Secure Multi-Factor Authentication System

1. Introduction

In today's digital landscape, securing user accounts is of utmost importance. Traditional password-based authentication methods are vulnerable to brute force attacks, phishing, and credential leaks. To enhance security, Multi-Factor Authentication (MFA) is implemented using cryptographic techniques such as password hashing, salting, and Time-based One-Time Password (TOTP) generation.

This case study explores the design and implementation of an MFA system using **PBKDF2 for password hashing**, **TOTP for two-factor authentication**, and **Base32 encoding** for secure key storage.

---

2. Problem Statement

The increasing frequency of cyberattacks exposes the weaknesses of single-factor authentication. Attackers use credential stuffing, brute-force attacks, and phishing to gain unauthorized access. A robust authentication system with an additional layer of security is needed to protect user accounts.

Problem

Organizations and users require a **secure, scalable, and efficient authentication system** that:

1. **Prevents password compromise** through strong cryptographic hashing with salt.
2. **Verifies user identity beyond passwords** using Time-Based One-Time Passwords (TOTP) for two-factor authentication (2FA).
3. **Ensures secure storage and processing of credentials** while minimizing attack vectors such as brute-force and replay attacks.
4. **Tracks authentication attempts** to monitor unauthorized access attempts.

5. **Provides a seamless user experience** while maintaining security standards.

Proposed Solution

To address these challenges, the system will implement:

1. **Password Hashing with PBKDF2 & Salt** – Prevents password cracking and database leaks.
2. **TOTP-Based MFA** – Enhances security by requiring a temporary OTP generated every 30 seconds.
3. **Secure Random Salt Generation** – Ensures password uniqueness and defends against rainbow table attacks.
4. **Cryptographic Challenge Generation** – Prevents replay attacks during authentication.
5. **User Authentication Logging** – Tracks authentication attempts to detect unauthorized access.

Expected Outcome

The implementation of this authentication system will provide a **secure, scalable, and reliable** way for users to log in using **passwords and MFA**, reducing the risk of unauthorized access and ensuring data protection.

---

3. Objectives

Securely store user credentials using **PBKDF2 hashing and salting**.

- Generate a **TOTP secret** for multi-factor authentication.
- Validate authentication using **both password and TOTP**.
- Ensure **cryptographic security** through industry-standard algorithms.

---

4. Methodologies & Algorithms

4.1 Password Hashing and Salting

- **Algorithm Used:** PBKDF2 (Password-Based Key Derivation Function 2)
- **Process:**
  - A user's password is combined with a unique salt.
  - PBKDF2 applies multiple iterations (e.g., 100,000) to derive a cryptographic key.
  - The resulting hash is stored securely instead of the plaintext password.

PBKDF2 is a key derivation function that iteratively hashes a password with a cryptographic salt to produce a secure hash.

🔹 How It Works in Your Code

📌 Function: deriveKey(password, salt)
📌 Algorithm: PBKDF2 with SHA-1 or SHA-256
📌 Key Features:

- Uses 100,000 iterations (iterations = 100000)
- Key size = 256 bits (32 bytes)
- Uses CryptoJS.PBKDF2 for implementation

Code Implementation:

```
import CryptoJS from 'crypto-js';

export const deriveKey = (password: string, salt: string): string ≥ {
  const iterations = 100000;
  const keySize = 256;
  return CryptoJS.PBKDF2(password, salt, {
    keySize: keySize / 32,
    iterations,
  }).toString();
};
```

Visualization:

[User Password] + [Salt] –> [PBKDF2] –> [Hashed Password]

4.2 Salt Generation

Salting ensures that even if two users choose the same password, their hashes will be different.

🔹 How It Works in Your Code

📌 Function: generateSalt()
📌 Algorithm: Uses crypto.getRandomValues() (built-in Web Crypto API)
📌 Salt Length: 16 bytes (128 bits)
📌 Hex Encoding: Converts bytes into a hex string.

