BUENAS PRÁCTICAS DE SEGURIDAD EN APIS

01. Usar Autenticación Segura (JWT)

02. Implementar Autorización con RBAC o ACL

03. Usar HTTPS para Cifrar Datos

04. Implementar CORS Seguro

05. Proteger Contra Cross-Site Scripting

06. Validar y Sanitizar JSON de Entradas

07. Usar Headers de Seguridad con Helmet

08. Registrar Logs y Monitorea Actividad

09. Limitar Peticiones para Prevenir Ataques de Fuerza Bruta, Implementar Autorización con RBAC (Roles) o ACL (permisos)

03. USAR HTTPS PARA CIFRAR DATOS

Siempre usa HTTPS para proteger los datos en tránsito.

forza HTTPS con middleware:

```
app.use((req, res, next) => {
  if (!req.secure) {
    return res.redirect('https://' + req.headers.host + req.url);
  }
  next();
});
```

04. IMPLEMENTAR CORS SEGURO

Permite solo dominios específicos y bloquea peticiones sospechosas.

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

const corsOptions = {
  origin: ['https://tu-sitio.com'], // Dominios permitidos
  methods: 'GET,POST,PUT,DELETE',
  allowedHeaders: ['Content-Type', 'Authorization']
};

app.use(cors(corsOptions));
```

05.CROSS-SITE SCRIPTING (XSS)

Cross-Site Scripting (XSS) es una vulnerabilidad de seguridad web que permite a un atacante inyectar código JavaScript malicioso en una aplicación web. Este código se ejecuta en el navegador del usuario y puede robar datos, secuestrar sesiones o modificar la página.

01. Escapar y Sanitizar Entradas del Usuario

Nunca confíes en los datos de entrada. Usa librerías para limpiar y validar lo que los usuarios envían.

Librerías recomendadas: express-validator y sanitize-html para sanitizar datos:

```
const express = require('express');
const { body, validationResult } = require('express-validator');
const sanitizeHtml = require('sanitize-html');

const app = express();
app.use(express.json());

app.post('/comment',
  body('comment').isString().trim().escape(), // Valida y escapa entrada
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });

    const safeComment = sanitizeHtml(req.body.comment); // Limpia HTML peligroso
    res.send({ message: "Comentario seguro", safeComment });
});

app.listen(3000, () => console.log("Servidor en http://localhost:3000"));
```

02. Configurar Content Security Policy (CSP)

CSP evita la ejecución de scripts no autorizados, como aquellos inyectados por XSS.

Instala helmet para configurar CSP automáticamente.

Esto evita que scripts maliciosos inyectados se ejecuten.

```
const helmet = require('helmet');
app.use(helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"], // Solo permite recursos del mismo dominio
      scriptSrc: ["'self'", "'https://trusted-cdn.com'"], // Bloquea scripts externos
      objectSrc: ["'none'"], // Bloquea Flash y otros objetos inseguros
      upgradeInsecureRequests: [],
    }
  }
}));
```

03. Usar httpOnly y Secure en Cookies

Si usas JWT o sesiones, evita que los scripts accedan a las cookies para prevenir robo de tokens.

Configuración de cookies seguras en Express:

```
app.use(require('cookie-parser')());

