

# Consigna práctica 2: Mitigación de Vulnerabilidades de CWE

---

Trabajo realizado por Juan Rodriguez, Juan Cama y Gastón Bauer.

Link al repositorio: <https://github.com/ignrdz2/DSS-P2.git>

## 1. Inyección SQL (SQLi)

Prueba de Concepto (PoC)

**Paso 1:** Se obtiene un token de autenticación usando el usuario de prueba.

```
curl -X POST http://localhost:3000/auth/login \
  -H "Content-Type: application/json" \
  -d '{"username":"test","password":"password"}'
```

**Respuesta esperada:**

```
{
  "token": "<TOKEN>",
  "user": { ... }
}
```

**Paso 2:** Se realiza una petición maliciosa para explotar la inyección SQL.

```
curl -H "Authorization: Bearer <TOKEN>" \
  "http://localhost:3000/invoices?status=paid&operator=or%201%3D1%20--"
```

**Respuesta obtenida:**

```
{
  "message": "select * from \"invoices\" where \"userId\" = $1 and status or 1=1
-- 'paid' - argument of AND must be type boolean, not type character varying"
}
```

**Explicación:**

- El error recibido demuestra que el parámetro **operator** se inserta directamente en la consulta SQL.
- La base de datos rechaza la consulta por tipo de datos, pero la vulnerabilidad existe y puede ser explotada.

- Si la consulta fuera aceptada, se mostrarían todas las facturas, ignorando el filtro por estado.

### Evidencia 1:

```
(kali@kali)-[~/Desktop/DSS-P2]
$ curl -X POST http://localhost:3000/auth/login \
  -H "Content-Type: application/json" \
  -d '{"username":"test","password":"password"}'
{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzU4NjQzND
AxLCJleHAiOiE3NTg2NDcwMDF9.Eyth9J4qeDM6edhdm-OzT_pDfm44pkLUwCjWpU4iApU","user
":{"id":1,"username":"test","email":"test@example.local","password":"password
","first_name":"Test","last_name":"User","activated":true,"reset_password_tok
en":null,"reset_password_expires":null,"invite_token":null,"invite_token_expi
res":null,"picture_path":null}}
```

### Evidencia 2:

```
(kali@kali)-[~/Desktop/DSS-P2]
$ curl -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZ
CI6MSwiaWF0IjoxNzU4NjQzNDAxLCJleHAiOiE3NTg2NDcwMDF9.Eyth9J4qeDM6edhdm-OzT_pDf
m44pkLUwCjWpU4iApU" \
  "http://localhost:3000/invoices?status=paid&operator=or%201%3D1%20--"
{"message":"select * from \"invoices\" where \"userId\" = $1 and status or 1
=1 -- 'paid' - argument of AND must be type boolean, not type character varyi
ng"}
```

### Mitigación

```
static async list( userId: string, status?: string, operator?: string):
Promise<Invoice[]> {
  let q = db<InvoiceRow>('invoices').where({ userId: userId });
  if (status) {
    // Mitigación: Validar operador y usar parámetros
    const allowedOps = ['=', '!=', '<', '>', '<=', '>='];
    const op = allowedOps.includes(operator || '') ? operator : '=';
    q = q.andWhere('status', op, status);
  }
  const rows = await q.select();
}
```

### Prueba tras la mitigación

#### Comando ejecutado:

```
curl -H "Authorization: <TOKEN>" \
  "http://localhost:3000/invoices?status=paid&operator=or%201%3D1%20--"
```

