Correcciones de Seguridad a aplicar

Analista: Xavier Santilari Herrera
Conocimientos de lenguajes para la lectura recomendados: JavaScript y Python 💀

Análisis del Código

—

Notas

---

1. **Introducción: Objetivos del Informe**

- 1. Identificación y Evaluación de Prácticas de Seguridad Actuales: El primer objetivo de este análisis es identificar y evaluar las prácticas de seguridad que se han implementado en la aplicación web, como la validación de entradas, encriptación de contraseñas, manejo de tokens JWT, control de acceso basado en roles, entre otros. Esto incluye una revisión detallada de cómo estas prácticas se han integrado en el código de la aplicación, tanto en JavaScript como en Python.

- 2. Asegurar la Conformidad con los Estándares de Seguridad: Este análisis tiene como objetivo garantizar que el código de la aplicación cumpla con los estándares de seguridad reconocidos en la industria. Se buscará alinear las prácticas de codificación con las directrices de organizaciones de seguridad de TI de renombre, tales como OWASP (Open Web Application Security Project).

- 3. Identificar Vulnerabilidades y Áreas de Riesgo: Un componente esencial del análisis es la identificación de cualquier vulnerabilidad de seguridad en el código actual. Esto incluye la búsqueda de debilidades potenciales que podrían ser explotadas por ataques comunes, como inyecciones SQL, Cross-Site Scripting (XSS), y otros.

- 4. Proponer Mejoras y Soluciones: Basándose en los hallazgos, el análisis deberá proponer mejoras específicas y soluciones para fortalecer la seguridad de la aplicación.

Esto puede incluir la recomendación de prácticas de codificación más seguras, la implementación de nuevas herramientas de seguridad, o la modificación de las características existentes.

- 5. Creación de Documentación y Guías: Parte del análisis implica la creación de documentación detallada y guías de implementación, como las plantillas en JavaScript para el proceso de hash de contraseñas y los scripts de Python para la encriptación de datos, que sirven tanto para educar al equipo de desarrollo como para proporcionar un punto de referencia para futuras auditorías de seguridad.

- 6. Preparación para Auditorías de Seguridad Futuras: Este análisis también tiene como objetivo preparar el código y las prácticas de seguridad para auditorías de seguridad y pruebas de penetración más rigurosas, asegurando que la aplicación pueda resistir evaluaciones exhaustivas y mantenga su integridad y confiabilidad.

2. Metodología de Análisis : Herramientas y Técnicas Utilizadas

En el análisis de la seguridad y el código de nuestra aplicación web, se han empleado diversas herramientas y técnicas especializadas para garantizar un examen exhaustivo y preciso. A continuación, se describen las principales:

- Herramientas de Análisis Estático de Código (SAST): Se utilizó software de análisis estático para revisar el código fuente tanto de JavaScript como de Python. Esto incluyó herramientas como ESLint para JavaScript y Pylint para Python, que ayudaron a identificar patrones de codificación inseguros y posibles vulnerabilidades y una base del funcionamiento del código para la revisión más a fondo manual posterior.

- Herramientas de Análisis Dinámico de Aplicaciones (DAST): Se emplearon herramientas DAST (OWASP ZAP) para realizar pruebas en el entorno de ejecución de la aplicación. Estas pruebas fueron cruciales para identificar problemas de seguridad que solo se manifiestan durante la ejecución del código y compilar una lista para saber que modificaciones pedir a full stack en el código.

- Herramientas Front-End Específicas: React Developer Tools para aplicaciones React

- Herramientas para API y Servicios Web: Como Postman o Swagger para probar y validar APIs RESTful.

- Revisión Manual de Código: Además de las herramientas automáticas, se realizó una revisión manual detallada del código. Este enfoque permitió un análisis más profundo de la lógica de negocio y las implementaciones específicas de seguridad, como el manejo de contraseñas y tokens JWT. Así como una edición de este para incluir soluciones a los problemas encontrados durante el análisis.

- Herramientas de Análisis de Dependencias: Se utilizó software como npm audit y PyUp para examinar las dependencias utilizadas en el proyecto. Estas herramientas me ayudaron a identificar bibliotecas obsoletas o con vulnerabilidades conocidas. Por desgracia no pude generar un reporte concluyente debido al tiempo disponible una vez la app estaba desarrollada y completamente relacionada correctamente.

El uso combinado de estas herramientas y técnicas proporcionó una visión completa de la seguridad de la aplicación, asegurando que todos los aspectos, desde el código fuente hasta el comportamiento en tiempo de ejecución, fueran examinados minuciosamente y yo pudiese desarrollar las correcciones necesarias para mandar a full stack y cloud para su corrección. La opinión de la No aplicación de estas no está contemplada en esta guía.

3. Hallazgos de Seguridad: Detalles de las prácticas de seguridad actuales y Vulnerabilidades identificadas.

A pesar de las implementaciones de seguridad actuales (nótese la ironía), el análisis reveló varias vulnerabilidades críticas que necesitaban ser abordadas para garantizar la integridad y seguridad de la aplicación web. Estas incluyen:

- 1. Almacenamiento Inseguro de Contraseñas: Se identificó que las contraseñas se almacenaban en la base de datos sin la aplicación adecuada de hashing y salting. Esto representaba un riesgo significativo, ya que las contraseñas podrían ser comprometidas en caso de una brecha de datos. Se desarrolló un template en JavaScript utilizando bcrypt para solucionar este problema, implementando un método seguro de hashing con salting para las contraseñas de los usuarios.

- 2. Encriptación de Datos Sensibles: La aplicación no encriptaba los datos sensibles antes de almacenarlos en la base de datos. Se generó un script en Python para encriptar estos datos, utilizando el hash de la contraseña del usuario como clave de encriptación, aumentando así la seguridad de la información almacenada.

