🛠️ Ejercicio Práctico

📌 **Título:** Identificación de vulnerabilidades en una API RESTful simulada

---

🎯 Objetivo:

Analizar una API RESTful básica utilizando una herramienta como **Postman** o **cURL**, para identificar posibles **fallas de autenticación, autorización y validación de entrada**, y proponer medidas de mitigación apropiadas.

🧱 Escenario:

Estás evaluando una API RESTful de prueba que gestiona usuarios en el siguiente endpoint:
`http://localhost:3000/api/users`

La API permite las siguientes operaciones sin necesidad de autenticación:

- `GET /api/users` → Lista todos los usuarios
- `POST /api/users` → Crea un nuevo usuario
- `DELETE /api/users/1` → Elimina el usuario con ID 1

🧪 Actividades:

✅ Parte 1 – Análisis de Acceso

1. Envía una solicitud **GET** a `/api/users` sin ningún token o autenticación.
    - ¿Recibes los datos de todos los usuarios?

- ¿Puedes acceder a información sensible como correos electrónicos o contraseñas?

✅ Parte 2 – Prueba de Autorización

2. Intenta ejecutar una solicitud **DELETE** para eliminar un usuario, sin autenticación:
    - ¿La acción se ejecuta exitosamente?
    - ¿Deberías poder eliminar usuarios sin ser administrador?

✅ Parte 3 – Prueba de Validación de Entradas

3. Ejecuta un **POST** con datos inválidos, por ejemplo:

```
{
  "username": "<script>alert('XSS')</script>",
  "email": "notanemail"
}
```

- ¿La API acepta los datos sin validación?
- ¿Hay algún filtro o sanitización?

📋 Entregables:

1. Capturas de pantalla de cada prueba (GET, POST, DELETE).
2. Observaciones sobre las respuestas de la API.
3. Identificación de al menos **2 vulnerabilidades**.
4. Propuesta de al menos **2 medidas de seguridad** para mitigar las fallas detectadas.

🧠 Preguntas de reflexión:

- ¿Qué riesgos existen si una API no controla quién puede acceder o modificar sus datos?
- ¿Por qué es importante validar cada dato enviado a una API, incluso si proviene de un “cliente confiable”?
- ¿Cómo protegerías esta API usando estándares como JWT, roles de usuario y validación de esquema?

✅ Solución Modelo – Ejercicio Práctico

📌 Evaluación de Seguridad en una API RESTful Simulada

🔍 Parte 1 – Análisis de Acceso (GET /api/users)

📥 Solicitud:
GET http://localhost:3000/api/users

✅ Resultado observado:

- La API devolvió una lista completa de usuarios sin requerir autenticación.
- Los datos incluían campos sensibles como:

```
[
  {
    "id": 1,
    "username": "admin",
    "email": "admin@example.com",
    "password": "123456"
  },
  ...
]
```

⚠️ Observación:

- **Grave fallo de confidencialidad**: Exposición de contraseñas sin cifrado.
- No se aplica ningún mecanismo de autenticación o control de acceso.

🔍 Parte 2 – Prueba de Autorización (DELETE /api/users/1)

📥 Solicitud:
DELETE http://localhost:3000/api/users/1

✅ Resultado observado:

- El usuario fue eliminado exitosamente sin ningún tipo de autenticación ni token.

⚠️ Observación:

- **Fallo de autorización crítica**: Cualquier usuario puede eliminar recursos sin privilegios.
- No se implementa control de roles (ej. solo administradores deberían tener ese permiso).

🔍 Parte 3 – Prueba de Validación de Entradas (POST /api/users)

📥 Payload enviado:

```
{
  "username": "<script>alert('XSS')</script>",
  "email": "notanemail"
}
```

✅ Resultado observado:

- La API aceptó los datos sin validación.
- No hubo rechazo del email mal formado ni sanitización del campo `username`.

⚠️ Observación:

- **Riesgo de XSS almacenado** si estos datos son mostrados en el frontend.
- **Falta de validación de formato y tipo de datos** (no se valida email).

📊 Vulnerabilidades identificadas

Tipo de Falla	Descripción
Autenticación ausente	No se exige ningún tipo de token o login
Autorización rota	Se permite eliminar usuarios sin privilegios
Falta de validación	Entrada con JavaScript malicioso y emails falsos
Exposición de datos	Muestra contraseñas en texto plano

🛡️ Recomendaciones de mitigación

1. **Implementar Autenticación con JWT o API Keys**
    - Todo endpoint debe requerir verificación de identidad.
2. **Aplicar Autorización basada en roles (RBAC)**
    - Solo usuarios autorizados pueden acceder a acciones críticas (como DELETE).
3. **Validación y Sanitización de Entradas**
    - Validar formatos (`email`, `string`, `number`) y bloquear scripts maliciosos.
    - Usar bibliotecas como **Joi**, **Yup** o **class-validator**.
4. **Nunca exponer contraseñas en respuestas**
    - Cifrar con **bcrypt** y omitir del payload de respuesta.
5. **Documentar los endpoints con Swagger u OpenAPI**
    - Indicar claramente los requisitos de seguridad por ruta.

📘 Conclusión

Esta práctica reveló cómo una API mal protegida puede convertirse en un punto de entrada crítico para atacantes. La seguridad no puede ser un añadido posterior: **debe ser parte del diseño de la API desde el inicio**. Validar,

autenticar, autorizar y restringir son pilares para cualquier sistema expuesto públicamente.
