Informe de Análisis de Seguridad y Mitigación de Vulnerabilidades OWASP Top 10 en la aplicación AthletIA

Fecha: 19/10/2025

Integrantes: Gabriel Cevallos, David Paccha, Ivan Fernandez

Proyecto: Backend de la aplicación AthletIA

Introducción:

El presente informe detalla el análisis de seguridad realizado sobre la parte del backend de la aplicación AthletIA, desarrollado con el framework NestJS y TypeScript, un superset de JavaScript. El objetivo es identificar y mitigar las principales vulnerabilidades de seguridad según el estándar OWASP Top 10, garantizando un sistema robusto, seguro y éticamente responsable.

1. Identificación de vulnerabilidades OWASP

1.1. Cryptographic Failures (Fallos Criptográficos): La aplicación maneja contraseñas de usuario y tokens de sesión (JWT). Si las contraseñas se almacenan en texto plano o los tokens no se gestionan de forma segura, un atacante que acceda a la base de datos podría suplantar la identidad de cualquier usuario.

1.2. Broken Access Control (Control de Acceso Roto): En una aplicación como Athletia, donde existen datos de usuario, rutinas y perfiles, etc. Es crítico que un usuario no pueda ver o modificar datos de otro. Por ejemplo, un usuario no debería poder editar o eliminar la rutina de entrenamiento de otro usuario.

1.3. Injection (Inyección): Esta vulnerabilidad ocurre cuando datos no confiables son enviados a un intérprete como parte de un comando o consulta. En AthletIA, esto podría manifestarse con una inyección SQL al buscar ejercicios o al filtrar rutinas, permitiendo a un atacante leer, modificar o borrar información de la base de datos.

1.4. Identification and Authentication Failures (Fallos de Identificación y Autenticación): Relacionado con la gestión de sesiones. Si los tokens nunca expiran o si no se invalidan al cerrar sesión, un token robado podría ser usado indefinidamente para acceder a la cuenta de un usuario.

1.5. Security Misconfiguration (Configuración de Seguridad Incorrecta): Errores en la configuración del servidor, el framework o de las dependencias pueden exponer la aplicación. Por ejemplo, dejar mensajes de error detallados en producción podría revelar información sensible sobre la estructura interna de la aplicación.

2. Análisis del Riesgo e Impacto Potencial

2.1. Fallos Criptográficos:

Riesgo: Crítico

Impacto: La exposición de contraseñas en texto plano o como un hash débil comprometería todas las cuentas de usuario. Un atacante podría robar la base de datos y reutilizar esas credenciales en otros servicios.

2.2. Control de Acceso Roto:

Riesgo: Alto

Impacto: Un usuario malintencionado podría acceder a los perfiles, cuentas, rutinas y otros aspectos de los usuarios, violando su privacidad. Podría modificar o eliminar sus datos de entrenamiento, creando una pésima experiencia de usuario y pérdida de confianza en la aplicación.

2.3. Inyección:

Riesgo: Alto

Impacto: Una inyección SQL exitosa podría permitir a un atacante eludir la autenticación, obtener toda la información almacenada en la base de datos, modificar registros o incluso borrar la base de datos por completo.

2.4. Fallos de Autenticación:

Riesgo: Alto

Impacto: Un token JWT robado que no expira, otorga acceso perpetuo a la cuenta de la víctima aún si cambia su contraseña. Esto permite al atacante actuar en su nombre de forma indefinida.

2.5. Configuración de Seguridad Incorrecta:

Riesgo: Medio

Impacto: Exponer cabeceras HTTP con información del servidor o mostrar errores detallados en producción facilita a los atacantes el reconocimiento del sistema, ayudándoles a encontrar vulnerabilidades conocidas para el software específico que está utilizando.

3. Medidas de Mitigación Aplicadas

Para abordar las vulnerabilidades identificadas, se han implementado las siguientes medidas técnicas en el código y la configuración del proyecto “AthletIA”.

3.1 Mitigación de Fallos Criptográficos:

- **Hashing de contraseñas:** Se utilizó la librería “argon2” para generar un hash seguro de las contraseñas de los usuarios antes de almacenarlas en la base de datos.
- **Gestión de Secretos:** La clave secreta del token JWT se gestiona a través de variables de entorno (archivo .env) y es cargada mediante el ConfigModule de NestJS, evitando que quede expuesta en el código fuente.

3.2 Mitigación de Control de Acceso Roto:

- I**mplementación de Guards:** Se utiliza el sistema de Guards de NestJS. Esto se aplica a las rutas protegidas para asegurar que solo los usuarios autenticados puedan acceder. Adicionalmente, dentro de los servicios, se valida que el ID de los usuarios que realizan la petición coincida con el ID del propietario del recurso que se intenta modificar.