- 3. Vulnerabilidades en la Autenticación y Sesión: Se detectaron debilidades en el manejo de sesiones y autenticaciones, incluyendo la falta de protección contra ataques de fuerza bruta y la ausencia de mecanismos robustos para la gestión de tokens JWT. Estas vulnerabilidades fueron abordadas mediante la implementación de límites en los intentos de inicio de sesión y mejorando el manejo de JWT.

- 4. Falta de Validaciones en el Lado del Cliente: La aplicación carecía de validaciones de entrada en el lado del cliente, lo que podría facilitar ataques como Cross-Site Scripting (XSS) o inyecciones SQL. Se propuso la implementación de validaciones en el cliente para complementar las validaciones del lado del servidor.

- 5. Comunicaciones No Seguras: Se observó que partes de la aplicación no utilizaban HTTPS, lo que podría exponer las comunicaciones a interceptaciones y ataques man-in-the-middle. Se recomendó la implementación de HTTPS en estas para asegurar todas las comunicaciones entre el cliente y el servidor.

Además de las vulnerabilidades ya mencionadas y abordadas, se detectaron otras áreas críticas que aún requieren atención:

- 1. Falta de Protección contra CSRF (Cross-Site Request Forgery): Se identificó que la aplicación no tiene medidas implementadas para proteger contra ataques CSRF, especialmente en las áreas donde se utilizan cookies para la autenticación. Este tipo de ataque podría permitir a un atacante inducir a los usuarios a realizar acciones no deseadas en la aplicación haciéndose pasar por el servidor.

- 2. Insuficiente Registro y Monitoreo de Actividades: Se observó una carencia en el sistema de registro y monitoreo de actividades sospechosas o inusuales. Esta limitación dificulta la capacidad de detectar y responder a tiempo ante posibles incidentes de seguridad.

- 3. Configuración Inadecuada del .gitignore: Se encontró que el archivo .gitignore no incluye todas las entradas necesarias, lo que podría llevar a la exposición accidental de archivos y datos sensibles en los repositorios de código, como claves API, archivos de configuración, entre otros.

- 4. Vulnerabilidad a Path Traversal: La aplicación carece de protecciones adecuadas contra ataques de Path Traversal, lo que podría permitir a un atacante acceder a archivos y directorios almacenados fuera del directorio raíz web. Esta vulnerabilidad es particularmente crítica, ya que podría exponer información sensible del sistema.

4. Análisis, Mejoras y Recomendaciones: Ojo, código más adelante, apartar la vista cuando sea necesario para evitar daños 🏥

4.1 Hash y salt:

En el proceso de revisión y mejora de la seguridad de nuestra aplicación web, se identificó la necesidad de implementar un sistema robusto para el manejo de contraseñas. A continuación, se presenta una explicación detallada y un template de código en Node.js para la correcta aplicación del hash y salt en contraseñas, crucial para la seguridad de la información de usuario.

Nota: Dado los desconocimientos en la materia, creo unos plantillas y explicación para fullstack para la correcta aplicación y un conocimiento básico del hash y salt para contraseñas y cómo implementarlo con Node.js. Procedo a incorporar aqui como introducción a este infierno:

- Explicación sobre Hashing de Contraseñas:

El proceso de hashing es esencial en la seguridad de las contraseñas. Un hash es una función que transforma cualquier entrada, como una contraseña, en una cadena de caracteres de longitud fija y es unidireccional, lo que hace prácticamente imposible revertir el hash para obtener la contraseña original. Al almacenar contraseñas, lo que realmente guardamos es su versión hasheada, lo que impide que un atacante pueda descifrarlas fácilmente en caso de acceso a nuestra base de datos.

El hashing asegura que incluso un cambio mínimo en la entrada resulte en un hash completamente diferente. Así, dos contraseñas similares tendrán hashes distintos. Esto aumenta la seguridad, pero aquí es donde el "salt" entra en juego para añadir aún más protección.

- El papel del "Salt" en el Hashing:

El concepto de "salt" en el contexto de la seguridad de contraseñas es un elemento crucial para proteger las contraseñas almacenadas. Imagina que tienes una contraseña simple, como "1234". Esta contraseña, por ser tan común, puede estar fácilmente en una tabla precalculada (llamada tabla rainbow) que los hackers usan para descifrar hashes de contraseñas. Pero aquí es donde el "salt" marca la diferencia.

Un "salt" es básicamente una cadena de caracteres aleatorios que se añade a la contraseña antes de aplicar el proceso de hash. Al añadir un "salt" único a cada contraseña antes de hashearla,

incluso contraseñas idénticas resultarán en hashes diferentes. Esto significa que, aunque dos usuarios tengan la misma contraseña, sus hashes almacenados serán diferentes debido a los diferentes "salts" utilizados. El "salt" transforma una contraseña común en algo único y mucho más difícil de descifrar.(si, los hashes se pueden descifrar, sorpresa)

ej practico pass 1234:

Cada vez que hashes "1234" sin un salt, obtendrás el mismo hash. (se puede revertir con tabla rainbow)

aplicamos un salt:

Generamos un salt aleatorio. Por ejemplo: "aB1!"

Añadimos el salt a la contraseña: "1234" + "aB1!" = "1234aB1!"

Aplicamos la función de hash a la combinación de contraseña y salt.

Hash (con salt): [Hash de 1234aB1!]

Hash(sin salt): [Hash de [[Hash de 1234aB1!]]

