

SISTEMAS DE SEGURIDAD A IMPLEMENTAR

- Introducción.
- 1) Variables de entorno para credenciales.
- 2) Reducir privilegios del usuario MySQL
- 3) Hashear contraseñas de usuarios
- 4) Usar consultas parametrizadas
- 5) Restringir acceso a la base de datos (VPC)
- 6) Restringir CORS
- 7) Autenticación y gestión de sesiones
- 8) Uso seguro de fetch con tokens
- Conclusión.

INTRODUCCIÓN

Integrar la seguridad en cada fase del desarrollo

En muchos proyectos web, es común escuchar la frase “**primero hacemos la web y luego le añadimos seguridad**”, como si la seguridad fuera un complemento opcional que puede aplicarse al final, casi como una capa de pintura. Esta forma de pensar es, sin duda, **uno de los mayores errores** que pueden cometerse en el desarrollo de software moderno.

La seguridad no es una etapa final, sino una mentalidad que **debe acompañar al proyecto desde el primer boceto hasta el despliegue** en producción. Este enfoque se conoce como **Security by Design** y consiste en integrar la seguridad en cada fase del ciclo de vida del desarrollo del software (SDLC).

En este informe **analizamos las principales medidas de seguridad que debemos implementar** no solo en el backend y en la infraestructura cloud, sino también en el frontend y la comunicación cliente-servidor, dos áreas a menudo subestimadas, pero igual de críticas.

Desde el **uso correcto de fetch** en JavaScript, hasta el **endurecimiento de políticas en HTML** mediante cabeceras, **controles de CORS** y **validaciones** tanto del lado **del cliente** como **del servidor**, este documento explora una visión integral de cómo proteger un sistema web de manera proactiva y eficaz.

1) Variables de entorno para credenciales.



Actualmente, tenemos las credenciales en texto plano en la clase MotorSql.java de esta manera:

```
public class MotorSql implements IMotorSql {  @ mantel5
    private static final String DRIVER_NAME = "com.mysql.cj.jdbc.Driver";  1 usage
    public static final String MYSQL_URL = "jdbc:mysql://retodb.cwaiwnvcnh5j.us-e
    public static final String MYSQL_USER = "root";  1 usage
    public static final String MYSQL_PASS = "Aa12021888.";  1 usage
```

Para evitar el robo de las mismas en el caso de que alguien entrara al código fuente del backend, tenemos que:

1- Modificar dicha clase de java creando un archivo con extensión **.env** para meter ahí las credenciales.

2- Usar una **librería** en el archivo pom.xml para leer el **.env** como por ejemplo [dotenv-java](#).

```
<dependency>
  <groupId>io.github.cdimascio</groupId>
  <artifactId>dotenv-java</artifactId>
  <version>2.2.4</version>
</dependency>
```

3- Por último, **modificar el MotorSql.java para implementar las variables en lugar de las credenciales** directamente y así estar más protegidos ante este posible ataque.

```
public static final String MYSQL_URL = dotenv.get("MYSQL_URL");
public static final String MYSQL_USER = dotenv.get("MYSQL_USER");
public static final String MYSQL_PASS = dotenv.get("MYSQL_PASS");
```

2) Reducir privilegios del usuario MySQL

A día de hoy tenemos establecido el usuario root con todos los privilegios. En caso de que un atacante entrase en la DB a través de inyección SQL u otro medio, podría usar estas vulnerabilidades para ejecutar comandos directamente sobre la base de datos.

Por esto, tenemos que, desde la consola de MySQL meter estos comandos para **crear un usuario y darle solo los permisos de SELECT, INSERT, y UPDATE**. De esta forma si alguien entra a la misma solo podría realizar consultas para obtener información, insertar registros o actualizarlos.

```
CREATE USER 'reto_app'@'%' IDENTIFIED BY 'clave_segura';  
GRANT SELECT, INSERT, UPDATE ON esquema_reto.* TO 'reto_app'@'%';
```

3) Hashear contraseñas de usuarios

Otro error grave que debemos corregir es el de guardar las contraseñas de los usuarios en texto plano.

