EDUCACIÓN CONTINUA PUCP

Facultad de Ciencias e Ingeniería

Programa de Especialización en **Desarrollo Web, Front y Back End**

Curso Pruebas, Seguridad y DevOps

Sesión 4

Tema 2: Seguridad en desarrollo web

Docente: Jean Paul Curiñaupa Taype

EDUCACIÓN CONTINUA PUCP

Facultad de Ciencias e Ingeniería

Índice

EDUCACIÓN CONTINUA PUCP

Facultad de Ciencias e Ingeniería

1. Herramienta de seguridad con Node.js

Node.js cuenta con diversas herramientas y bibliotecas para mejorar la seguridad en aplicaciones web. Estas herramientas ayudan a prevenir ataques comunes, gestionar datos sensibles y proteger la infraestructura. A continuación, se describen en las siguientes diapositivas las principales herramientas y cómo usarlas.

EDUCACIÓN CONTINUA PUCP | Facultad de Ciencias e Ingeniería

1. Herramienta de seguridad con Node.js

1.1 Helmet: es una biblioteca que ayuda a configurar cabeceras HTTP seguras para proteger tu aplicación de ataques como XSS, clickjacking y ataques de inyección de contenido.

Características principales:

- Configuración automática de cabeceras HTTP seguras.
- Prevención de ataques relacionados con CSP (Content Security Policy).
- Fácil de integrar en aplicaciones Express.

Uso básico:

```
const helmet = require("helmet");
const express = require("express");
const app = express();

app.use(helmet());
```

Qué protege:

- XSS: Añade cabeceras que evitan la ejecución de scripts maliciosos.
- Clickjacking: Configura X-Frame-Options para evitar la incrustación no autorizada de tu sitio.
- MIME Sniffing: Configura X-Content-Type-Options para que los navegadores no adivinen el tipo de contenido.

EDUCACIÓN CONTINUA PUCP
Facultad de Ciencias e Ingeniería

1. Herramienta de seguridad con Node.js

1.2 Bcrypt: Es una biblioteca para el hashing de contraseñas. Garantiza que las contraseñas se almacenan de forma segura, incluso si la base de datos es comprometida.

Características principales:

- Generación de contraseñas "hasheadas" con sal para mayor seguridad.
- Resistente a ataques de fuerza bruta y rainbow tables.

Uso básico:

```
const bcrypt = require("bcrypt");

const saltRounds = 10;
const password = "miContraseñaSegura";

// Hash de una contraseña
bcrypt.hash(password, saltRounds, (err, hash) => {
  console.log("Hashed password:", hash);

  // Comparar contraseña con hash
  bcrypt.compare(password, hash, (err, result) => {
    console.log("¿Contraseña válida?", result);
  });
});
```

Qué protege:

- Evita el almacenamiento de contraseñas en texto plano.
- Dificulta el descifrado de contraseñas incluso si se obtiene acceso a la base de datos.

EDUCACIÓN CONTINUA PUCP | Facultad de Ciencias e Ingeniería

1. Herramienta de seguridad con Node.js

1.3 Jsonwebtoken (JWT): Es una herramienta para implementar autenticación basada en tokens. Permite autenticar usuarios y proteger rutas en aplicaciones.

Características principales:

- Generación de tokens firmados para sesiones de usuario.
- Fácil integración con middleware.
- Expiración configurable de tokens.

Uso básico:

```
const jwt = require("jsonwebtoken");

const secretKey = "miClaveSecreta";
const userData = { id: 1, role: "admin" };

// Generar token
const token = jwt.sign(userData, secretKey, { expiresIn: "1h" });
console.log("Token:", token);

// Verificar token
jwt.verify(token, secretKey, (err, decoded) => {
  if (err) {
    console.log("Token inválido");
  } else {
    console.log("Datos del usuario:", decoded);
  }
});
```

Qué protege:

- Control de acceso seguro basado en tokens.
- Prevención de sesiones no autorizadas.