...

hasta hacer esto 1024 veces para factor de trabajo 10 (edited)

bcrypt.genSalt(10); para otro user con la misma pass 1234 como la salt es aleatoriamente generada será un hash diferente

Implementación en Node.js con bcrypt

Para reforzar la seguridad en nuestra aplicación, desarrollé un template en Node.js utilizando bcrypt. La implementación de bcrypt en Node.js sigue este proceso:

Funciones para el template separadas:

- Instalación de bcrypt(bash):

```
npm install bcrypt
```

- Importar bcrypt:

```
const bcrypt = require('bcrypt');
```

- Generación de Salt:

```
const salt = await bcrypt.genSalt(10);
```

- Hash de la Contraseña:

```
const hashedPassword = await bcrypt.hash('myPlaintextPassword', salt);
```

- Verificación de la Contraseña:

```
const isValidPassword = await bcrypt.compare('enteredPassword', hashedPassword);
```

Generé un template completo para poder aplciar ellos.
Esta implementación en Node.js con bcrypt representa una mejora significativa en la manera en que manejamos la seguridad de las contraseñas en nuestra aplicación. Proporciona un método seguro y eficiente para el almacenamiento y verificación de contraseñas, protegiendo así la información de usuario contra accesos no autorizados y posibles brechas de seguridad.
Template:

```
async function hashPassword(password) {
  const salt = await bcrypt.genSalt(10);
  const hashedPassword = await bcrypt.hash(password, salt);
  return hashedPassword;
}

async function checkPassword(password, hashedPassword) {
  const isValidPassword = await bcrypt.compare(password, hashedPassword);
  return isValidPassword;
}
```

4.2 Encrypt:

Encyptado de la información sensible de los usuarios antes de ser almacenada en la base de datos para cumplir con la legislación del Reglamento General de Protección de Datos (GDPR)

Introducción:

La seguridad de la información en bases de datos es una preocupación central en el desarrollo de aplicaciones. La necesidad de proteger datos sensibles, como contraseñas y otra información confidencial, ha llevado a la implementación de técnicas avanzadas, como el cifrado. El código proporcionado utiliza el módulo 'crypto' de Node.js y 'mysql' para aplicar cifrado de datos en una base de datos MySQL, fortaleciendo la seguridad y protegiendo la confidencialidad de la información almacenada.

Motivo de uso:

La principal razón para aplicar este código es la protección de datos sensibles almacenados en una base de datos. El cifrado se utiliza para convertir información legible en una forma ilegible, y solo aquellos con la clave correcta pueden revertir este proceso. En el código, se utiliza el cifrado simétrico AES-256-CBC, que es ampliamente reconocido por su robustez y seguridad. Además, se emplea una técnica conocida como "salting", que añade una capa adicional de seguridad al generar datos únicos (la "salt") para cada conjunto de datos. Este cifrado se aplica antes de poner la información en la base de datos SQL y también en el código se implementa la funcionalidad para decodificar la información con el hash de la contraseña del usuario logueado.

Código:

Welcome | Database_Encrypt.js | DockerScript.sh

Code2Fix > Database_Encrypt.js > ...

```
const mysql = require('mysql');
const crypto = require('crypto');

// Función para generar una sal aleatoria para el cifrado
function generateSalt(length) {
    return crypto.randomBytes(Math.ceil(length/2))
                 .toString('hex') // Convertir a formato hexadecimal
                 .slice(0,length); // Devolver la cantidad requerida de caracteres
}

// Función para cifrar los datos
function encrypt(text, password) {
    const salt = generateSalt(16); // Generar una sal
    const key = crypto.scryptSync(password, salt, 32); // Generar una clave usando la contraseña y la sal
    const iv = crypto.randomBytes(16); // Generar un vector de inicialización
    const cipher = crypto.createCipheriv('aes-256-cbc', key, iv); // Crear un cifrador usando la clave y el vector de inicialización

    let encrypted = cipher.update(text, 'utf8', 'hex'); // Cifrar el texto
    encrypted += cipher.final('hex'); // Finalizar el cifrado
    return { salt, iv: iv.toString('hex'), encrypted }; // Devolver la sal, el vector de inicialización y los datos cifrados
}
```

```
// Función para descifrar los datos
function decrypt(encryptedText, password, salt, iv) {
    const key = crypto.scryptSync(password, salt, 32); // Generar la clave usando la contraseña y la sal
    const decipher = crypto.createDecipheriv('aes-256-cbc', key, Buffer.from(iv, 'hex')); // Crear un descifrador usando la clave y el vector de inicialización

    let decrypted = decipher.update(encryptedText, 'hex', 'utf8'); // Descifrar el texto
    decrypted += decipher.final('utf8'); // Finalizar el descifrado
    return decrypted; // Devolver los datos descifrados
}

// Crear una conexión a la base de datos
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'tu_usuario',
  password: 'tu_contraseña',
  database: 'tu_base_de_datos'
});
```

```
connection.connect();

// Cifrar los datos y guardarlos en la base de datos
const password = 'clave secreta';
const data = 'datos a encriptar';
const encryptedData = encrypt(data, password);

connection.query('INSERT INTO tu_tabla (salt, iv, data) VALUES (?, ?, ?)', [encryptedData.salt, encryptedData.iv, encryptedData.encrypted], function (error, results, fields) {
  if (error) throw error;
  console.log('Datos encriptados guardados en la base de datos');
});

// Recuperar los datos de la base de datos y descifrarlos
connection.query('SELECT * FROM tu_tabla WHERE id = ?', [1], function (error, results, fields) {
  if (error) throw error;
  const decryptedData = decrypt(results[0].data, password, results[0].salt, results[0].iv);
  console.log('Datos desencriptados:', decryptedData);
});

connection.end();
```

Funciones clave en el código:

generateSalt(length):

Genera una cadena de caracteres aleatorios, llamada "sal", que se utiliza como entrada adicional en la derivación de la clave de cifrado. Esta sal única mejora la seguridad al prevenir ataques de fuerza bruta.

encrypt(text, password):

Cifra los datos utilizando el algoritmo AES-256-CBC. La función genera una salt aleatoria, crea una clave de cifrado mediante la derivación de la contraseña y la sal, y luego utiliza esta clave junto con un vector de inicialización (IV) único para cifrar los datos.

decrypt(encryptedText, password, salt, iv):

Descifra los datos utilizando el mismo algoritmo AES-256-CBC. La función utiliza la contraseña, la sal y el IV proporcionados para generar la clave de descifrado y luego aplica esta clave para revertir el cifrado y obtener los datos originales.

Funcionamiento del código:

Este código Node.js tiene como objetivo implementar un sistema seguro de cifrado y descifrado para datos sensibles almacenados en una base de datos MySQL. Utiliza el algoritmo de cifrado simétrico AES-256-CBC junto con el concepto de "salting" para fortalecer la seguridad de la información.