Para ello, usaremos **jBCrypt**, una biblioteca de Java que se utiliza para implementar el algoritmo **bcrypt**, que es una función de **hashing** segura y diseñada para proteger contraseñas en aplicaciones.

Así pues, insertaremos la dependencia en el pom.xml de esta manera:

```
<dependency>  
  <groupId>org.mindrot</groupId>  
  <artifactId>jbcrypt</artifactId>  
  <version>0.4</version>  
</dependency>
```

Y posteriormente, en el archivo UsuariosDao del backend (que es donde se ejecutan las sentencias SELECT, INSERT INTO, UPDATE y DELETE de los usuarios) añadiremos estas líneas de código para mitigar la amenaza.

```
import org.mindrot.jbcrypt.BCrypt;

// Al registrar
String hashed = BCrypt.hashpw(plainPassword, BCrypt.gensalt());

// Al autenticar
if (BCrypt.checkpw(plainPassword, hashedPasswordFromDB)) {
    // ok
}
```

Con esto, se importa la librería antes insertada para generar un **hash seguro** que es lo que realmente se guarda en la base de datos.

Lo que hace el if es que cuando un usuario intenta iniciar sesión, necesitamos verificar que la contraseña que ingresó sea la misma que la que se guardó en la base de datos, pero como **no tenemos la contraseña original** (solo el hash), debemos verificar que la contraseña ingresada coincida con el hash almacenado.

4) Usar consultas parametrizadas (PreparedStatement)

Actualmente, **solo en algunas clases** Dao del backend (recordemos, son las que ejecutan las sentencias SELECT, INSERT INTO, UPDATE y DELETE de la tabla en cuestión) tenemos precisamente configuradas las sentencias con parametrización predefinida para evitar y prevenir una **posible inyección SQL**.

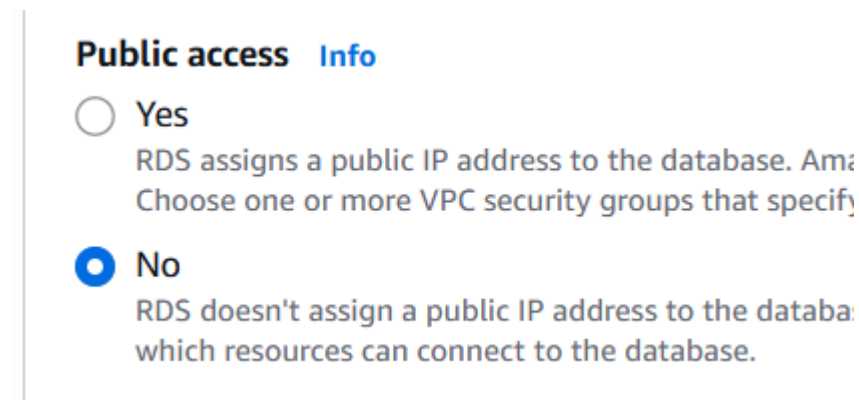
Deberemos cambiar esto, poniendo el PreparedStatement en todos ellos previniendo el ataque.

```
String sql = "SELECT * FROM usuarios WHERE correo = ?";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setString(1, email);
ResultSet rs = ps.executeQuery();
```

5) Restringir acceso a la base de datos (VPC)

Hemos puesto la base de datos en Amazon Web Services (AWS) como pública porque así nos lo dijo Santos. Para una mayor seguridad esto deberemos subvertirlo de la siguiente forma:

Cambiaremos la base de datos a red privada desactivando la opción de **Public access**. Y asegurarnos de que el grupo de seguridad solo acepte el tráfico desde nuestra instancia EC2 (en la que está el proyecto corriendo). Así:



Public access [Info](#)

☐ Yes
RDS assigns a public IP address to the database. Amazon chooses one or more VPC security groups that specify which resources can connect to the database.

☒ No
RDS doesn't assign a public IP address to the database, which resources can connect to the database.