Code Implementation:

```
export const generateSalt = (): string ≥ {
  const array = new Uint8Array(16);
  crypto.getRandomValues(array);
  return Array.from(array, byte ≥ byte.toString(16).padStart(2, '0')).join('');
};
```

Visualization:

[Random Values] –> [Hexadecimal Encoding] –> [Salt]

4.3 TOTP Secret Generation

- A 20-byte cryptographically secure random key is generated.
- It is Base32 encoded for compatibility with authentication apps.

TOTP is a one-time password algorithm that generates a 6-digit code every 30 seconds, using:

- A shared secret key (base32-encoded)
- The current timestamp
- A cryptographic hash function (HMAC-SHA1 or HMAC-SHA256)

📌 Function: generateTOTPSecret()
📌 Algorithm: Uses HMAC-based OTP (RFC 6238)
📌 Key Length: 20 bytes (160 bits)
📌 Encoding: Base32 (RFC 4648) for compatibility with Google Authenticator

Code Implementation:

```
import base32Encode from 'base32-encode';

export const generateTOTPSecret = (): string ≥ {

  const array = new Uint8Array(20);

  crypto.getRandomValues(array);

  return base32Encode(array, 'RFC4648', { padding: false });

};
```

Visualization:

[Random Bytes] –> [Base32 Encoding] –> [TOTP Secret]

4.4 TOTP Verification

- A TOTP is generated using the stored secret and the current timestamp.
- The user-provided TOTP is compared to the generated one.

🔹 How TOTP is Verified

- 📌 Function: verifyTOTP(token, secret)
  📌 Algorithm: Uses totp-generator package to generate the expected TOTP and compare it with the user's input.

Code Implementation:

```
import totp from 'totp-generator';

export const verifyTOTP = (token: string, secret: string): boolean ≥ {

 try {

  const currentToken = totp(secret);

  return token === currentToken;

 } catch (error) {

  console.error('TOTP verification error:', error);

  return false;

 } };
```

Visualization:

[TOTP Secret] + [Current Time] –> [TOTP Algorithm] –> [Generated Code]

4.5 QR Code Generation for TOTP Setup

- The Base32-encoded secret is embedded in a URL that authentication apps can recognize.
- A QR code is generated from this URL for easy scanning.

5.6 Key Algorithms & Methodologies

Feature	Algorithm Used	Purpose
Password Hashing	**PBKDF2 with 100,000 iterations**	**Secure password storage**
Salt Generation	**crypto.getRandomValues() (16 bytes)**	**Prevents rainbow table attacks**
TOTP (2FA)	**HMAC-based OTP (HOTP/TOTP - RFC 6238)**	**Generates time-sensitive authentication codes**
TOTP Secret Encoding	**Base32 (RFC 4648)**	**Stores TOTP secret in a format compatible with Authenticator apps**
TOTP Verification	**totp-generator (HMAC-SHA1 or SHA256)**	**Compares entered OTP with expected OTP**

---

5. User Data Storage

5.1 User Model

A user object stores essential details like ID, username, hashed password, salt, TOTP secret, and role.

User Schema:

```
export interface User {
  id: string;
  username: string;
  passwordHash: string;
  salt: string;
  totpSecret?: string;
  isAdmin?: boolean;
  registeredAt: number;
}
```

5.2 Authentication Attempts Logging

- Tracks authentication attempts to detect anomalies.

Authentication Schema:

```
export interface AuthenticationAttempt {
  timestamp: number;
  success: boolean;
  method: 'password' | 'totp';
  userId: string;
}
```

---

6. Authentication Process

6.1 Registration Process

1. The user provides a username and password.

2. A salt is generated and used to hash the password.
3. A TOTP secret is generated and stored for MFA.

6.2 Login Process

1. The user enters their credentials.
2. If password verification succeeds, the system requests a TOTP.
3. The user enters the TOTP from their authenticator app.
4. The system validates the TOTP and grants access if correct.

Visualization:

[User Login] –> [Password Verification] –> [TOTP Request] –> [User Enters TOTP] –> [Verification Success]

---

7. Security Considerations

- **PBKDF2 with high iterations**: Protects against brute-force attacks.
- **Salting passwords**: Prevents dictionary and rainbow table attacks.
- **TOTP expiration**: Ensures time-sensitive authentication codes.
- **No plaintext storage of passwords**: Only hashes are stored.
- **Secure key generation**: Cryptographic methods prevent weak keys.

