# Seguridad Avanzada en APIs RESTful

La seguridad en APIs RESTful es fundamental para proteger datos sensibles, prevenir ataques y garantizar la integridad de las aplicaciones. Este documento cubre prácticas avanzadas más allá de la autenticación básica con JWT, abordando vulnerabilidades comunes y medidas de protección robustas.

## 1. Vulnerabilidades Comunes y Mitigaciones

### 1.1. OWASP API Security Top 10

Las principales vulnerabilidades según OWASP incluyen:

- **API1: Broken Object Level Authorization**
  - **Riesgo**: Acceso no autorizado a recursos de otros usuarios.
  - **Mitigación**: Validar permisos en cada endpoint. Ejemplo:

```
javascript

// Middleware de autorización
const checkResourceOwnership = async (req, res, next) => {
  const resource = await Resource.findById(req.params.id);
  if (resource.userId.toString() !== req.user.id) {
    return res.status(403).json({ error: "Acceso denegado" });
  }
  next();
};
```

- **API2: Broken Authentication**
  - **Riesgo**: Tokens débiles o mal gestionados.
  - **Mitigación**:
    - Usar JWT con firmas robustas (HS256 o RS256).

- Implementar refresh tokens con rotación.
- Almacenar tokens en cookies HttpOnly y Secure.

### 1.2. Ataques de Inyección

- **SQL/NoSQL Injection**:
  - **Mitigación**: Usar consultas parametrizadas. Ejemplo en Mongoose:

```javascript
// ❌ Vulnerable
User.find({ email: req.query.email });

// ✅ Seguro
User.find({ email: { $eq: req.query.email } });
```

- **XSS (Cross-Site Scripting)**:
  - **Mitigación**: Sanitizar entradas con librerías como dompurify o validator.

## 2. Configuración Avanzada de Middlewares

### 2.1. Helmet.js

Configura headers de seguridad HTTP:

```javascript
import helmet from "helmet";

app.use(helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'", "trusted-cdn.com"],
    },
  },
  crossOriginEmbedderPolicy: true,
}));
```

### 2.2. Rate Limiting

Previene ataques de fuerza bruta con express-rate-limit:

```javascript
import rateLimit from "express-rate-limit";

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutos
  max: 100, // Límite de 100 requests por IP
  message: "Demasiadas solicitudes desde esta IP",
});
app.use("/api/auth/", limiter); // Aplicar solo a endpoints críticos
```

### 2.3. CORS Seguro

Restringe origins permitidos:

```javascript
import cors from "cors";

app.use(cors({
  origin: process.env.FRONTEND_URL, // Solo permitir el frontend
  methods: ["GET", "POST", "PUT", "DELETE"],
  credentials: true, // Permitir cookies
}));
```

### 3. Validación y Sanitización de Datos

#### 3.1. Express-Validator

Valida y sanitiza entradas:

```javascript
import { body, validationResult } from "express-validator";

const validateUserInput = [
  body("email").isEmail().normalizeEmail(),
  body("password").isLength({ min: 8 }).escape(),
];

app.post("/api/register", validateUserInput, (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  // Procesar datos sanitizados
});
```

#### 3.2. Sanitización Personalizada

Elimina caracteres peligrosos:

```
const sanitizeInput = (input) => {
  return input.replace(/[<>]/g, ""); // Elimina < y >
};
```

## 4. Gestión Segura de Tokens

### 4.1. Refresh Tokens con Rotación

- **Almacenamiento**: En base de datos con hash (bcrypt).
- **Rotación**: Invalidar tokens previos al generar nuevos.

```
// Ejemplo de rotación
user.refreshTokens = user.refreshTokens.filter(token => token !== oldRefreshToken);
await user.save();
```

### 4.2. Short-Lived Access Tokens

- **Duración**: 15-30 minutos para access tokens.
- **Refresh automático**: Interceptor en frontend:

```
javascript

axios.interceptors.response.use(null, async (error) => {
  if (error.response.status === 401) {
    await refreshToken(); // Lógica de refresh
    return axios(error.config); // Reintentar request
  }
  return Promise.reject(error);
});
```

## 5. Monitorización y Logging

### 5.1. Detección de Anomalías

- **Logs de seguridad**: Registrar intentos de acceso fallidos.
- **Herramientas**: Use morgan para logging HTTP:

```javascript
import morgan from "morgan";

app.use(morgan("combined")); // Logs completos
```

### 5.2. Auditoría de Seguridad

- **Checks automáticos**: Con herramientas como npm audit o snyk.
- **Penetration testing**: Simular ataques con OWASP ZAP o Burp Suite.

## 6. Configuración para Producción

### 6.1. Variables de Entorno Críticas

```env
# JWT
JWT_SECRET=clave-mínimo-32-caracteres-aleatoria
JWT_EXPIRE=15m

# Base de datos
DB_URI=mongodb+srv://user:pass@cluster...?retryWrites=true&w=majority

# CORS
FRONTEND_URL=https://midominio.com
```

### 6.2. Headers de Seguridad

- **HSTS: Force HTTPS.**
- **X-Content-Type-Options: Previene MIME sniffing.**

```javascript
app.use(helmet.hsts({ maxAge: 31536000, includeSubDomains: true }));
```

**Herramientas Recomendadas**

1. OWASP ZAP: Escáner de vulnerabilidades.
2. Postman: Testing de endpoints con scripts de seguridad.
3. Snyk: Detección de vulnerabilidades en dependencias.
4. Helmet.js: Configuración automática de headers seguros.

La seguridad avanzada en APIs requiere un enfoque en capas: validación, autenticación robusta, headers seguros y monitorización. Implementar estas prácticas reduce riesgos y protege tanto los datos como la infraestructura.