6) Restringir CORS

Configuramos el CORS para sortear cualquier problema a la hora de lanzar el proyecto. Debido a esto pusimos un asterisco permitiendo que **cualquier dominio** pudiera realizar solicitudes al servidor sin importar su origen. Como se muestra aquí:

```
response.setHeader("Access-Control-Allow-Origin", "*"); //Permito a todos para pruebas
```

Debemos **cambiar el asterisco por nuestro dominio** para restringir que este sea el único que tenga permiso para acceder al servidor. De esta forma, tendremos:

```
response.setHeader("Access-Control-Allow-Origin", "https://meetgrill.retocsv.es");
```

7) Autenticación y gestión de sesiones

En un principio no tenemos control del login ni gestión de sesiones, cuando lo tengamos, deberemos añadir otra dependencia al pom.xml como java-jwt para realizar este control, creando así un token JWT para **autenticar y verificar la identidad de un usuario**.

Los tokens en URLs son un riesgo de seguridad porque pueden ser **registrados en logs de servidores o capturados en el historial del navegador**. Además, las URLs pueden ser fácilmente visibles en la barra de direcciones o pueden ser compartidas sin querer. Así que lo evitaremos de esta forma:

```
Algorithm algorithm = Algorithm.HMAC256("secret");
String token = JWT.create()
    .withSubject(userId)
    .withExpiresAt(new Date(System.currentTimeMillis() + 86400000))
    .sign(algorithm);
```

Así, creamos un algoritmo de **firma HMAC con SHA-256** que es un algoritmo de hash basado en una clave secreta.

Luego, **creamos el token JWT**, agregándole información relevante, y firmándolo con el algoritmo que definimos previamente.

Almacena el token en sessionStorage para que solo esté disponible durante la sesión activa, lo que mitiga los riesgos de XSS.

Siempre incluye el token en el encabezado **Authorization**, utilizando el formato **Bearer token**.

8) Uso seguro de fetch con tokens

Es tan importante securizar el backend como el frontend, los atacantes pueden entrar por cualquier vulnerabilidad esté en el lugar que esté.

Así pues, debemos mejorar nuestra conexión con el backend en el fetch usando un token. Ahora tenemos nuestra llamada a la api así:

```
async function cargarOfertas() {  
  try {  
    const response = await fetch('http://3.232.93.217:8080/api/ofertas');  
    if (!response.ok) throw new Error('Error al cargar ofertas');  
  }  
}
```

Cambiaremos esto de tal forma que crearemos un token para llamar a la api y recoger los datos del backend de forma controlada y segura. Así:

```
async function cargarOfertas() {  
  
  const token = sessionStorage.getItem("token");  
  
  if (!token) {  
    console.error('Token no encontrado. El usuario no está autenticado.');    return;  
  }  
  
  const response = await fetch('http://3.232.93.217:8080/api/ofertas', {  
    method: 'GET',  
    headers: {  
      'Authorization': `Bearer ${token}`,  
    },  
  });  
}
```


CONCLUSIÓN

La seguridad no es un “extra”, no es una tarea pendiente para cuando el producto esté terminado ni un simple parche para calmar auditorías. Es un requisito esencial y transversal que debe integrarse desde el minuto cero del desarrollo.

Este enfoque, conocido como Security by Design, nos obliga a repensar nuestras decisiones técnicas, a anticipar posibles vectores de ataque y a proteger todos los frentes: desde el backend y la base de datos, hasta la nube, el frontend y cada punto de comunicación con el usuario.

Hemos visto cómo errores comunes, como guardar contraseñas en texto plano, usar credenciales embebidas en el código, o dejar expuestos endpoints sin validación ni autenticación, pueden convertirse en vulnerabilidades críticas. Pero también hemos visto cómo prevenirlos: aplicando validaciones del lado cliente y servidor, utilizando tokens seguros, headers HTTP protectores, cifrado, y una correcta gestión de permisos y configuraciones.

Seguridad no es solo tecnología: es cultura, es metodología, es previsión. Cuanto antes la integremos en el trabajo, más resiliente será frente a las amenazas reales que existen ahí fuera.

Esperamos que estas soluciones a los distintos problemas clasificados sean correctas y funcionales para integrarlas en el proyecto.

Gracias por tu tiempo y te pedimos disculpas por no haber realizado inicialmente la debida securización del mismo.