Documentación técnica: Clase CertificateManager

Sistema de Gestión de Certificados - Proyecto Deasy

16 de junio de 2025

Índice

1. Introducción

El presente documento describe detalladamente el funcionamiento de la clase `CertificateManager`, desarrollada en JavaScript con Node.js y MongoDB. Esta clase permite el cifrado y almacenamiento seguro de archivos `.p12` (certificados digitales), así como su posterior recuperación y descifrado.

Se hace uso de las siguientes tecnologías y librerías:

- `mongoose`: ODM para MongoDB.
- `crypto`: Módulo nativo de Node.js para operaciones criptográficas.
- `fs-extra`: Extensión de `fs` para manejo de archivos.

2. Teoría del Cifrado Utilizado

El sistema implementado utiliza un esquema de cifrado simétrico, en el cual una misma clave es utilizada para cifrar y descifrar información. A continuación, se detallan los componentes criptográficos empleados en la implementación.

2.1. Derivación de Clave: PBKDF2 con SHA-256

Para convertir una contraseña provista por el usuario en una clave criptográfica segura de 256 bits, se utiliza el algoritmo **PBKDF2** (Password-Based Key Derivation Function 2), definido en la especificación `RFC 8018`.

- **Entrada**: contraseña (`password`) + valor aleatorio (`salt`).
- **Función hash**: `HMAC-SHA-256`.
- **Iteraciones**: 100,000 (mayor número implica mayor resistencia a ataques de fuerza bruta).
- **Salida**: clave derivada de 32 bytes (256 bits), adecuada para cifrado AES-256.

Ventajas del uso de PBKDF2:

- Introduce una sobrecarga computacional intencional que dificulta ataques de fuerza bruta.
- La utilización de un `salt` previene ataques con tablas arcoíris (rainbow tables).

2.2. Cifrado Simétrico: AES-256-CBC

Una vez derivada la clave criptográfica, se utiliza el algoritmo **AES (Advanced Encryption Standard)** en modo `CBC (Cipher Block Chaining)`.

Características del esquema AES-256-CBC:

- **Tamaño de clave**: 256 bits (alta seguridad).
- **Modo de operación**: CBC, donde cada bloque cifrado depende del bloque anterior, introduciendo aleatoriedad y propagación de errores intencional.
- **Inicialización**: requiere un vector de inicialización (`IV`) de 16 bytes (128 bits).

Importancia del IV:

- Debe ser único y aleatorio para cada operación de cifrado.
- No necesita ser secreto, pero sí debe almacenarse para poder descifrar.
- Asegura que dos archivos cifrados con la misma clave produzcan salidas distintas.

2.3. Almacenamiento Seguro en Base de Datos

El certificado cifrado, junto con los valores `salt` y `IV`, se almacena en una base de datos MongoDB. Estos valores permiten la reconstrucción segura del proceso de descifrado sin almacenar la clave ni la contraseña del usuario.

2.4. 4. Proceso de Descifrado

El descifrado se realiza de forma inversa:

1. Se recuperan el `salt` y el `IV` desde la base de datos.
2. Se deriva nuevamente la clave a partir de la contraseña proporcionada.
3. Se utiliza AES-256-CBC con la clave derivada y el IV para descifrar el contenido.

Nota importante: La seguridad completa del esquema depende de la confidencialidad de la contraseña del usuario. El sistema nunca almacena ni transmite esta contraseña, únicamente la utiliza en memoria volátil para derivar la clave.

2.5. 5. Resumen del Esquema Criptográfico

Componente	**Detalle**
Algoritmo de cifrado	AES-256 en modo CBC
Derivación de clave	PBKDF2 con HMAC-SHA-256, 100,000 iteraciones
Longitud de clave	256 bits (32 bytes)
Salt	16 bytes aleatorios, codificados en hexadecimal
IV (vector inicialización)	16 bytes aleatorios, codificados en hexadecimal
Almacenamiento	Certificado cifrado + salt + IV en MongoDB

2.6. 6. Seguridad del Esquema

El esquema es considerado seguro bajo las siguientes condiciones:

- El número de iteraciones de PBKDF2 es suficientemente alto.
- Las claves derivadas no se almacenan.
- Los valores de `salt` y `IV` son únicos por certificado.
- El acceso a la base de datos y al código está restringido.

3. Estructura del código

El código define una clase con dos métodos principales:

- `encryptAndStoreCertificate`: cifra un certificado con una clave derivada de una contraseña y lo guarda en MongoDB.
- `decryptAndRetrieveCertificate`: recupera un certificado cifrado de MongoDB y lo descifra.