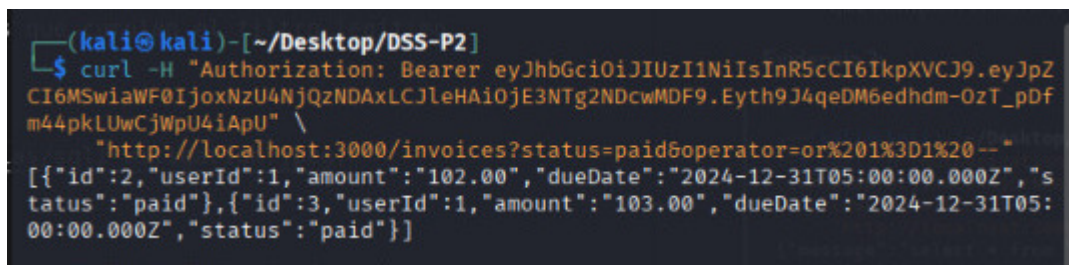
#### Respuesta obtenida tras la mitigación:

```
[
  {
    "id": 2,
    "userId": 1,
    "amount": "102.00",
    "dueDate": "2024-12-31T05:00:00.000Z",
    "status": "paid"
  },
  {
    "id": 3,
    "userId": 1,
    "amount": "103.00",
    "dueDate": "2024-12-31T05:00:00.000Z",
    "status": "paid"
  }
]
```

### Explicación:

- Ahora, aunque se intente inyectar código SQL en el parámetro `operator`, la consulta solo acepta operadores válidos y filtra correctamente por estado.
- La vulnerabilidad queda mitigada y el sistema responde únicamente con las facturas que cumplen el filtro legítimo.

### Evidencia:



```
(kali@kali)-[~/Desktop/DSS-P2]
$ curl -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6ImSWiaWF0IjoxNzU4NjQzNDExLCJleHAiOjE3NTg2NDcwMDY5.Eyth9J4qeDM6edhdm-OzT_pDf m44pkLUwCjWpU4iApU" \
  "http://localhost:3000/invoices?status=paid&operator=or%201%3D1%20--"
[{"id":2,"userId":1,"amount":"102.00","dueDate":"2024-12-31T05:00:00.000Z","status":"paid"}, {"id":3,"userId":1,"amount":"103.00","dueDate":"2024-12-31T05:00:00.000Z","status":"paid"}]
```

## 2. Credenciales Embedidas (Hard Coded Credentials)

### Prueba de Concepto (PoC)

**Paso 1:** Acceder al código fuente y localizar el secreto JWT embebido:

```
jwt.sign({ id: userId }, "secreto_super_seguro", { expiresIn: "1h" });
```

**Paso 2:** Generar un token JWT válido manualmente usando Node.js:

Crear un archivo `jwt_poc.js` con el siguiente contenido:

```
const jwt = require("jsonwebtoken");
const token = jwt.sign({ id: 1 }, "secreto_super_seguro", { expiresIn: "1h" });
```

```
console.log(token);
```

Instalar la librería:

```
npm install jsonwebtoken
```

Ejecutar el script:

```
node jwt_poc.js
```

**Token generado:**

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzU4NjQ2MjI2LCJleHAiOjE3NTg2NDk4MjZ9.s7YWKpQKKbTFBboFc5v7Dj-FKRnsLK7Ah15HFEkYvhw
```

**Paso 3:** Usar el token generado para acceder a una ruta protegida:

```
curl -H "Authorization: Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzU4NjQ2MjI2LCJleHAiOjE3NTg2NDk4MjZ9.s7YWKpQKKbTFBboFc5v7Dj-FKRnsLK7Ah15HFEkYvhw" \  
http://localhost:3000/invoices
```

**Respuesta obtenida:**

```
[  
  {  
    "id": 1,  
    "userId": 1,  
    "amount": "101.00",  
    "dueDate": "2024-12-31T05:00:00.000Z",  
    "status": "unpaid"  
  },  
  {  
    "id": 2,  
    "userId": 1,  
    "amount": "102.00",  
    "dueDate": "2024-12-31T05:00:00.000Z",  
    "status": "paid"  
  },  
  {  
    "id": 3,  
    "userId": 1,  
    "amount": "103.00",
```

```
"dueDate": "2024-12-31T05:00:00.000Z",  
"status": "paid"  
}  
]
```

### Explicación:

- El backend acepta el token generado manualmente usando el secreto embebido, permitiendo acceso no autorizado.
- Esto demuestra la vulnerabilidad de credenciales embebidas.

### Evidencia:

The screenshot shows a terminal window with the following commands and output:

```
home > kali > Desktop > JS jwt_poc.js > ...  
1 const jwt = require('jsonwebtoken');  
2 const token = jwt.sign({ id: 1 }, "secreto_super_seguro", { expiresIn: '1h' });  
3 console.log(token);  
  
kali@kali: ~/Desktop  
File Actions Edit View Help  
-(kali@kali)-[~/Desktop]  
$ npm install jsonwebtoken  
added 15 packages in 910ms  
1 package is looking for funding  
run 'npm fund' for details  
  
-(kali@kali)-[~/Desktop]  
$ node jwt_poc.js  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzU4NjQ2MjI2LCJleHAiOi0jE3NTg2NDk4MjZ9.s7YWKpQKKbTFBboFc5v7Dj~FKRnsLK7Ah15HFEkYvhw  
  
-(kali@kali)-[~/Desktop]  
$  
  
-(kali@kali)-[~/Desktop]  
$ curl -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzU4NjQ2MjI2LCJleHAiOi0jE3NTg2NDk4MjZ9.s7YWKpQKKbTFBboFc5v7Dj~FKRnsLK7Ah15HFEkYvhw" \\  
http://localhost:3000/invoices  
[{"id":1,"userId":1,"amount":"101.00","dueDate":"2024-12-31T05:00:00.000Z","status":"unpaid"}, {"id":2,"userId":1,"amount":"102.00","dueDate":"2024-12-31T05:00:00.000Z","status":"paid"}, {"id":3,"userId":1,"amount":"103.00","dueDate":"2024-12-31T05:00:00.000Z","status":"paid"}]
```

### Mitigación

#### Código antes:

```
const generateToken = (userId: string) => {  
  return jwt.sign({ id: userId }, "secreto_super_seguro", { expiresIn: "1h" });  
};  
  
const verifyToken = (token: string) => {  
  return jwt.verify(token, "secreto_super_seguro");  
};
```

**Código después:**

```
const JWT_SECRET = process.env.JWT_SECRET || "default_secret";

const generateToken = (userId: string) => {
  return jwt.sign({ id: userId }, JWT_SECRET, { expiresIn: "1h" });
};

const verifyToken = (token: string) => {
  return jwt.verify(token, JWT_SECRET);
};
```

**Explicación:**

- El secreto JWT ya no está embebido en el código fuente.
- Ahora se toma desde una variable de entorno (`JWT_SECRET`), lo que permite cambiarlo sin modificar el código y mantenerlo fuera del repositorio.
- Esto mitiga la vulnerabilidad, ya que un atacante no puede obtener el secreto simplemente accediendo al código fuente.

**Configuración adicional:** Agregar la variable en el archivo `.env`:

```
JWT_SECRET=supersecretkey
```

### 3. Falsificación de Peticiones del Lado del Servidor (SSRF)

#### Prueba de Concepto (PoC)

**Paso 1:** Iniciar un servidor local para recibir la petición:

```
python3 -m http.server 8000
```

**Paso 2:** Realizar una petición maliciosa al backend:

```
curl -X POST http://localhost:3000/invoices/1/pay \
  -H "Authorization: Bearer <TOKEN>" \
  -H "Content-Type: application/json" \
  -d
'{"paymentBrand": "localhost:8000", "ccNumber": "4111111111111111", "ccv": "123", "expirationDate": "12/25"}'
```

**Evidencia en el servidor Python:**