EDUCACIÓN CONTINUA PUCP | Facultad de Ciencias e Ingeniería

1. Herramienta de seguridad con Node.js

1.4 Express-rate-limit: Es una herramienta que permite limitar el número de solicitudes que un cliente puede realizar en un periodo de tiempo definido, ayudando a prevenir ataques de fuerza bruta o denegación de servicio (DoS).

Características principales:

- Configuración personalizada de límites de solicitudes.
- Registro de intentos fallidos.

Uso básico:

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

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

app.use(limiter);
```

Qué protege:

- Prevención de ataques de fuerza bruta en formularios de inicio de sesión.
- Reducción de sobrecarga en el servidor causada por ataques DoS.

EDUCACIÓN CONTINUA PUCP
Facultad de Ciencias e Ingeniería

1. Herramienta de seguridad con Node.js

1.5 Cors: Es una herramienta que gestiona las políticas de intercambio de recursos de origen cruzado, permitiendo o restringiendo solicitudes desde diferentes dominios.

Características principales:

- Configuración granular de orígenes permitidos.
- Protección contra ataques relacionados con solicitudes no autorizadas entre dominios.

Uso básico:

```
const cors = require("cors");
const express = require("express");
const app = express();

app.use(cors({ origin: "https://example.com" })); // Solo permite solicitudes desde examp
```

Qué protege:

- Prevención de accesos no autorizados desde dominios externos.

1. Herramienta de seguridad con Node.js

1.6 NPM Audit: Es una herramienta integrada en Node.js para analizar vulnerabilidades en las dependencias de un proyecto.

Características principales:

- Identificación de paquetes con vulnerabilidades conocidas.
- Proporciona pasos para corregir las vulnerabilidades.

Uso básico:

```
npm audit
npm audit fix
```

Qué protege:

- Detección temprana de dependencias inseguras.

EDUCACIÓN CONTINUA PUCP | Facultad de Ciencias e Ingeniería

1. Herramienta de seguridad con Node.js

1.7 Winston: Es una biblioteca para el registro de logs, permitiendo registrar eventos, errores y actividades sospechosas en la aplicación.

Características principales:

- Registro de logs en archivos o servicios externos.
- Configuración de niveles de severidad.

Uso básico:

```
const winston = require("winston");

const logger = winston.createLogger({
  level: "info",
  format: winston.format.json(),
  transports: [
    new winston.transports.File({ filename: "error.log", level: "error" }),
    new winston.transports.Console(),
  ],
});

logger.info("Mensaje informativo");
logger.error("Mensaje de error");
```

Qué protege:

- Identificación de actividades sospechosas mediante auditoría de logs.

EDUCACIÓN CONTINUA PUCP
Facultad de Ciencias e Ingeniería

1. Herramienta de seguridad con Node.js

1.8 Dotenv: Permite gestionar variables de entorno para proteger datos sensibles como claves API y contraseñas.

Características principales:

- Almacena datos sensibles en un archivo ".env".
- Facilita la separación de configuraciones en diferentes entornos (desarrollo, producción).

Uso básico:

```
require("dotenv").config();

console.log("Clave API:", process.env.API_KEY);
```

Qué protege:

- Prevención de exposición de datos sensibles en el código fuente.

2. Buenas prácticas en seguridad

2.1. Protección de la autenticación:

2.1.1 Almacenamiento seguro de contraseñas:

✓ Usa bcrypt u otra biblioteca de hashing con sal para almacenar contraseñas.

✓ No almacenes contraseñas en texto plano.

```
const bcrypt = require("bcrypt");
const hashedPassword = await bcrypt.hash("miContraseña", 10);
```

2.1.2. Implementa autenticación basada en tokens

✓ Usa JWT (Json Web Tokens) o OAuth para manejar sesiones de usuario.

✓ Establece una fecha de expiración para los tokens y almacena los Refresh Tokens de forma segura.

2. Buenas prácticas en seguridad

2.1.3 Protege los endpoints de autenticación

✓ Limita los intentos de inicio de sesión con herramientas como express-rate-limit.

✓ Implementa CAPTCHAs para prevenir bots.

