Algorithms & Methodologies Used in Your Code

Your code implements **multi-factor authentication (MFA)** using **password hashing with PBKDF2**, **TOTP-based 2FA**, and **cryptographic random values** for security. Below is a **detailed breakdown of the algorithms and methodologies used**.

🛠️ 1. Password Hashing & Storage (PBKDF2 Algorithm)

🔶 Algorithm: PBKDF2 (Password-Based Key Derivation Function 2)

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:**

```
export const deriveKey = (password: string, salt: string): string => {
  const iterations = 100000;
  const keySize = 256; // Key length in bits
  return CryptoJS.PBKDF2(password, salt, {
    keySize: keySize / 32,  // Convert bits to words
    iterations,
  }).toString();
};
```

🔹 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()

A **cryptographically secure random number generator (CSPRNG)** is used to generate a **unique salt** for each password.

🔹 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);  // 16 bytes = 128 bits

  crypto.getRandomValues(array);

  return Array.from(array, byte => byte.toString(16).padStart(2, '0')).join('');

};
```

🔹 Why Use a Salt?

✅ **Prevents precomputed attacks (Rainbow Table attacks)**
✅ **Ensures each password has a unique hash**, even if users choose the same password.

🛠️ 3. TOTP (Time-Based One-Time Password) for Two-Factor Authentication (2FA)

🔸 Algorithm: TOTP (Time-Based One-Time Password)

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:**

```
export const generateTOTPSecret = (): string => {
```

```
  const array = new Uint8Array(20);  // 20 bytes = 160 bits
  crypto.getRandomValues(array);
  return base32Encode(array, 'RFC4648', { padding: false });
};
```

🔹 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:**

```
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;
  }
};
```

🔹 Why TOTP for 2FA?

✅ **Time-based OTPs are more secure than static passwords**
✅ **Defends against password leaks**—an attacker needs both **password + TOTP** to log in.

🛠️ 4. Cryptographic Challenge Generation (Anti-Replay Protection)

🔸 Algorithm: Secure Random Challenge Generation

A cryptographic **challenge** is generated using **crypto.getRandomValues()** to prevent **replay attacks**.

📌 **Function:** generateChallenge()
📌 **Algorithm:** Generates a **32-byte (256-bit) random challenge**

📌 **Encoding:** Hexadecimal

📌 **Code Implementation:**

```
export const generateChallenge = (): string => {
  const array = new Uint8Array(32);  // 32 bytes = 256 bits
  crypto.getRandomValues(array);
  return Array.from(array, byte => byte.toString(16).padStart(2, '0')).join('');
};
```

🔹 Why Use Challenges?

✅ **Prevents replay attacks** in authentication workflows
✅ Can be used in **OAuth, FIDO2, and WebAuthn** for secure authentication

🛠️ 5. Authentication State Management (Auth Flow)

🔸 Methodology: Auth State Handling

An authentication system must **track user logins and failed attempts**.

📌 **Interface:** AuthState

```
export interface AuthState {
  isAuthenticated: boolean;
  currentUser: User | null;
  attempts: AuthenticationAttempt[];
}
```

📌 **Interface:** AuthenticationAttempt

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

🔹 Why Track Auth Attempts?

✅ Helps detect **brute-force attacks**
✅ Allows implementation of **account lockout policies**

🔹 Summary of 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
Challenge Generation	crypto.getRandomValues() (32 bytes)	Protects against replay attacks
Auth Tracking	State management with AuthState & AuthenticationAttempt	Tracks login attempts for security