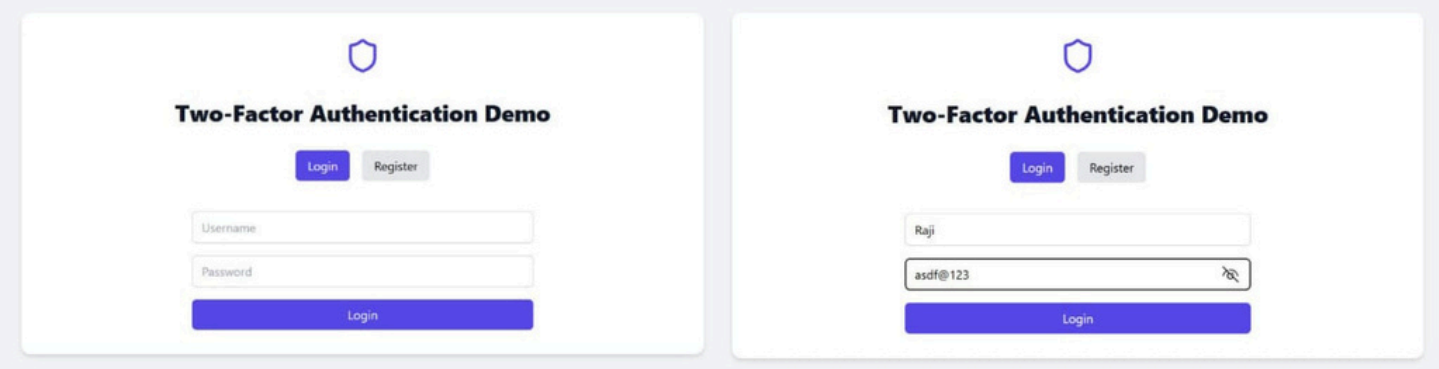
---

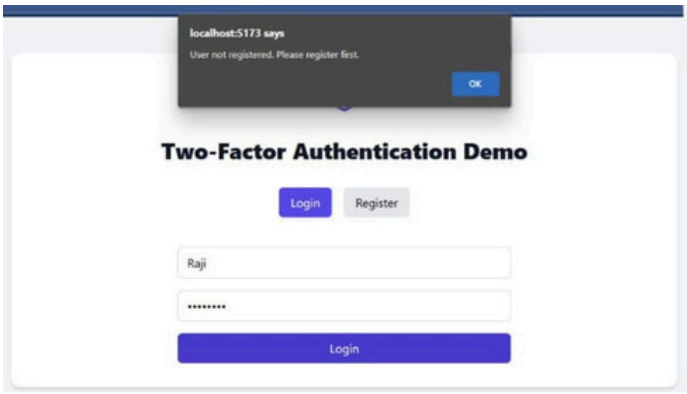
8. Conclusion

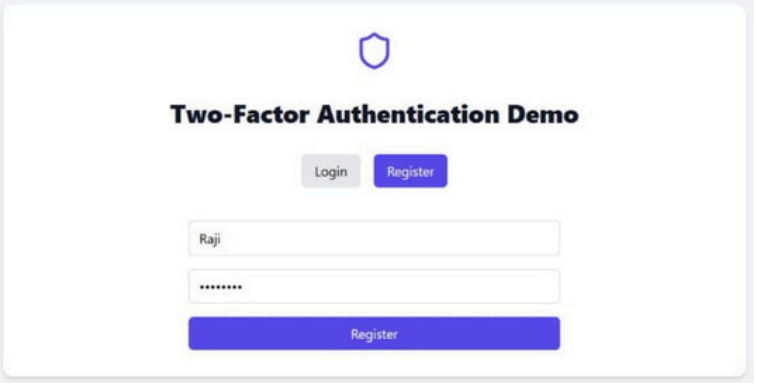
This case study demonstrates a secure MFA system using cryptographic principles. By combining password hashing, salting, and TOTP, the system significantly enhances authentication security. Future improvements will focus on increasing usability and security further.