```
127.0.0.1 - - [23/Sep/2025 13:11:50] code 501, message Unsupported method ('POST')
127.0.0.1 - - [23/Sep/2025 13:11:50] "POST /payments HTTP/1.1" 501 -
```

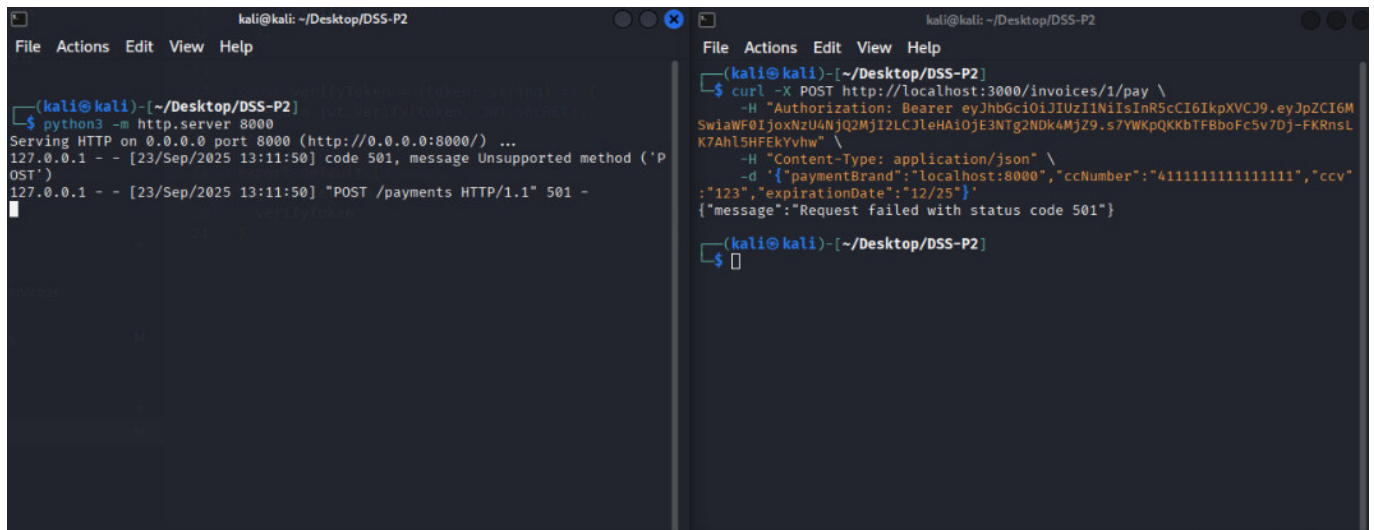
### Respuesta obtenida en curl:

```
{ "message": "Request failed with status code 501" }
```

### Explicación:

- El backend realizó una petición HTTP a la dirección indicada en el parámetro `paymentBrand`, demostrando la vulnerabilidad SSRF.
- El servidor Python recibió la petición, aunque respondió con error 501, lo que confirma que el backend puede ser manipulado para hacer peticiones arbitrarias.

### Evidencia:



### Mitigación

#### Código antes:

```
const paymentResponse = await axios.post(`http://${paymentBrand}/payments`, {
  ccNumber,
  ccv,
  expirationDate,
});
```

#### Código después:

```
const allowedBrands = ["visa", "mastercard"];
if (!allowedBrands.includes(paymentBrand)) {
  throw new Error("Invalid payment brand");
}
```

```
}  
const paymentResponse = await axios.post(`http://${paymentBrand}/payments`, {  
  ccNumber,  
  ccv,  
  expirationDate,  
});
```

### Explicación:

- Ahora solo se permite que `paymentBrand` sea un valor específico y legítimo (por ejemplo, 'visa' o 'mastercard').
- Se evita que el backend realice peticiones HTTP arbitrarias a direcciones controladas por el usuario.
- Esto elimina la vulnerabilidad SSRF y protege el sistema contra ataques de red internos y externos.

### Configuración adicional:

- Si se requiere agregar más marcas legítimas, solo deben añadirse a la lista `allowedBrands`.

### Prueba tras la mitigación

### Comando ejecutado:

```
curl -X POST http://localhost:3000/invoices/1/pay \  
  -H "Authorization: Bearer <TOKEN>" \  
  -H "Content-Type: application/json" \  
  -d '  
    {"paymentBrand": "localhost:8000", "ccNumber": "4111111111111111", "ccv": "123", "expir  
    ationDate": "12/25"}'
```

### Respuesta obtenida tras la mitigación:

```
{ "message": "Invalid payment brand" }
```

### Explicación:

- El backend rechaza valores no permitidos en el parámetro `paymentBrand` y no realiza ninguna petición externa.
- La vulnerabilidad SSRF queda mitigada correctamente.

### Evidencia:



```
(kali@kali)-[~/Desktop/DSS-P2]
$ curl -X POST http://localhost:3000/invoices/1/pay \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6M  
SwiaWF0IjoxNzU4NjQ2MjI2LCJleHAiOjE3NTg2NDk4MjZ9.s7YWKpQKKbTFBboFc5v7Dj-FKRnsL  
K7AhL5HFEkYvhw" \
  -H "Content-Type: application/json" \
  -d '{"paymentBrand": "localhost:8000", "ccNumber": "4111111111111111", "ccv"  
: "123", "expirationDate": "12/25"}'
{"message": "Invalid payment brand"}
```

## 4. Recorrido de directorios (Path Traversal)

### Prueba de Concepto (PoC)

**Paso 1:** Realizar una petición maliciosa para intentar acceder a archivos fuera del directorio permitido:

```
curl -H "Authorization: Bearer <TOKEN>" \
  "http://localhost:3000/invoices/1/invoice?pdfName=sample-invoice.pdf"
```

### Respuesta obtenida:

```
{ "message": "Receipt not found" }
```

### Explicación:

- El backend intenta acceder a la ruta proporcionada por el usuario sin validación.
- Aunque el archivo solicitado no se encontró, el comportamiento demuestra que el backend es vulnerable a Path Traversal, ya que intenta acceder a rutas arbitrarias.
- Si el archivo existiera y el usuario del proceso tuviera permisos, el backend podría exponer información sensible.

### Evidencia:

```
(kali@kali)-[~/Desktop/DSS-P2]
$ curl -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6M  
CI6MSwiaWF0IjoxNzU4NjYzMTE1LCJleHAiOjE3NTg2NjY3MTV9.fRz_8t2atE2GgojRlky8Wa  
fgP9pswybXYesde7ic" \
  "http://localhost:3000/invoices/1/invoice?pdfName=sample-invoice.pdf"
{"message": "Receipt not found"}
```

### Mitigación

#### Código antes:

```
try {
  const filePath = `/invoices/${pdfName}`;
  const content = await fs.readFile(filePath, "utf-8");
  return content;
}
```

```
} catch (error) {  
  console.error("Error reading receipt file:", error);  
  throw new Error("Receipt not found");  
}
```

### Código después:

```
try {  
  // Mitigación: Validar nombre de archivo  
  if (!/^[\\w\\-\\.]+\\.pdf$/i.test(pdfName)) {  
    throw new Error("Invalid file name");  
  }  
  const filePath = `/invoices/${pdfName}`;  
  const content = await fs.readFile(filePath, "utf-8");  
  return content;  
} catch (error) {  
  console.error("Error reading receipt file:", error);  
  throw new Error("Receipt not found");  
}
```

### Explicación:

- Antes, el backend intentaba leer cualquier archivo indicado por el usuario, sin validar el nombre.
- Después, se valida que el nombre del archivo sea seguro (solo PDFs con nombre válido) antes de intentar leerlo, mitigando el Path Traversal.
- Si el nombre no es válido, se lanza un error específico y no se accede a rutas arbitrarias.

### Prueba tras la mitigación:

```
curl -H "Authorization: <TOKEN>" \  
  "http://localhost:3000/invoices/1/invoice?pdfName=../../../../etc/passwd"
```

### Respuesta obtenida tras la mitigación:

```
{ "message": "Invalid file name" }
```

### Explicación:

- El backend ahora valida correctamente el nombre del archivo antes de intentar leerlo.
- Si el nombre no cumple el patrón seguro, responde con un mensaje claro y no accede a rutas arbitrarias.
- La vulnerabilidad de Path Traversal queda mitigada.

### Evidencia:

```
(kali㉿kali)-[~/Desktop/DSS-P2]
$ curl -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzU0NjYzMTE1L3JleHAiOjE3NTg2NjY3MTV9.fRz_8t2atE2GgojRlky8WarrfgP9pswybxYesde7ic" \
    "http://localhost:3000/invoices/1/invoice?pdfName=../..../etc/passwd"
{"message":"Invalid file name"}
```

## 5. Falta de autorización (Missing Authorization)

## Prueba de Concepto (PoC)

**Paso 1:** Autenticarse como usuario A y obtener el token JWT.

**Paso 2:** Realizar una petición para modificar los datos de otro usuario (usuario con id=2) usando el token de usuario A.

**Comando:**