En primer lugar, se genera una salt aleatoria mediante la función generateSalt que utiliza el módulo 'crypto'. Esta sal actúa como un componente único en la generación de claves de cifrado, proporcionando una capa adicional de seguridad.

La función encrypt toma un texto y una contraseña como entrada, generando dinámicamente una "sal", una clave y un vector de inicialización (IV). Luego, utiliza estos elementos para cifrar el texto mediante el algoritmo AES-256-CBC. La salida de esta función es un objeto que contiene la "sal", el IV y los datos cifrados.

Por otro lado, la función decrypt toma el texto cifrado, la contraseña, la salt y el IV como parámetros. Genera la clave de descifrado y utiliza el algoritmo AES-256-CBC para revertir el cifrado, recuperando así los datos originales.

El código establece una conexión a la base de datos MySQL, cifra los datos sensibles utilizando la función encrypt, y luego inserta estos datos, junto con la salt y el IV, en la base de datos. Posteriormente, realiza una consulta para recuperar los datos cifrados, utilizando la función decrypt para desencriptarlos y mostrarlos en la consola.

Este enfoque integral de cifrado y descifrado garantiza una mayor seguridad en la gestión de datos sensibles, minimizando los riesgos asociados con el acceso no autorizado a la base de datos. La atención especial a la protección de la contraseña utilizada para derivar las claves es esencial para mantener la integridad del sistema de cifrado.

4.3 Análisis Código vulnerable Javascript (página index, aka: login)

(Versión resumida del análisis por falta de tiempo con comentarios sin editar, disculpen las molestias)

En este apartado me voy a centrar en los archivos que conforman el índice con el login de la página. Concretamente login.controller.js y users.controller.js

Login.controller.js original:

```
const usersModels = require('../models/users.model');
const { createToken } = require('../config/jsonWebToken');
var bcrypt = require('bcryptjs');
const { validationResult } = require('express-validator');

const login = async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }

    try {
        const { email, password } = req.body;
        const user = await usersModels.findOne({ where: { email: email} });
        if(user){
            bcrypt.compare(password, user.password).then((result)=>{
                if (result) {
                    const token = createToken({ email: user.email, role: user.role });
                    res.status(200)
                        .cookie('access_token', token, { secure: true, httpOnly: true })
                        .json({ role: user.role, cookie: token });
                } else {
                    res.status(400).json({ msg: "wrong credentials invalid password" });
                }
            }).catch((error) => {
                res.status(500).json({ msg: "Internal server error" });
            });
        }else{
            res.status(400).json({ msg: "wrong credentials user not found" });
        }

    } catch (error) {
        res.status(500).json({ msg: "Internal server error" });
    }
};

const logout = async (req, res) => {
    try {
        res.status(200)
            .cookie('access_token', "", { secure: true, httpOnly: true })
            .send();
    } catch (error) {
        res.status(500).json({ msg: "Internal server error" });
    }
};


const getAllUsers = async (req, res) => {
    try {
        const users = await usersModels.getAllUsers();
        console.log(users);
        res.status(200).json(users);
    } catch (error) {
        res.status(400).json({ msg: error.message })
    }
}


const loginController = {
    login,
    logout,
    getAllUsers
};
```

Despues del analisis del middleware saco las conclusiones:

Buenas Prácticas Implementadas en `login.controller.js`

Validación de Entradas

- **Uso de `validationResult` de `express-validator`**: Esto indica que las entradas están siendo validadas correctamente, lo cual es una buena práctica en el desarrollo web.

Manejo de Contraseñas

- **Uso de `bcrypt` para comparar contraseñas**: Indica un manejo seguro de las contraseñas, clave para la seguridad de la aplicación.

Cookies Seguras

- **Configuración de Cookies**: Al establecer opciones como `secure: true` y `httpOnly: true`, se sigue una práctica de seguridad recomendada para el manejo de cookies.

Manejo de Errores

- **Devolución de Mensajes de Error**: Se observa un manejo adecuado de errores, aunque se recomienda revisar el contenido de estos mensajes para evitar revelar información sensible.

2DO:

Aspectos No Implementados o No Evidentes

- **Encriptación de Contraseñas en el Registro**: Es necesario encriptar las contraseñas antes de almacenarlas en la base de datos. (no me acuerdo que querian hacer los de full tbh, need sleep)

- **Mensajes de Error Genéricos en Autenticación**: Usar mensajes de error más genéricos para evitar revelar información sobre cuentas de usuario.

- **Prevención de Ataques de Fuerza Bruta**: Implementar medidas para limitar los intentos de inicio de sesión fallidos.

- **HTTPS en Todo el Sitio**: Es importante que todas las comunicaciones entre el cliente y el servidor se realicen a través de HTTPS.

- **Auditoría de Seguridad y Pruebas**: pentest victor pendiente y post revision de código.

- **Revisión del Uso de JWT y Cookies**: Hay que asegurarse de que los tokens JWT se manejen de manera segura, tanto en almacenamiento como en transmisión.

- **Protección contra CSRF (Cross-Site Request Forgery)**: Si se utilizan cookies para la autenticación, es vital implementar medidas contra ataques CSRF.

- **Validaciones en el Lado del Cliente**: Es importante realizar validaciones de entrada en el lado del cliente para mejorar la seguridad.

- **Registro y Monitoreo de Actividades**: Se recomienda implementar un sistema de registro para monitorear actividades sospechosas o inusuales.