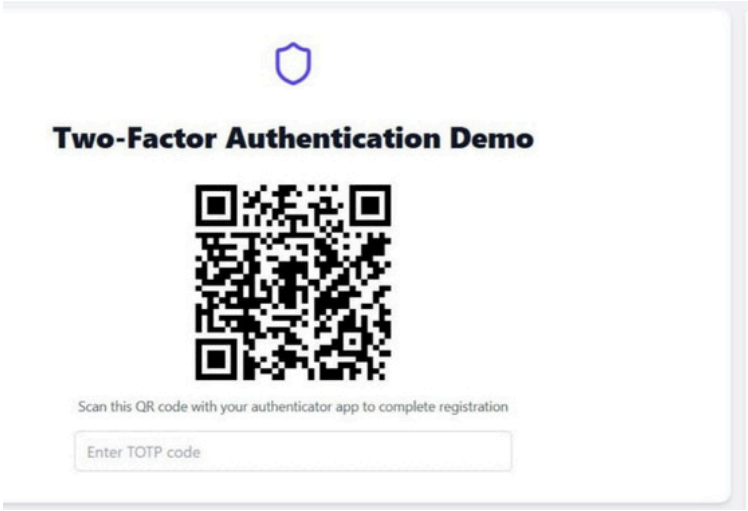
---

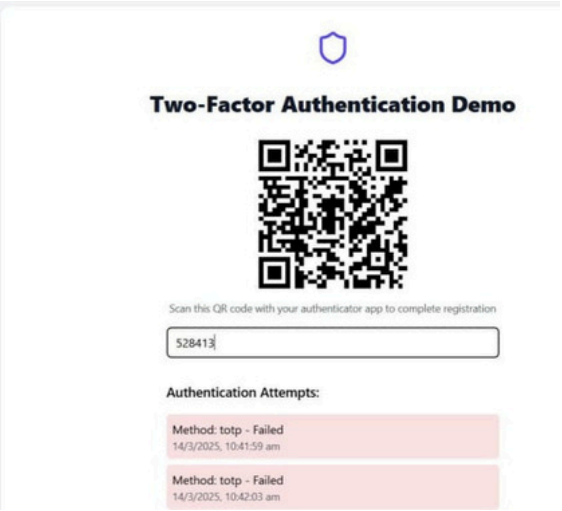
9. Output

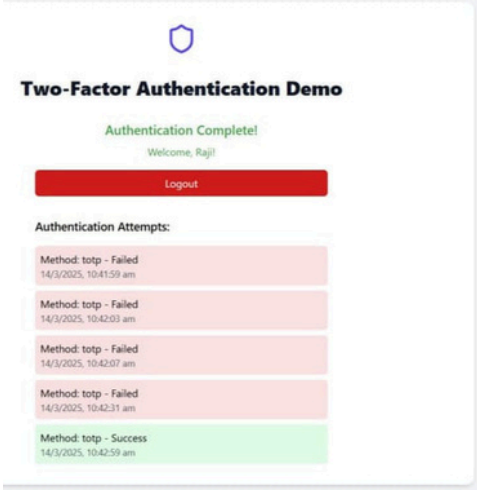


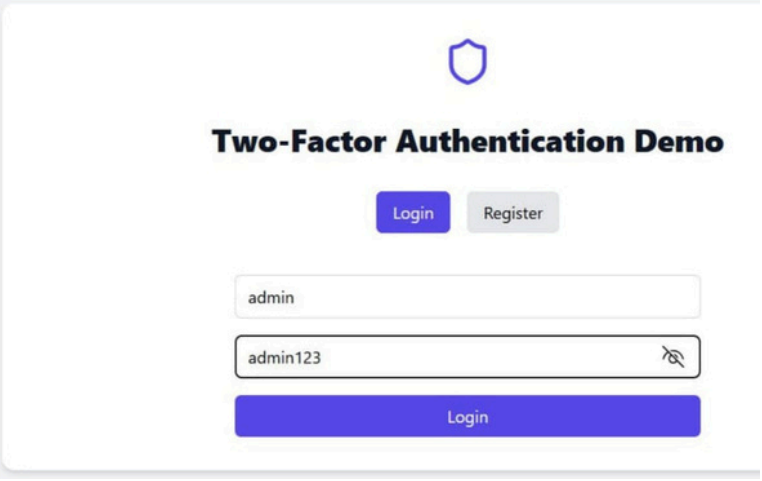

localhost:5173 says
User not registered. Please register first.
OK
Two-Factor Authentication Demo
Login
Register
Raji
Login