```
const rateLimit = require("express-rate-limit");
app.use("/login", rateLimit({ windowMs: 15 * 60 * 1000, max: 5 }));
```

2.2 Gestión de datos sensibles

2.2.1. Configuración de variables de entorno

✓ Usa dotenv para manejar claves API, contraseñas de bases de datos y secretos de JWT.

```
require("dotenv").config();
const secretKey = process.env.SECRET_KEY;
```

EDUCACIÓN CONTINUA PUCP
Facultad de Ciencias e Ingeniería

2. Buenas prácticas en seguridad

2.2.2 Cifra datos sensibles

✓ Usa bibliotecas como crypto para cifrar datos importantes antes de almacenarlos.

```
const crypto = require("crypto");
const encryptedData = crypto.createCipher("aes-256-cbc", secretKey).update("data", "utf8",
```

2.3. Validación de entrada

2.3.1 Sanitiza y valida datos de usuario

✓ Usa bibliotecas como Joi o validator.js para validar y sanitizar entradas.

```
const Joi = require("joi");
const schema = Joi.object({ email: Joi.string().email().required() });
const { error } = schema.validate({ email: "example@domain.com" });
if (error) throw new Error(error.message);
```

2.3.2 Evita inyecciones SQL

✓ Usa consultas parametrizadas o bibliotecas ORM como Sequelize o TypeORM.

```
const { Pool } = require("pg");
const pool = new Pool();
await pool.query("SELECT * FROM users WHERE id = $1", [userId]);
```

2. Buenas prácticas en seguridad

2.4. Configuración del servidor

2.4.1 Usa HTTPS

✓ Asegúrate de que tu servidor utiliza HTTPS para proteger los datos en tránsito.

✓ Usa Let's Encrypt o servicios similares para certificados SSL gratuitos.

2.4.2 Configura cabeceras HTTP seguras

✓ Usa Helmet para prevenir ataques comunes como XSS y clickjacking.

```
const helmet = require("helmet");
app.use(helmet());
```

2.4.3 Configura CORS adecuadamente

✓ Restringe los orígenes permitidos con la biblioteca cors.

```
const cors = require("cors");
app.use(cors({ origin: "https://example.com" }));
```

2. Buenas prácticas en seguridad

2.5. Protección contra ataques

2.5.1 Evita ataques de fuerza bruta y DoS

✓ Limita las solicitudes con express-rate-limit.

✓ Usa herramientas como ddos para mitigar ataques de denegación de servicio.

2.5.2 Prevén Cross-Site Scripting (XSS)

✓ Escapa los datos antes de renderizarlos en el frontend.

✓ Implementa una política de seguridad de contenido (CSP) con Helmet.

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

2.5.3 Prevén Cross-Site Request Forgery (CSRF)

✓ Usa bibliotecas como csurf para proteger formularios contra ataques CSRF.

```
const csurf = require("csurf");
app.use(csurf());
```

2. Buenas prácticas en seguridad

2.6. Política de acceso y permiso

2.6.1 Implementa control de acceso basado en roles

✓ Usa middleware para verificar permisos.

```
const authorize = (role) => {
  return (req, res, next) => {
    if (req.user.role !== role) {
      return res.status(403).send("Acceso denegado");
    }
    next();
  };
};
app.get("/admin", authorize("admin"), (req, res) => {
  res.send("Bienvenido, admin");
});
```

2.6.2 Limita los permisos del servidor

✓ Asegúrate de que tu servidor y archivos tengan los permisos mínimos necesarios para operar.

EDUCACIÓN CONTINUA PUCP

Facultad de Ciencias e Ingeniería

¡Excelente,
seguimos avanzando!

Tema: Seguridad en desarrollo Web

Docente: Jean Paul Curiñaupa Taype

Programa **Desarrollo Web, Front y Back End**
Curso Pruebas, Seguridad y DevOps