Errores Genericos en login.controller:

Nota: si el pass y el user invalidos tienen el mismo error es mas dificil crear tablas de usuario por diccionario.
Arreglo:

Original:

```
if (user) {
    bcrypt.compare(password, user.password).then((result) => {
        if (result) {
            // ...
        } else {
            res.status(400).json({ msg: "wrong credentials invalid password" });
        }
    }).catch((error) => {
        res.status(500).json({ msg: "Internal server error" });
    });
} else {
    res.status(400).json({ msg: "wrong credentials user not found" });
}
```

(edited)

Cambiar por:

```
if (user) {
    bcrypt.compare(password, user.password).then((result) => {
        if (result) {
            // ...
        } else {
            res.status(401).json({ msg: "Invalid credentials" });
        }
    }).catch((error) => {
        res.status(500).json({ msg: "Internal server error" });
    });
} else {
    res.status(401).json({ msg: "Invalid credentials" });
}
```

Mas por hacer:

Registrar Intentos Fallidos: Podríamos mantener un registro de los intentos fallidos de inicio de sesión para cada usuario. Esto podría hacerse usando una base de datos o una caché en memoria (como Redis).

Bloquear Temporalmente Después de Intentos Fallidos: Después de un cierto número de intentos fallidos (por ejemplo, 5), bloquear temporalmente al usuario de intentar iniciar sesión durante un período de tiempo (por ejemplo, 15 minutos).

Modificar login.controller.js: agregar la lógica para registrar y verificar los intentos fallidos en la función login

Plantilla:

template: (OJO! ya he modificado los mensajes de error para que coincidan, ver punto anterior)

```
// Ejemplo del concepto, se necesita una implementación real de registro de intentos
const loginAttempts = {};

const login = async (req, res) => {
    // ...

    try {
        const { email, password } = req.body;
        // Verificar si el usuario está temporalmente bloqueado
        if (loginAttempts[email] && loginAttempts[email].blockedUntil > Date.now()) {
            return res.status(429).json({ msg: "Too many failed attempts. Try again later. Go away to think about what you did. (lol, users these days are getting dumber by the minute)" });
        }

        const user = await usersModels.findOne({ where: { email: email } });
        if (user) {
            bcrypt.compare(password, user.password).then((result) => {
                if (result) {
                    // Restablecer los intentos fallidos
                    loginAttempts[email] = null;
                    // ...
                } else {
                    // Registrar intento fallido
                    if (!loginAttempts[email]) {
                        loginAttempts[email] = { attempts: 1, blockedUntil: null };
                    } else {
                        loginAttempts[email].attempts++;
                        if (loginAttempts[email].attempts >= 5) {
                            // Bloquear por 15 minutos
                            loginAttempts[email].blockedUntil = Date.now() + 15 * 60 * 1000;
                        }
                    }
                    res.status(401).json({ msg: "Invalid credentials" });
                }
            });
        } else {
            res.status(401).json({ msg: "Invalid credentials" });
        }
    } catch (error) {
        // ...
    }
};
```

Implementación (con comentarios para no perderse 😀):

función login modificada para incluir el control de intentos fallidos:

```
const login = async (req, res) => {
    // ... [Resto del código inicial, incluyendo la validación de entradas]

    try {
        const { email, password } = req.body;

        // Verificar si el usuario está temporalmente bloqueado
        if (loginAttempts[email] && loginAttempts[email].blockedUntil > Date.now()) {
            return res.status(429).json({ msg: "Too many failed attempts. Try again later. Go away to think about what you did. (lol, users these days are getting dumber by the minute)" });
        }

        const user = await usersModels.findOne({ where: { email: email } });
        if (user) {
            bcrypt.compare(password, user.password).then((result) => {
                if (result) {
                    // Restablecer los intentos fallidos en caso de éxito
                    loginAttempts[email] = null;

                    const token = createToken({ email: user.email, role: user.role });
                    res.status(200)
                        .cookie('access_token', token, { secure: true, httpOnly: true })
                        .json({ role: user.role, cookie: token });
                } else {
                    // Registrar intento fallido
                    registerFailedAttempt(email);
                    res.status(401).json({ msg: "Invalid credentials" });
                }
            });
        } else {
            // Registrar intento fallido
            registerFailedAttempt(email);
            res.status(401).json({ msg: "Invalid credentials" });
        }
    } catch (error) {
        res.status(500).json({ msg: "Internal server error" });
    }
};

// Objeto para almacenar los intentos fallidos
const loginAttempts = {};

// Función para registrar un intento fallido
function registerFailedAttempt(email) {
    if (!loginAttempts[email]) {
        loginAttempts[email] = { attempts: 1, blockedUntil: null };
    } else {
        loginAttempts[email].attempts++;
        if (loginAttempts[email].attempts >= 5) { // Número máximo de intentos
            // Bloquear por 15 minutos
            loginAttempts[email].blockedUntil = Date.now() + 15 * 60 * 1000;
        }
    }
}
```

Notas:

No arreglar:la info de sesiones se pierde si se reinicia la app porque patatas o otro motivo ya que lo guardo en una variable en memoria (loginAttempts).

Vulne porque programo como el culo: La implementación actual bloquea el inicio de sesión basándose en el correo electrónico proporcionado, incluso si el correo electrónico no está asociado a ninguna cuenta. Esto podría permitir a un atacante bloquear el inicio de sesión para cualquier correo electrónico simplemente realizando suficientes intentos fallidos. Para evitar esto, registrar los intentos fallidos solo después de verificar que el correo electrónico proporcionado está asociado a una cuenta existente.

Limpieza de datos: Actualmente, los datos de los intentos de inicio de sesión fallidos se mantienen en loginAttempts indefinidamente (a menos que un usuario inicie sesión con éxito).