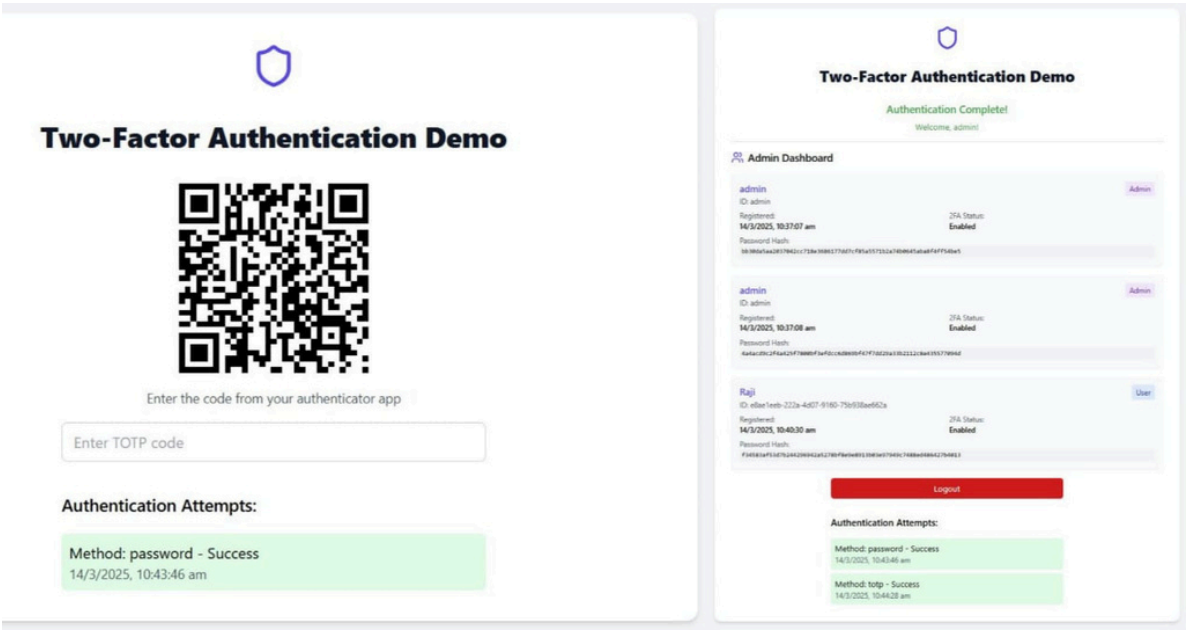

Two-Factor Authentication Demo
Login
Register
Raji
Register

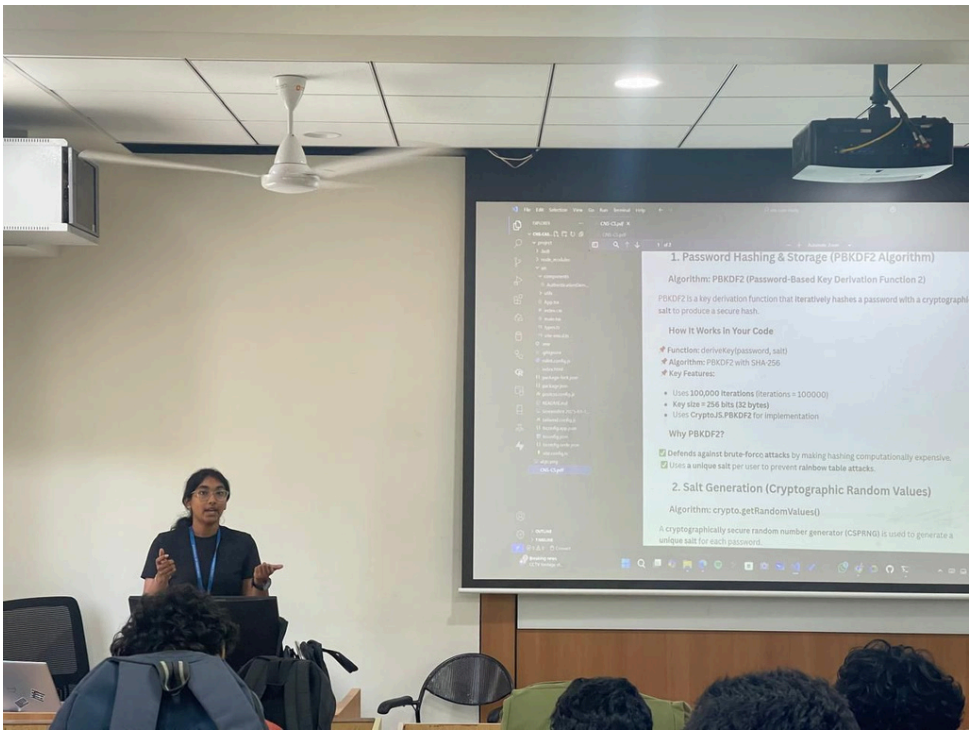

Two-Factor Authentication Demo
Scan this QR code with your authenticator app to complete registration
Enter TOTP code