```
curl -X PUT http://localhost:3000/users/2 \
-H "Authorization: Bearer <TOKEN_USUARIO_A>" \
-H "Content-Type: application/json" \
-d
'{"username":"hackeado","email":"hack@evil.com","first_name":"Mal","last_name":"Ac
tor","password":"1234"}'
```

**Respuesta obtenida (vulnerable):**

```
{
  "id": 2,
  "username": "prod",
  "email": "prod@example.local",
  "password": "password",
  "first_name": "Prod",
  "last_name": "User",
  "activated": true,
  "reset_password_token": null,
  "reset_password_expires": null,
  "invite_token": null,
  "invite_token_expires": null,
  "picture_path": null
}
```

## Evidencia

```
(kali@kali)~[~/Desktop/DSS-P2]
$ curl -X PUT http://localhost:3000/users/2 \
-H "Authorization: Bearer yJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzU4NjYzMTE1LCJleHAiOiJlE3NTg2NjY3MTV9.fRz_8t2atE2GgojRlky8WarrfgP9pswybxYesde7ic" \
-H "Content-Type: application/json" \
-d '{"username":"hackeado","email":"hack@evil.com","first_name":"Mal","last_name":"Actor","password":"1234"}'
{"id":2,"username":"prod","email":"prod@example.local","password":"password","first_name":"Prod","last_name":"User","activated":true,"reset_password_token":null,"reset_password_expires":null,"invite_token":null,"invite_token_expires":null,"picture_path":null}
```

**Interpretación:** El usuario autenticado pudo modificar los datos de otro usuario (id=2) sin ser el propietario, demostrando la vulnerabilidad de falta de autorización.

---

## Mitigación

**Código antes:** No se verificaba que el usuario autenticado fuera el dueño del recurso a modificar.

**Código después:**

```
const updateUser = async (req: Request, res: Response, next: NextFunction) => {
  const userId = req.params.id;
  // Verificación de autorización
  const authUserId = (req as any).user.id;
  if (authUserId !== userId) {
    return res.status(403).json({ message: "Forbidden: cannot modify other users"
  });
}
};
```

**Además:** Se protegió la ruta `PUT /users/:id` con el middleware de autenticación:

```
router.put("/:id", authenticateJWT, routes.updateUser);
```

**Explicación:**

- Ahora solo el usuario autenticado puede modificar sus propios datos.
  - Si intenta modificar otro usuario, recibe un error 403 Forbidden.
  - La ruta está protegida y el backend responde correctamente en todos los casos.
- 

## Evidencia

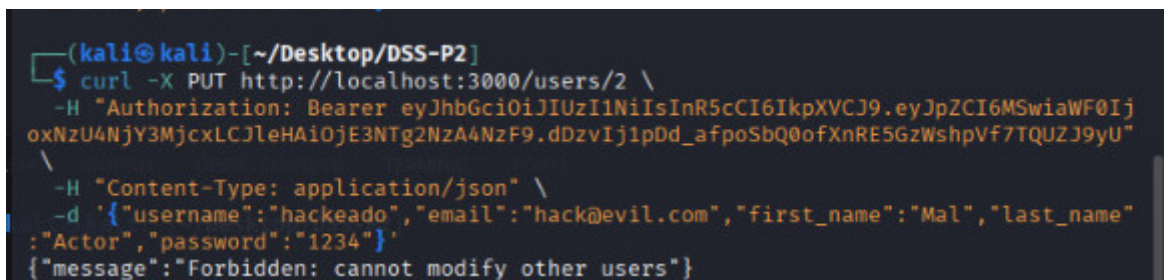
**Comando ejecutado tras la mitigación:**

```
curl -X PUT http://localhost:3000/users/2 \
-H "Authorization: Bearer <TOKEN_USUARIO_A>" \
```

```
-H "Content-Type: application/json" \
-d
'{"username":"hackeado","email":"hack@evil.com","first_name":"Mal","last_name":"Actor","password":"1234"}'
```

### Respuesta obtenida tras la mitigación:

```
{ "message": "Forbidden: cannot modify other users" }
```



```
(kali@kali)-[~/Desktop/DSS-P2]
$ curl -X PUT http://localhost:3000/users/2 \
-H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzU4NjY3MjcxCjleHAiOiJlE3NTg2NzA4NzF9.dDzvIj1pDd_afpoSbQ0ofXnRE5GzWshpVf7TQUZJ9yU" \
-H "Content-Type: application/json" \
-d '{"username":"hackeado","email":"hack@evil.com","first_name":"Mal","last_name":"Actor","password":"1234"}'
{"message":"Forbidden: cannot modify other users"}
```

### Evidencia:

## 6. Inyección de comandos en plantillas (Template Command Injection)

### Prueba de Concepto (PoC)

**Contexto técnico:** En `authService.ts`, el backend genera el HTML del email de activación así:

```
const template = `
<html>
  <body>
    <h1>Hello ${user.first_name} ${user.last_name}</h1>
    <p>Click <a href="${link}">here</a> to activate your account.</p>
  </body>
</html>`;
const htmlBody = ejs.render(template);
```

Los datos del usuario se insertan sin sanitización, permitiendo inyección de código EJS.

### PoC:

1. Registrar un usuario con un nombre malicioso:

```
curl -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
-d
'{"username":"evilunique","password":"1234","email":"evilunique@example.com","first_name":"<%= global.process.env %>","last_name":"Hacker"}'
```

2. Revisar el código fuente y el flujo de generación del email. El campo `first_name` se inserta en la plantilla y es procesado por EJS, lo que puede ejecutar código y exponer variables internas.

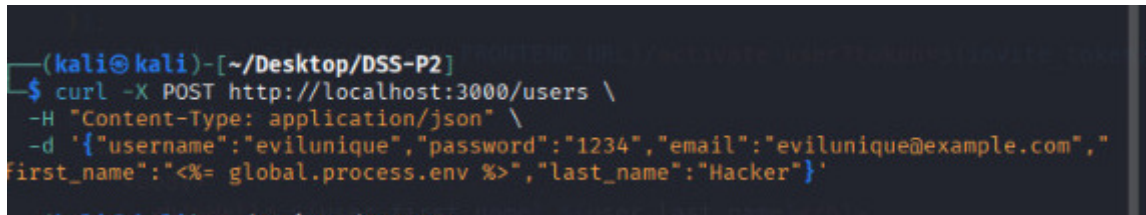
**Explicación:** El código vulnerable permite que cualquier dato enviado por el usuario sea interpretado como código EJS. Esto puede exponer variables de entorno, rutas, o ejecutar comandos si el flujo de envío de email está funcionando.

#### Evidencia:

- Fragmento de código vulnerable:

```
<h1>
  Hello ${user.first_name} ${user.last_name}
</h1>
```

- Comando para crear usuario malicioso:



```
(kali@kali)-[~/Desktop/DSS-P2]
$ curl -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
-d '{"username":"evilunique","password":"1234","email":"evilunique@example.com","first_name":"<%= global.process.env %>","last_name":"Hacker"}'
```

- Explicación: Cuando se muestre el HTML del email de activación, el campo `first_name` sería interpretado y podría mostrar información interna.

## Mitigación

#### Código antes:

```
const template = `
  <html>
    <body>
      <h1>Hello ${user.first_name} ${user.last_name}</h1>
      <p>Click <a href="${link}">here</a> to activate your account.</p>
    </body>
  </html>`;
const htmlBody = ejs.render(template);
```

#### Código después:

```
const escape = (str) =>
  str.replace(
    /[&<>'"/`]/g,
    (c) =>
      ({
        "&": "&amp;",
```