Template para la limpieza:

```
const login = async (req, res) => {
    // ...

    try {
        const { email, password } = req.body;

        // ...

        const user = await usersModels.findOne({ where: { email: email } });
        if (user) {
            bcrypt.compare(password, user.password).then((result) => {
                if (result) {
                    // Restablecer los intentos fallidos
                    delete loginAttempts[email];  // Modificado aquí

                    // ...

                } else {
                    // ...

                }
            }).catch((error) => {
                res.status(500).json({ msg: "Internal server error" });
            });
        } else {
            res.status(401).json({ msg: "Invalid credentials" });
        }

    } catch (error) {
        res.status(500).json({ msg: "Internal server error" });
    }
};
```

(edited)

Implementacion y arreglo de vulnerabilidades generadas:

```
const usersModels = require('../models/users.model');
const { createToken } = require('../config/jsonWebToken');
var bcrypt = require('bcryptjs');
const { validationResult } = require('express-validator');

// Ejemplo hipotético, se necesita una implementación real de registro de intentos
const loginAttempts = {};

const login = async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }

    try {
        const { email, password } = req.body;

        // Verificar si el usuario está temporalmente bloqueado
        if (loginAttempts[email] && loginAttempts[email].blockedUntil > Date.now()) {
            return res.status(429).json({ msg: "Too many failed attempts. Try again later. Go away to think about what you did. (lol, users these days are getting dumber by the minute)" });
        }

        const user = await usersModels.findOne({ where: { email: email } });
        if (user) {
            bcrypt.compare(password, user.password).then((result) => {
                if (result) {
                    // Restablecer los intentos fallidos version 2.0
                    delete loginAttempts[email];

                    const token = createToken({ email: user.email, role: user.role });
                    res.status(200)
                        .cookie('access_token', token, { secure: true, httpOnly: true })
                        .json({ role: user.role, cookie: token });
                } else {
                    // Registrar intento fallido
                    if (!loginAttempts[email]) {
                        loginAttempts[email] = { attempts: 1, blockedUntil: null };
                    } else {
                        loginAttempts[email].attempts++;
                        if (loginAttempts[email].attempts >= 5) {
                            // Bloquear por 15 minutos
                            loginAttempts[email].blockedUntil = Date.now() + 15 * 60 * 1000;
                        }
                    }
                    res.status(401).json({ msg: "Invalid credentials" });
                }
            }).catch((error) => {
                res.status(500).json({ msg: "Internal server error" });
            });
        } else {
            res.status(401).json({ msg: "Invalid credentials" });
        }

    } catch (error) {
        res.status(500).json({ msg: "Internal server error" });
    }
};
```

```
const logout = async (req, res) => {
    try {
        res.status(200)
            .cookie('access_token', "", { secure: true, httpOnly: true })
            .send();
    } catch (error) {
        res.status(500).json({ msg: "Internal server error" });
    }
};

const getAllUsers = async (req, res) => {
    try {
        const users = await usersModels.getAllUsers();
        console.log(users);
        res.status(200).json(users);
    } catch (error) {
        res.status(400).json({ msg: error.message })
    }
}

const loginController = {
    login,
    logout,
    getAllUsers
};

module.exports = loginController;
```

Template arreglo vulnerabilidad:

arreglando vulnerabilidad de bloqueo de cuentas por no verificar si el correo es valido o no antes de introducir el log de error:

```
const login = async (req, res) => {
    // ...

    try {
        const { email, password } = req.body;

        // ...

        const user = await usersModels.findOne({ where: { email: email } });
        if (user) {
            bcrypt.compare(password, user.password).then((result) => {
                if (result) {
                    // Restablecer los intentos fallidos
                    delete loginAttempts[email];

                    // ...

                } else {
                    // Registrar intento fallido
                    if (!loginAttempts[email]) {
                        loginAttempts[email] = { attempts: 1, blockedUntil: null };
                    } else {
                        loginAttempts[email].attempts++;
                        if (loginAttempts[email].attempts >= 5) {
                            // Bloquear por 15 minutos
                            loginAttempts[email].blockedUntil = Date.now() + 15 * 60 * 1000;
                        }
                    }
                    res.status(401).json({ msg: "Invalid credentials" });
                }
            }).catch((error) => {
                res.status(500).json({ msg: "Internal server error" });
            });
        } else {
            res.status(401).json({ msg: "Invalid credentials" });
        }

    } catch (error) {
        res.status(500).json({ msg: "Internal server error" });
    }
};
```

En este punto tengo que aplicar un merge con los push aplicados por full stack al proceso de seguridad que tengo hasta ahora:

Nuevo código para el controlador(con los push nuevos):

```
const usersModels = require('../models/users.model');
const { createToken } = require('../config/jsonWebToken');
let bcrypt = require('bcryptjs');
const { validationResult } = require('express-validator');

const login = async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }

    try {
        const { email, password } = req.body;
        const user = await usersModels.findOne({ where: { email: email} });
        if(user){
            bcrypt.compare(password, user.password).then((result)=>{
                if (result) {
                    const token = createToken({ email: user.email, role: user.rol });
                    res.status(200)
                        .cookie('access_token', token, { secure: true, httpOnly: true })
                        .json({ role: user.rol, cookie: token });
                } else {
                    res.status(400).json({ msg: "wrong credentials invalid password" });
                }
            }).catch((error) => {
                res.status(500).json({ msg: "Internal server error" });
            });
        }else{
            res.status(400).json({ msg: "wrong credentials user not found" });
        }

    } catch (error) {
        res.status(500).json({ msg: "Internal server error" });
    }
};

const logout = async (req, res) => {
    try {
        res.status(200)
            .cookie('access_token', "", { secure: true, httpOnly: true })
            .send();
    } catch (error) {
        res.status(500).json({ msg: "Internal server error" });
    }
};

const loginController = {
    login,
    logout,
};


module.exports = loginController;
```

Diferencias con el original:
Cambio en la Declaración de bcrypt: El cambio de var a let para importar bcryptjs

Corrección de la Propiedad de Rol: La modificación de user.role a user.rol sugiere una corrección de un error tipográfico? o una adaptación a la nomenclatura de la base de datos o del modelo de usuario?

Eliminación de la Función Adicional: La versión actualizada no incluye la función getAllUsers. Esto podría indicar una decisión de reducir la funcionalidad del módulo para enfocarse solamente en la autenticación, eliminando la funcionalidad no relacionada con el inicio o cierre de sesión.

Reducción del Objeto Exportado: La versión actualizada exporta solamente las funciones login y logout en el objeto loginController, lo cual es consistente con la eliminación de la función getAllUsers. Esto simplifica el módulo y lo enfoca más en la autenticación.

