## Análisis de seguridad

1. **Fuerza bruta en el login**

Si al momento de ingresar contraseñas no existe un límite de intentos fallidos, ni se aplican bloqueos o retrasos, un atacante puede adivinar una contraseña aplicando fuerza bruta probando con muchas combinaciones automáticamente en el endpoint de login /api/auth/login, pudiendo descubrir las cuentas de usuarios con contraseñas débiles.

**Contramedidas efectivas:**

1. **Rate limiting:** lo que se hace es limitar la cantidad de login por IP o por usuario en un tiempo determinado. Para ello se puede implementar utilizando la tecnología “express-rate-limit” la cual limita la cantidad de peticiones de un endpoint por IP.

**express-rate-limit**: librería de Node.js para Express que sirve para limitar la cantidad de solicitudes que un cliente puede hacer a tu API en un período de tiempo determinado.

Ejemplo de implementación de express-rate-limit:

Texto

El contenido generado por IA puede ser incorrecto.

En este ejemplo luego de 5 intentos fallidos, se aplica un bloqueo y se deben esperar 15 minutos para poder volver a ingresar una contraseña.

1. **Bloqueo temporal del usuario:** después de una N cantidad de intentos fallidos, se bloquea la cuenta por un tiempo.

**Implementación:**

Una captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

Se guarda un campo failed\_attemps y LockUntil en la base de datos y a través de la lógica del backend se incrementa failed\_attemps en cada login fallido, y si se supera el límite se establece lock\_until = NOW() + INTERVAL '30 minutes', y se rechaza el login hasta que expire el bloqueo.

1. **Falta de expiración y revocación de sesiones activas**

Actualmente, cuando un usuario cierra sesión, el token JWT se borra del frontend, pero **sigue siendo válido en el backend hasta que expire** (2 horas). Esto significa que, si un token es robado, un atacante podría usarlo para acceder a la cuenta durante todo el tiempo de vida del token. No existe un mecanismo para revocar o invalidar tokens antes de su expiración, lo que representa un riesgo significativo de acceso no autorizado.

**Solución:**

**Implementar una lista de revocación de token:** Cada vez que un usuario cierra sesión, el frontend manda el token al backend y guarda el jti del token junto con la fecha de expiración en la tabla Tokens\_revocados. Si alguien intenta usar ese token el backend lo detecta en la tabla y no permite el acceso. Cada cierto tiempo se borran de la tabla los tokens que ya expiraron.

**Implementación:**

En la **base de datos** se necesitaría crear la tabla Tokens\_revocados con las columnas de token\_id, user\_id, y fecha de expiración.

En el **backend**:

* Generación de tokens: incluir el campo jti único por token al generarlo.
* Limpiar en la tabla los tokens expirados.
* Middleware de autenticación:
* Verificar la firma y expiración del jwt y consultar la tabla de tokens revocados antes de permitir el acceso.

**Endpoints:**

* Logout: guarda el token en la base de datos al cerrar sesión.

1. **Contraseñas sin requisitos de complejidad**

**Descripción del riesgo**

Actualmente en el sistema en la función register, no se valida el nivel de seguridad de la contraseña creada, ni se exigen requisitos mínimos para la creación de estas como el uso de mayúsculas, símbolos, números o una cantidad de caracteres mínima. Esto le permite a los usuarios utilizar contraseñas comunes y poco seguras como 1234, qwery, contraseña entre otras. Esto debilita la autenticación permitiendo a los atacantes por medio de fuerza bruta o diccionarios, adivinar las contraseñas de los usuarios tener acceso a la información personal y financiera de los usuarios.

**Solución propuesta:**

Implementar políticas de complejidad de contraseñas que obliguen a los usuarios a definir credenciales seguras:

* Longitud mínima de 8 caracteres.
* Inclusión de mayúsculas, minúsculas números y símbolos especiales.
* Rechazo de contraseñas muy comunes.

**Implementación:**

* En el backend utilizar una librería como *validator* para validar el formato de la contraseña al registrar un usuario y al querer cambiar una contraseña.
* Rechazar cualquier contraseña que no cumpla con las políticas definidas.
* Crear una tabla en la base de datos con contraseñas comunes y siempre antes de guardar una nueva contraseña consultar esta base de datos y rechazarla si esta en la lista.

1. **No hay verificación de email**

**Descripción del riesgo:**

Actualmente en el sistema permite que cualquier persona se registre con cualquier dirección de correo electrónico, lo cual conlleva riesgos como suplantación de identidad, en el caso de el usuario olvide la contraseña no pueda recuperarla y la creación de cuentas spam que ocupen espacio en la base de datos.

**Solución propuesta:**

Implementar un proceso de verificación de email al registrarse un nuevo usuario:

* Se envía un correo electrónico de confirmación con un enlace único y temporal.
* La cuenta queda en estado pendiente hasta que el usuario haga click en el enlace
* Solo después de confirmar el email la cuenta se activa y puede iniciar sesión.

**Implementación técnica:**

En la base de datos se agrega un campo email\_verificado en la tabla usuarios.

**Registro**: Al registrarse, el backend genera un **token de verificación único** utilizando la librería nativa de Node.js crypto. Este token se convierte a cadena hexadecimal y se asocia al usuario en la base de datos, junto con una fecha de expiración. Luego, se envía un correo electrónico al usuario utilizando el servicio **SendGrid**, incluyendo un enlace con el token de verificación.

Texto

El contenido generado por IA puede ser incorrecto.

**Endpoint de verificación:** cuando el usuario hace clic en el enlace, el backend valida el token y marca el campo *email\_verificado* como *true*.

En autenticación no permitir el login si el email no ha sido verificado.

1. **Falta de validación y sanitización de datos de entrada en la función register**

**Descripción del riesgo:**

En la función register, los campos nombre, email y contraseña se reciben directamente del cuerpo de la petición (req.body) y se almacenan en la base de datos sin ninguna validación ni limpieza. Esto permite que usuarios malintencionados introduzcan datos peligrosos o inconsistentes.

Esto puede tener como consecuencia un ataque de tipo XSS almacenado, en el momento que un atacante ingrese por ejemplo en el campo nombre un script malicioso y puede ejecutar el código.

**Aclaración:**  
Aunque el frontend (React) suele escapar los datos al renderizarlos, guardar datos maliciosos en la base de datos es una mala práctica y puede ser peligroso si en el futuro se cambia el framework, se exportan datos, o se usan en otros contextos.

**Solución:**

**Validación de entrada**: en el backend se debe verificar que los campos cumplan con el formato esperado donde en el nombre solo se permitan letras y el email tenga un formato valido.

**Sanitización de datos:** Antes de guardar en la base de datos se deben eliminar caracteres peligrosos (<,>,&,”), evitando que se ejecute código malicioso si los datos se muestran en el frontend.

**Uso de librerías de sanitización y validación:** Librerias como validator.js para validadr emails (validator.isEmail) y escapar de caracteres peligrosos (validator.escape).