3.3 Mitigación de Inyección:

- **Uso de TypeORM y DTOs:** El uso de ORM y DTO (Data Transfer Object) para todas las interacciones con la base de datos previene la inyección SQL, ya que parametriza las consultas de forma automática. Además, se emplean DTOs con el paquete class-validator para validar y sanear rigurosamente todos los datos de entrada en los controladores.

3.4 Mitigación de Fallos de Autenticación:

- **Expiración de JWT:** Dentro del código el JwtModule está configurado para que los tokens de acceso tengan una vida útil corta dentro de un tiempo configurado, lo que limita la ventana de oportunidad para un atacante en caso de robo del token.

3.5 Mitigación de Configuración de Seguridad Incorrecta:

- **Uso de Helmet:** Se añadió el middleware “helmet” dentro del código fuente. Este paquete establece automáticamente cabeceras HTTP seguras, como Strict-Transport-Security, y elimina la cabecera X-Powered-By que expone la información del servidor.

- **Entornos de Producción:** La configuración de la aplicación diferencia entre el entorno de desarrollo y el de producción, asegurando que los errores detallados y las trazas de pila solo se muestran en desarrollo.
- **Implementación de Refresh Tokens:** Se ha implementado una estrategia de tokens de refresco. Estos tokens de mayor duración se almacenan de forma segura y se utilizan exclusivamente para solicitar nuevos tokens de acceso, mejorando la seguridad sin perjudicar la experiencia del usuario.

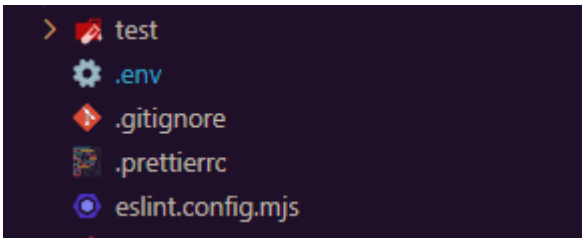
4. Evidencias Técnicas

4.1 Mitigación de Fallos Criptográficos:

- **Hashing de contraseñas**

```
async create(registerRequest: RegisterAccountRequest): Promise<Account> {
  const account = this.accountsRepository.create({
    ...registerRequest,
    password: await argon2.hash(registerRequest.password),
  });
  await this.accountsRepository.save(account);
  return account;
}
```

- **Gestión de Secretos**




```
imports: [
  PassportModule.register({ session: false }),
  JwtModule.register({
    global: true,
    secret: process.env.JWT_SECRET_KEY_ACCESS || 'defaultSecretKey',
  }),
  AccountsModule,
],
```

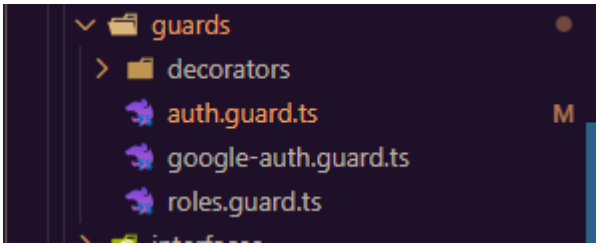
```ts
import { TypeOrmModuleOptions } from '@nestjs/typeorm';
import * as dotenv from 'dotenv';

dotenv.config();

export const typeOrmConfig: TypeOrmModuleOptions = {
  type: 'postgres',
  host: 'localhost',
  port: parseInt(process.env.DB_PORT!),
  username: process.env.DB_USERNAME!,
  password: process.env.DB_PASSWORD!,
  database: process.env.DB_NAME!,
  entities: [__dirname + '../../../**/*.entity{.ts,.js}'],
  synchronize: true,
  dropSchema: true,
};
```

4.2 Mitigación de Control de Acceso Roto:

- **Implementación de Guards**



src > auth > guards > auth.guard.ts > AuthGuard > canActivate

```ts
import { Reflector } from '@nestjs/core';
import { IS_PUBLIC_KEY } from 'src/auth/guards/decorators/public.decorator';
import { AccountsService } from 'src/users/accounts/accounts.service';
import { AccountStatus } from 'src/users/accounts/enum/account-status.enum';
import { UserPayload } from '../interfaces/user-payload.interface';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(
    private jwtService: JwtService,
    private reflector: Reflector,
    private accountsService: AccountsService,
  ) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const isPublic = this.reflector.getAllAndOverride<boolean>(IS_PUBLIC_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);

    if (isPublic) {
      return true;
    }

    const request = context.switchToHttp().getRequest<Request>();
    const token = this.extractTokenFromHeader(request);

    if (!token) {
      throw new UnauthorizedException();
    }

    try {
      const payload = this.jwtService.verify<UserPayload>(token, {
        secret: jwtConstants.secret,
```

src > auth > guards > google-auth.guard.ts > GoogleAuthGuard

```ts
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class GoogleAuthGuard extends AuthGuard('google') {}