Cambios a realizar en el merge:

#Uso de bcrypt en lugar de bcryptjs:

- Cambia la importación de bcryptjs a bcrypt.

#Actualización del Modelo y la Utilidad JWT:

- Cambia require('../models/users.model') a require('../models/users').

- Cambia require('../config/jsonWebToken') a require('../utils/jwt').

#Implementación del Manejo de Intentos de Inicio de Sesión:

- Añade la variable loginAttempts para llevar un registro de los intentos de inicio de sesión fallidos y potencialmente bloquear usuarios después de múltiples intentos fallidos.

- Dentro de la función login, implementa la lógica para incrementar el contador de intentos fallidos (loginAttempts[email].attempts++) y bloquear al usuario temporalmente si excede un número determinado de intentos (por ejemplo, 5 intentos).

#Manejo de Respuestas de Error:

- Asegúrate de que las respuestas de error sean consistentes. Por ejemplo, usa res.status(401).json({ msg: "Invalid credentials" }) para intentos de inicio de sesión fallidos.

#Función getAllUsers:

- Si decido mantener la función getAllUsers, asegurarse de que esté implementada de manera segura y adecuada.

#Exportación del Módulo:

- Mantener la exportación del loginController con las funciones login, logout, y opcionalmente getAllUsers, dependiendo de si decido incluir esta última función¿¿¿????

Aplicamos el merge y los templates pendientes. Comento el codigo por claridad y el niño Jesús:

```
// Importando módulos requeridos
const bcrypt = require('bcrypt');
const usersModels = require('../models/users');
const { createToken } = require('../utils/jwt');

// Objeto para almacenar los intentos de inicio de sesión
let loginAttempts = {};

// Función de inicio de sesión
const login = async (req, res) => {
    try {
        // Extrayendo email y contraseña del cuerpo de la solicitud
        const { email, password } = req.body;

        // Verificar si la cuenta está temporalmente bloqueada debido a intentos fallidos de inicio de sesión
        if (loginAttempts[email] && loginAttempts[email].blockedUntil > Date.now()) {
            return res.status(401).json({ msg: "Cuenta temporalmente bloqueada debido a múltiples intentos fallidos de inicio de sesión" });
        }

        // Buscando al usuario en la base de datos
        const user = await usersModels.findOne({ where: { email: email } });
        if (user) {
            // Comparando la contraseña proporcionada con la contraseña almacenada
            bcrypt.compare(password, user.password).then((result) => {
                if (result) {
                    // Si la contraseña es correcta, resetear los intentos fallidos
                    delete loginAttempts[email];

                    // Crear un nuevo token
                    const token = createToken({ email: user.email, role: user.rol });

                    // Enviar el token en una cookie y como parte de la respuesta
                    res.status(200)
                        .cookie('access_token', token, { secure: true, httpOnly: true })
                        .json({ role: user.rol, cookie: token });
                } else {
                    // Si la contraseña es incorrecta, registrar el intento fallido
                    if (!loginAttempts[email]) {
                        loginAttempts[email] = { attempts: 1, blockedUntil: null };
                    } else {
                        loginAttempts[email].attempts++;
                        // Si hay 5 o más intentos fallidos, bloquear la cuenta durante 15 minutos
                        if (loginAttempts[email].attempts >= 5) {
                            loginAttempts[email].blockedUntil = Date.now() + 15 * 60 * 1000;
                        }
                    }
                    res.status(401).json({ msg: "Credenciales inválidas" });
                }
            }).catch((error) => {
                // Manejar error de bcrypt
                res.status(500).json({ msg: "Error interno del servidor" });
            });
        } else {
            // Si no se encuentra al usuario, enviar una respuesta de error
            res.status(401).json({ msg: "Credenciales inválidas" });
        }

    } catch (error) {
        // Manejar cualquier otro error
        res.status(500).json({ msg: "Error interno del servidor" });
    }
};
```

```
// Función de cierre de sesión
const logout = async (req, res) => {
    try {
        // Limpiar la cookie del token de acceso
        res.status(200)
            .cookie('access_token', "", { secure: true, httpOnly: true })
            .send();
    } catch (error) {
        // Manejar cualquier error
        res.status(500).json({ msg: "Error interno del servidor" });
    }
};

// Exportando las funciones de inicio y cierre de sesión
const loginController = {
    login,
    logout
};

module.exports = loginController;
```

Termino con este y sigo con el analisis de user.controller.js:

```
const usersModel = require("../models/users.model");
const { validationResult } = require('express-validator');
let bcrypt = require('bcryptjs');
require('dotenv').config()
const uuidV4 = require('uuid')

const salt = process.env.SALT

const getAllUsers = async (req, res) => {
  try {
    let user = await usersModel.findAll();
    res.status(200).json(user);
  } catch (error) {
    console.log(`ERROR: ${error}`);
    res.status(400).json({ msj: `ERROR: ${error}` });
  }
};

const readUser = async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  try {
    const email = req.body.email
    let user = await usersModel.findOne({ where: { email: email } });
    if (user) {
      res.status(200).json(user);
    } else {
      res.status(400).json({ msg: "wrong credentials user not found" });
    }
  } catch (error) {
    console.log(`ERROR: ${error}`);
    res.status(400).json({ msj: `ERROR: ${error}` });
  }
};
```

```
const createUser = async (req, res) => {
  try {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    let {nombre,apellido,email,password,rol} = req.body;

    if (rol == null) {
      //default role unprivileged user
      rol = 'asesor';
    }
    password = await bcrypt.hash(password,10)

    const id_usuario = uuidV4.v4()

    const data = {id_usuario,nombre,apellido,email,password,rol}

    console.log('datos para guardar en dB ', data);
    let answer = await usersModel.create(data);
    res.status(201).json(answer);
  } catch (error) {
    console.log(`ERROR: ${error}`);
    res.status(400).json({ msj: `ERROR: ${error}` });
  }
};

const deleteUser = async (req, res) => {
  try {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    const {email} = req.body;
    if (email) {
      let result = await usersModel.destroy({ where: { email: email } });
      if (result.deletedCount == 0)
        res.status(400).json({ message: `User con email ${email} no encontrado` });
      else
        res
          .status(200)
          .json({ message: "User BORRADO", user: { data } });
    } else {
      res.status(400).json({ message: "formato de User erroneo" });
    }
  } catch (error) {
    console.log(`ERROR: ${error}`);
    res.status(400).json({ msj: `ERROR: ${error}` });
  }
};

module.exports = {
  readUser,
  createUser,
  deleteUser,
  getAllUsers,
};
```

