Storage Risk at KDC

- KDC stores all long-term secret keys.
- If compromised, full network security collapses.

3️⃣ Scalability Bottleneck

- All key requests must pass through KDC.
- In large systems → performance bottleneck.

4️⃣ Vulnerable to Replay (Original Version)

- Basic Needham–Schroeder protocol had replay attacks.
- Requires timestamps or additional protection.

(Kerberos fixed this using timestamps.)

5️⃣ Trust Dependency

- Entire system security depends on trusting the KDC completely.

📝 Final Exam Summary (Short Form)

Advantages:

- Reduces number of keys required
- Provides fresh session keys
- Enables mutual authentication
- Easy centralized management
- Efficient (symmetric cryptography)

Disadvantages:

- Single point of failure
- High-value attack target
- Scalability issues
- Possible replay attacks
- Requires full trust in KDC

If you want, I can also provide a **5-mark perfect exam answer format** (compact writing style).