```

src > auth > guards > roles.guard.ts > RolesGuard > canActivate

```ts
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Request } from 'express';
import { ROLES_KEY } from 'src/auth/guards/decorators/roles.decorator';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get<string[]>(ROLES_KEY, context.getHandler());
    if (!roles) {
      return true;
    }
    const request = context.switchToHttp().getRequest<Request>();
    const user = request.user;
    return user && roles.includes(user.role);
  }
}

```

4.3 Mitigación de Inyección:

- **Uso de TypeORM y DTOs**

```ts
@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
    }),
    TypeOrmModule.forRoot(typeOrmConfig),
    AuthModule,
    AccountsModule,
    ProfilesModule,
    ExercisesModule,
  ],
  controllers: [AppController],
  providers: [AppService, BootstrapService],
})
export class AppModule {}
```

src > users > accounts > TS account.entity.ts > Account > role

```ts
import {
  Column,
  Entity,
  JoinColumn,
  OneToOne,
  PrimaryGeneratedColumn,
} from 'typeorm';
import { AccountStatus } from './enum/account-status.enum';
import { Role } from './enum/role.enum';
import { Profile } from 'src/users/profiles/profile.entity';

@Entity()
export class Account {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ unique: true })
  email: string;

  @Column({ nullable: true })
  password: string;

  @Column({
    type: 'enum',
    enum: Role,
    default: Role.USER,
  })
  role: Role;

  @Column({
    type: 'enum',
    enum: AccountStatus,
    default: AccountStatus.UNPROFILED,
  })
```

src > workout > exercises > dto > TS exercises.dto.ts > ...

```ts
import {
  IsArray,
  IsEnum,
  IsNotEmpty,
  IsOptional,
  IsString,
  IsUrl,
} from 'class-validator';
import { MuscleTarget } from '../enum/muscle-target.enum';
import { ExerciseType } from '../enum/exercise-type.enum';

export class ExerciseRequest {
  @IsString()
  @IsNotEmpty()
  name: string;

  @IsString()
  @IsNotEmpty()
  description: string;

  @IsUrl()
  @IsNotEmpty()
  video: string;

  @IsArray()
  @IsEnum(MuscleTarget, {
    each: true,
    message: `Each muscleTarget must be one of: ${Object.values(MuscleTarget).join(', ')}`,
  })
  @IsNotEmpty()
  muscleTarget: MuscleTarget[];

  @IsArray()
  @IsEnum(ExerciseType, {
```

4.4 Mitigación de Fallos de Autenticación:

- **Expiración de JWT**

```
export const jwtConstants = {
  secret: process.env.JWT_SECRET_KEY,
  // expiration in seconds
  refreshExpiration: 60 * 60 * 24 * 7, // 7 days
  accessExpiration: 60 * 60, // 1 hour
};
```

4.5 Mitigación de Configuración de Seguridad Incorrecta:

- **Uso de Helmet**

```
app.use(
  helmet({
    // Keep defaults; customize as needed
    // Example: only enable HSTS in production behind HTTPS
    hsts: process.env.NODE_ENV === 'production' ? undefined : false,
    // Hide X-Powered-By
    hidePoweredBy: true,
    // Basic referrer policy
    referrerPolicy: { policy: 'no-referrer' },
    // Cross-Origin Resource Policy (adjust for your static hosting if needed)
    crossOriginResourcePolicy: { policy: 'cross-origin' },
  }),
);
```

- **Entornos de Producción**.

```
    "start:dev": "nest start --watch",
    "start:debug": "nest start --debug --watch",
    "start:prod": "node dist/main",
```

- **Implementación de Refresh Tokens**

```
  @Public()
  @Post('refresh-token')
  @HttpCode(200)
  async refreshToken(
    @Body('refreshToken') refreshToken: string,
  ): Promise<TokenResponse> {
    const tokenResponse = await this.authService.refreshToken(refreshToken);
    return tokenResponse;
  }

```

5. Reflexión Final

La realización de este análisis de seguridad ha sido un ejercicio de aprendizaje fundamental que ha transformado nuestra perspectiva sobre el desarrollo de software, pues, hemos comprendido que la seguridad es un pilar que debe ser integrado desde la concepción de la primera línea de código.

Comprender las vulnerabilidades OWASP Top 10 y sus mitigaciones es crucial para nuestro desarrollo profesional, permitiéndonos identificar fallos de seguridad en nuestra aplicación al igual que, implementar soluciones proactivas desde las primeras etapas del desarrollo, como el hashing de contraseñas, control de acceso basado en roles, validación de datos con DTOs y gestión de tokens JWT con expiración.