4. Dependencias

```
npm install mongoose crypto fs-extra
```

5. Modelo de Mongoose

Se asume que existe un modelo llamado `Certificate`, que contiene al menos los siguientes campos:

- `userId`: Referencia al usuario.
- `filename`: Nombre del archivo original.
- `certificateData`: Certificado cifrado (Buffer).
- `encryptionSalt`: Salt usado para derivar la clave.
- `encryptionKey`: Vector de inicialización (IV) usado en AES.

6. Métodos de la clase CertificateManager

6.1. `deriveKey(password, salt)`

Deriva una clave criptográfica de 256 bits usando PBKDF2 con SHA-256 y 100.000 iteraciones.

```
static deriveKey(password, salt) {
  return new Promise((resolve, reject) => {
    crypto.pbkdf2(password, salt, 100000, 32, 'sha256', (err, derivedKey)
       => {
      if (err) reject(err);
      resolve(derivedKey);
    });
  });
}
```

6.2. `encryptAndStoreCertificate(filePath, password, userId)`

1. Verifica que el archivo y el `userId` sean válidos.
2. Genera un `salt` y un `iv`.
3. Deriva una clave con `deriveKey`.
4. Cifra el contenido del archivo usando `AES-256-CBC`.
5. Almacena el certificado cifrado en la base de datos.

```
static async encryptAndStoreCertificate(filePath, password, userId) {
  // Validaciones
  if (!fs.existsSync(filePath)) throw new Error('El archivo .p12 no existe
     ');
  if (!mongoose.Types.ObjectId.isValid(userId)) throw new Error('ID
     inv lido ');

  // Derivaci n de clave
  const salt = crypto.randomBytes(16);
  const derivedKey = await this.deriveKey(password, salt.toString('hex'));
  const iv = crypto.randomBytes(16);

  const cipher = crypto.createCipheriv('aes-256-cbc', derivedKey, iv);
  const fileBuffer = fs.readFileSync(filePath);
  let encrypted = Buffer.concat([cipher.update(fileBuffer), cipher.final()
     ]);

  const certificate = new Certificate({
    userId: new mongoose.Types.ObjectId(userId),
    filename: filePath.split('/').pop(),
    certificateData: encrypted,
    encryptionSalt: salt.toString('hex'),
    encryptionKey: iv.toString('hex'),
```

```
  });

  await certificate.save();
  console.log('Archivo .p12 cifrado y almacenado con  xito ');
}
```

6.3. decryptAndRetrieveCertificate(certificateId, password, outputPath)

1. Recupera el certificado de la base de datos.
2. Usa el `salt` y `IV` almacenados para derivar la clave.
3. Descifra los datos y los guarda como un nuevo archivo.

```
static async decryptAndRetrieveCertificate(certificateId, password,
   outputPath) {
  const certificate = await Certificate.findById(certificateId);
  if (!certificate) throw new Error('Certificado no encontrado');

  const derivedKey = await this.deriveKey(password, certificate.
     encryptionSalt);
  const iv = Buffer.from(certificate.encryptionKey, 'hex');
  const decipher = crypto.createDecipheriv('aes-256-cbc', derivedKey, iv);

  const decrypted = Buffer.concat([
    decipher.update(certificate.certificateData),
    decipher.final()
  ]);

  fs.writeFileSync(outputPath, decrypted);
  console.log('Archivo .p12 recuperado y descifrado con  xito ');
}
```

7. Conexión con MongoDB

```
await mongoose.connect("mongodb://localhost:27017/deasy")
console.log("Connected to MongoDB")
```

8. Ejemplo de uso

8.1. Cifrado

```
const filePath = '../my_cert.p12';
const password = 'usuarioPassword123';
const userId = '677703fc9db105315f75d5b2';

CertificateManager.encryptAndStoreCertificate(filePath, password, userId);
```

8.2. Descifrado

```
const certificateId = '679f9dcaddb246cd2347c6ce';
const outputPath = './myDecryptedCert.p12';
const password = 'usuarioPassword123';

CertificateManager.decryptAndRetrieveCertificate(certificateId, password,
   outputPath);
```

9. Consideraciones de seguridad

- El uso de `PBKDF2` y `AES-256-CBC` provee una buena seguridad para el almacenamiento de certificados.
- La contraseña del usuario nunca se almacena.
- Se recomienda proteger el acceso a MongoDB y los logs del servidor.
- El IV y salt deben mantenerse únicos por cada certificado.

10. Posibles mejoras

- Validación de longitud mínima de contraseña.
- Almacenamiento seguro de logs (sin mostrar IV o claves derivadas).
- Uso de `dotenv` para credenciales y configuración.
- Cifrado adicional de campos en MongoDB.