app.use((req, res, next) => {
  res.cookie('sessionId', 'TOKEN_SEGURO', {
    httpOnly: true,  // Evita acceso desde JavaScript
    secure: true,    // Solo permite en HTTPS
    sameSite: 'Strict' // Previene CSRF
  });
  next();
});
```

04. Validar y Filtrar Datos en el Backend

No permitas que los usuarios envíen caracteres peligrosos o código malicioso.

Ejemplo de validación estricta:

Esto evita que los usuarios inyecten scripts en parámetros de búsqueda.

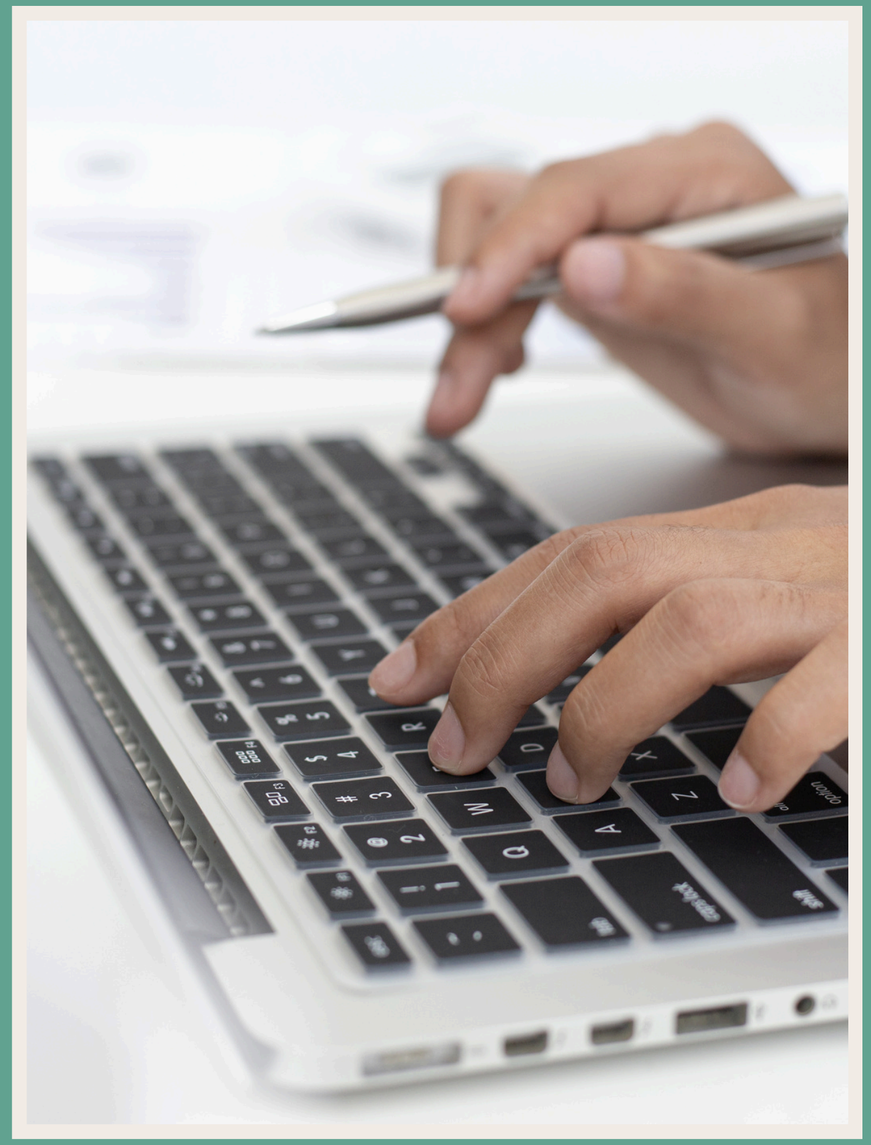
```
app.post('/search', (req, res) => {
  const query = req.body.query;

  if (!/^[a-zA-Z0-9\s]+$/.test(query)) { // Solo permite letras y números
    return res.status(400).send("Entrada inválida");
  }

  res.send({ message: "Búsqueda válida", query });
});
```

07.CONFIGURACIÓN DE CABECERAS HTTP SEGURAS EN EXPRESS.JS

Las cabeceras HTTP son clave para proteger una aplicación web contra ataques como XSS. En Express.js, podemos mejorar la seguridad configurando correctamente estas cabeceras.



USA HELMET PARA CONFIGURAR CABECERAS DE SEGURIDAD

La forma más sencilla de establecer cabeceras HTTP seguras en Express.js es usando Helmet.

```
const express = require('express');
const helmet = require('helmet');

const app = express();
app.use(helmet()); // Aplica todas las protecciones por defecto

app.get('/', (req, res) => {
  res.send('¡Aplicación segura con Helmet!');
});

app.listen(3000, () => console.log('Servidor corriendo en http://localhost:3000'));
```

CONFIGURACIÓN PERSONALIZADA DE HELMET

```
app.use(helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'", "'https://trusted-cdn.com'"], // Permite solo scripts
      objectSrc: ["'none'"],
      upgradeInsecureRequests: []
    }
  },
  frameguard: { action: 'deny' }, // Bloquea iframes
  hsts: { maxAge: 31536000, includeSubDomains: true }, // Fuerza HTTPS
  referrerPolicy: { policy: 'no-referrer' }, // No envía información de Referer
  xssFilter: true, // Habilita filtro XSS en navegadores
}));
```

08. REGISTRAR LOGS Y MONITOREAR ACTIVIDAD

Registra intentos de acceso y errores.

```
npm install morgan
```

```
const morgan = require('morgan');
app.use(morgan('combined')); // Registra todas las peticiones
```

09. LIMITA PETICIONES PARA PREVENIR ATAQUES DE FUERZA BRUTA

Usar rate limiting para bloquear intentos excesivos de login

```
npm install express-rate-limit
```

```
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutos
  max: 100, // Máximo 100 peticiones por IP
  message: "Demasiadas solicitudes, intenta más tarde",
});

app.use('/login', limiter);
```