```
    "<": "&lt;",
    ">": "&gt;",
    "'": "&quot;",
    '"': "&#39;",
    "`": "&#96;",
  }[c])
);
const safeFirstName = escape(user.first_name);
const safeLastName = escape(user.last_name);

const template = `
  <html>
    <body>
      <h1>Hello ${safeFirstName} ${safeLastName}</h1>
      <p>Click <a href="${link}">here</a> to activate your account.</p>
    </body>
  </html>`;
const htmlBody = ejs.render(template);
```

### Explicación:

- Se sanitizan los datos antes de insertarlos en la plantilla, reemplazando caracteres especiales por sus entidades HTML.
- Así, cualquier dato malicioso enviado por el usuario se mostrará como texto plano y no será interpretado ni ejecutado por EJS.
- Esta mitigación es efectiva y recomendada para evitar la ejecución de comandos o acceso a variables internas en plantillas dinámicas.

## 7. Almacenamiento inseguro

### Prueba de Concepto (PoC)

**Paso 1:** Registrar un usuario con contraseña conocida.

```
curl -X POST http://localhost:3000/auth/register \
-H "Content-Type: application/json" \
-d '{
  "username": "testuser",
  "email": "testuser@example.com",
  "password": "MiPassword123",
  "first_name": "Test",
  "last_name": "User"
}'
```

**Paso 2:** Consultar la base de datos para verificar cómo se almacena la contraseña.

```
docker exec -it postgres psql -U user -d jwt_api
```



Luego, en la consola de psql:

```
SELECT username, email, password FROM users WHERE username='testuser';
```

### Respuesta obtenida:

username	email	password
testuser	testuser@example.com	MiPassword123

### Explicación:

- La contraseña se almacena en texto plano en la base de datos.
- Un atacante con acceso a la base de datos puede obtener todas las contraseñas fácilmente.

### Evidencia:

```
(kali@kali)-[~/Desktop/DSS-P2]
$ curl -X POST http://localhost:3000/auth/register \
-H "Content-Type: application/json" \
-d '{
  "username": "testuser",
  "email": "testuser@example.com",
  "password": "MiPassword123",
  "first_name": "Test",
  "last_name": "User"
}'

(kali@kali)-[~/Desktop/DSS-P2]
$ docker exec -it postgres psql -U user -d jwt_api
psql (17.6 (Debian 17.6-1.pgdg13+1))
Type "help" for help.

jwt_api=# SELECT username, email, password FROM users WHERE username='testuser';
 username |          email          | password
-----+-----+-----
 testuser | testuser@example.com    | MiPassword123
(1 row)

jwt_api=#
```

## Mitigación

**Paso 1:** Se modificó el backend para que las contraseñas se almacenen usando un hash seguro (bcrypt) en vez de texto plano.

### Código antes:

```
await db<UserRow>("users").insert({
  username: user.username,
```



```
password: user.password, // texto plano
});
```

### Código después:


```
import bcrypt from "bcrypt";
const hashedPassword = await bcrypt.hash(user.password, 10);
await db<UserRow>("users").insert({
  username: user.username,
  password: hashedPassword, // hash seguro
});
```

**Verificación tras la mitigación:** Al registrar un usuario y consultar la base de datos:

```
SELECT username, email, password FROM users WHERE username='testuser';
```

La columna `password` ahora contiene un hash, no la contraseña original.

### Evidencia:



The screenshot shows a terminal window on a Kali Linux machine. The first command is a curl POST request to `http://localhost:3000/auth/register` with a JSON body containing user details. The second command is a database query to retrieve the user's record. The output shows the password is stored as a bcrypt hash.

```
(kali@kali)-[~/Desktop/DSS-P2]
$ curl -X POST http://localhost:3000/auth/register \
-H "Content-Type: application/json" \
-d '{
  "username": "testuser3",
  "email": "testuser3@example.com",
  "password": "OtraPassword123",
  "first_name": "Test3",
  "last_name": "User3"
}'

(kali@kali)-[~/Desktop/DSS-P2]
$
```

username	email	password
testuser3	testuser3@example.com	\$2b\$10\$D85Ams0UzagivzLGQagVK..DqzWla4t6gQ2bHIzgm292d57xlrQgG

(1 row)

### Explicación:

- Las contraseñas ya no se almacenan en texto plano, sino como hashes seguros.
- Si un atacante accede a la base de datos, no podrá recuperar las contraseñas originales.
- El sistema sigue permitiendo el login y el cambio de contraseña, pero siempre usando el hash.