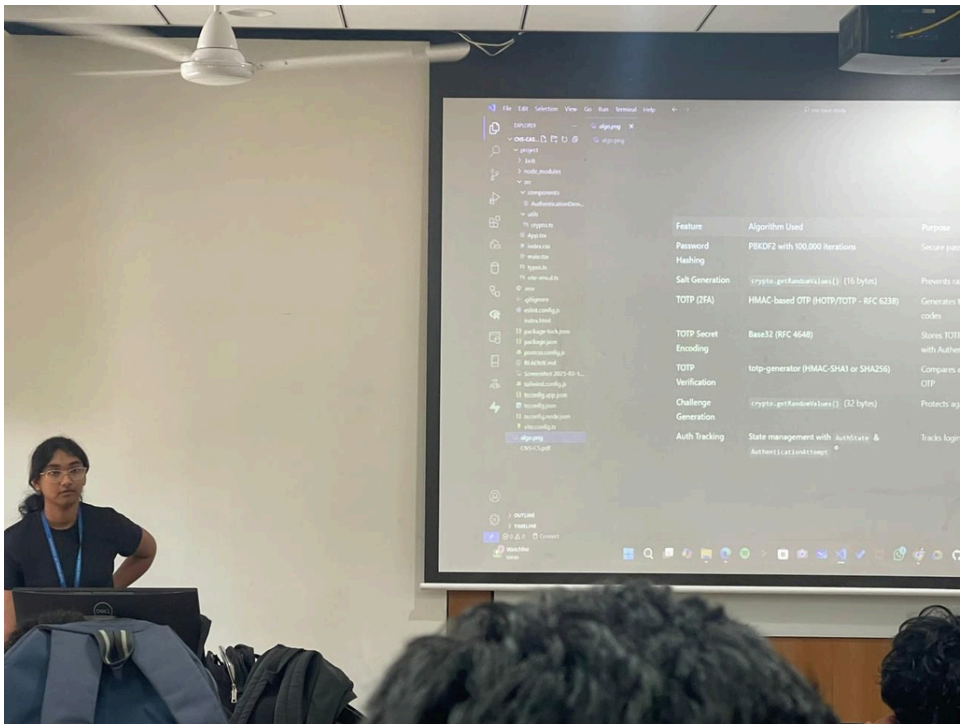

Two-Factor Authentication Demo
Scan this QR code with your authenticator app to complete registration
528413
Authentication Attempts:
Method: totp - Failed
14/3/2025, 10:41:59 am
Method: totp - Failed
14/3/2025, 10:42:03 am