Decido implementar un fragmento de código para añadir como middleware que se encargue del tratamiento de mensajes de error para no filtrar información del sistema que pueda ser aprovechado para ataques dirigidos a esta:

```
// Middleware de manejo de errores
function errorHandler (err, req, res, next) {
  console.error(`ERROR: ${err.message}`);
  res.status(500).json({ message: 'Ha ocurrido un error interno' });
}
```

Se añade al final del código tal que asi:

```
    } else {
      res.status(400).json({ message: "formato de User erroneo" });
    }
  } catch (error) {
    next(error);
  }
};

// Middleware de manejo de errores
function errorHandler (err, req, res, next) {
  console.error(`ERROR: ${err.message}`);
  res.status(500).json({ message: 'Ha ocurrido un error interno' });
}

module.exports = {
  readUser,
  createUser,
  deleteUser,
  getAllUsers,
  errorHandler,
};
```

Como buena praxis decido hacer un favor y comentar el codigo ya que me he molestado en entenderlo:

```
// Importando los módulos necesarios
const usersModel = require("../models/users.model");
const { validationResult } = require('express-validator');
let bcrypt = require('bcryptjs');
require('dotenv').config()
const uuidV4 = require('uuid')

// Obteniendo la sal del archivo de configuración
const salt = process.env.SALT

// Controlador para obtener todos los usuarios
const getAllUsers = async (req, res, next) => {
  try {
    // Buscando todos los usuarios en la base de datos
    let user = await usersModel.findAll();
    // Enviando la respuesta con los usuarios
    res.status(200).json(user);
  } catch (error) {
    // Si hay un error, lo pasamos al middleware de manejo de errores
    next(error);
  }
};

// Controlador para leer un usuario
const readUser = async (req, res, next) => {
  // Validando la entrada del usuario
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  try {
    // Obteniendo el email del cuerpo de la solicitud
    const email = req.body.email
    // Buscando el usuario en la base de datos
    let user = await usersModel.findOne({ where: { email: email } });
    if (user) {
      // Si el usuario existe, enviamos la respuesta con el usuario
      res.status(200).json(user);
    } else {
      // Si el usuario no existe, enviamos un mensaje de error
      res.status(400).json({ msg: "wrong credentials user not found" });
    }
  } catch (error) {
    // Si hay un error, lo pasamos al middleware de manejo de errores
    next(error);
  }
};

// Middleware de manejo de errores
function errorHandler (err, req, res, next) {
  // Registrando el error en la consola
  console.error(`ERROR: ${err.message}`);
  // Enviando una respuesta con un mensaje de error genérico
  res.status(500).json({ message: 'Ha ocurrido un error interno' });
}

// Exportando los controladores y el middleware de manejo de errores
module.exports = {
  readUser,
  getAllUsers,
  errorHandler,
};
```

5 Conclusiones y trabajo pendiente:

Implementaciones de Seguridad ATM:

- **Validación de Entradas**: Uso de `validationResult` en `login.controller.js` para la validación de entradas.

- **Encriptación de Contraseñas**: Utilización de `bcrypt` en `login.controller.js` para comparar contraseñas encriptadas.

- **Manejo de Tokens JWT**: Creación y validación de tokens JWT en `jsonWebToken.js` y `decodeToken.js`.

- **Control de Acceso Basado en Roles**: Implementado en `adminRoutes.js` y `clientRoutes.js`, verificando los roles del usuario.

- **Cookies Seguras**: Establecimiento de cookies con `httpOnly` y `secure` en `login.controller.js`.

- **Manejo de Errores**: Implementación en varios archivos para proporcionar respuestas en caso de errores.

- **Seguridad en el Modelo de Datos**: En `users.model.js`, estructura segura y adecuada del modelo de usuario.

- **Prevención de Ataques de Fuerza Bruta**: Implementación de límites en los intentos de inicio de sesión fallidos para prevenir ataques de fuerza bruta.

- **Mensajes de Error Genéricos en Autenticación**: Uso de mensajes de error más genéricos en la autenticación para evitar revelar información sobre cuentas de usuario.

- **Encriptación de Contraseñas en el Registro**: Asegurarse de que las contraseñas se encripten antes de almacenarlas en la base de datos.

Áreas de Mejora o Implementación Pendiente:

- **HTTPS en Todo el Sitio**: Todas las comunicaciones entre el cliente y el servidor deben realizarse a través de HTTPS.

- **Auditoría de Seguridad y Pruebas**: Realización pendiente de pruebas de penetración y revisión post-código, have fun víctor.

- **Revisión del Uso de JWT y Cookies**: Verificar que los tokens JWT se manejen de manera segura, tanto en almacenamiento como en transmisión.

- **Protección contra CSRF (Cross-Site Request Forgery)**: Implementar medidas contra ataques CSRF si se utilizan cookies para la autenticación.

- **Validaciones en el Lado del Cliente**: Realizar validaciones de entrada en el lado del cliente para mejorar la seguridad.

- **Registro y Monitoreo de Actividades**: Implementar un sistema de registro para monitorear actividades sospechosas o inusuales.

- **Vulnerabilidad a Path Traversal**: La aplicación carece de protecciones adecuadas contra ataques de Path Traversal, lo que podría permitir a un atacante acceder a archivos y directorios almacenados fuera del directorio raíz web. Esta vulnerabilidad es particularmente crítica, ya que podría exponer información sensible del sistema.