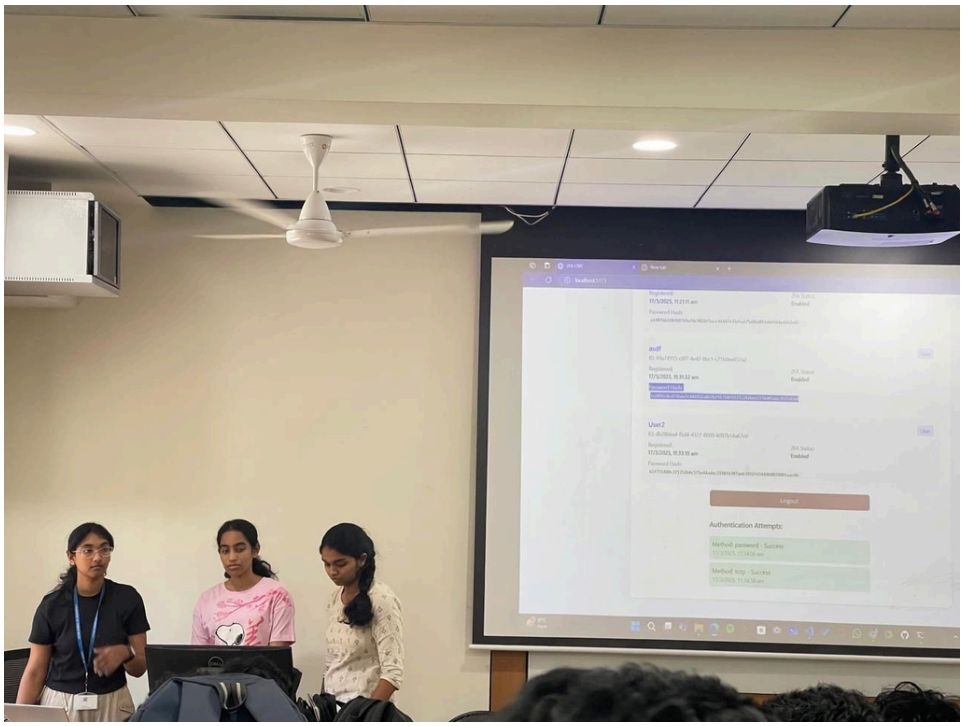

Two-Factor Authentication Demo
Authentication Complete!
Welcome, Raji!
Logout
Authentication Attempts:
Method: totp - Failed
14/3/2025, 10:41:59 am
Method: totp - Failed
14/3/2025, 10:42:03 am
Method: totp - Failed
14/3/2025, 10:42:07 am
Method: totp - Failed
14/3/2025, 10:42:31 am
Method: totp - Success
14/3/2025, 10:42:59 am


Two-Factor Authentication Demo
Login
Register
admin
admin123
Login


Two-Factor Authentication Demo
Enter the code from your authenticator app
Enter TOTP code
Authentication Attempts:
Method: password - Success
14/3/2025, 10:43:46 am
Two-Factor Authentication Demo
Authentication Complete!
Welcome, admin!
Admin Dashboard
admin
ID: admin
Registered:
14/3/2025, 10:37:07 am
2FA Status:
Enabled
Password Hash:
admin
ID: admin
Registered:
14/3/2025, 10:37:08 am
2FA Status:
Enabled
Password Hash:
Raji
ID: e8ae1eeb-222a-4d07-9160-75b938ae662a
Registered:
14/3/2025, 10:40:30 am
2FA Status:
Enabled
Password Hash:
Logout
Authentication Attempts:
Method: password - Success
14/3/2025, 10:43:46 am
Method: totp - Success
14/3/2025, 10:44:28 am


1. Password Hashing & Storage (PBKDF2 Algorithm)
Algorithm: PBKDF2 (Password-Based Key Derivation Function 2)
How It Works In Your Code
Function: deriveKey(password, salt)
Algorithm: PBKDF2 with SHA-256
Key Features:
Uses 100,000 iterations (iterations = 100000)
Key size = 256 bits (32 bytes)
Uses CryptoJS.PBKDF2 for implementation
Why PBKDF2?
Defends against brute-force attacks by making hashing computationally expensive.
Uses a unique salt per user to prevent rainbow table attacks.
2. Salt Generation (Cryptographic Random Values)
Algorithm: crypto.getRandomValues()





VU22CSEN0100432

Jyoshika Lalam

Presentation Date: 17-03-2025 - 11:30 AM

Thank You!