

**Objetivos:**

- I. Criação do servidor Node.js e BD;
- II. SQL Injection;
- III. Boas práticas para prevenção de SQL Injection.

**I. Criação do servidor Node.js e BD**

No pgAdmin, crie um banco de dados com o nome **bdaula** (ou outro nome de sua escolha). Em seguida, execute os comandos a seguir para criar a tabela users com registros de teste:

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  username VARCHAR NOT NULL,
  password VARCHAR NOT NULL
);

INSERT INTO users (username, password) VALUES
('admin', '123'),
('root', 'abc');
```

**Configuração do servidor**

A seguir, tem-se os passos para configurar um servidor Node.js utilizando o framework Express e TypeScript:

1. Criar a pasta do projeto
  - Crie uma pasta no seu computador para o projeto. Neste exemplo, usaremos o nome **server**. Evite nomes com espaços ou caracteres especiais.
2. Inicializar o projeto Node.js
  - Abra a pasta no terminal (CMD ou terminal integrado do VS Code);
  - Execute o comando `npm init -y`. O parâmetro **-y** evita perguntas interativas e cria automaticamente o arquivo package.json com configurações padrão;

**3. Instalar o Express**

```
npm i express
```

Esse comando cria a pasta node\_modules e adiciona a dependência no package.json;

```
npm i @types/express -D
```

Instale também as definições de tipo do Express:

O parâmetro **-D** indica que essas dependências são para desenvolvimento (devDependencies);

**4. Configurar o TypeScript**

- Inicialize o arquivo de configuração do TypeScript

```
npx tsc --init
```

**5. Instalar as demais dependências**

- Para carregar as variáveis de ambiente do arquivo .env:

```
npm i dotenv
npm i @types/cors -D
```

- Para acesso ao SGBD PostgreSQL:

```
npm i pg
npm i @types/pg -D
```

## 6. Instalar ferramentas para execução do TS

```
npm i typescript ts-node ts-node-dev -D
```

ts-node: permite executar arquivos TS diretamente no Node.js;

ts-node-dev: monitora alterações nos arquivos .ts e reinicia automaticamente o servidor.

Amplamente usando durante a codificação do projeto;

## 7. Adicionar scripts no package.json

- No arquivo package.json, adicione os seguintes scripts para facilitar a execução do projeto:

```
"scripts": {
  "dev": "ts-node-dev ./src",
  "start": "ts-node ./src"
},
```

## 8. Crie o arquivo .env no raiz do projeto e coloque as variáveis de ambiente:

```
PORT = 3001
BD_HOST = localhost
BD_USER = postgres # Altere conforme o seu usuário
BD_PASSWORD = 123 # Altere conforme a sua senha
BD_DATABASE = bdaula # Nome do banco criado no pgAdmin
BD_PORT = 5432 # Porta padrão do PostgreSQL
```

## 9. Crie o arquivo db.ts, na pasta src do projeto, e coloque o seguinte código para gerar o pool de conexões com o SBGD:

```
import { Pool } from "pg";
import dotenv from "dotenv";

dotenv.config(); // Carrega as variáveis de ambiente do arquivo .env

export default new Pool({
  host: process.env.BD_HOST,
  user: process.env.BD_USER,
  password: process.env.BD_PASSWORD,
  database: process.env.BD_DATABASE,
  port: Number(process.env.BD_PORT),
});
```

## 10. Crie o arquivo src/index.ts e coloque o seguinte código de teste. Nele existem duas rotas para fazer consultas na tabela users:

```
import express, { Request, Response } from "express";
```

```
import dotenv from "dotenv";
import db from "./db";

dotenv.config(); // Carrega as variáveis de ambiente do arquivo .env

const app = express(); // Instancia a aplicação Express

const PORT = process.env.PORT || 3000; // Porta que será usada pelo servidor

// Middleware para permitir o uso de JSON no corpo das requisições
app.use(express.json());

// Inicia o servidor e escuta na porta definida
app.listen(PORT, function () {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});

app.post("/login-inseguro", async function (req: Request, res: Response) {
  try {
    const { username, password } = req.body;

    // Código vulnerável a SQL Injection
    const query = `SELECT * FROM users
      WHERE username = '${username}' AND password = '${password}'`;
    const result = await db.query(query);
    if (result.rowCount !== 0) {
      res.json({ message: "Login bem-sucedido" });
    } else {
      res.status(401).json({ message: "Usuário ou senha inválidos" });
    }
  } catch (e: any) {
    res.status(500).json({ message: "Erro no servidor" });
  }
});

app.post("/login-seguro", async function (req: Request, res: Response) {
  try {
    const { username, password } = req.body;

    // Consultas parametrizadas
    const query = `SELECT * FROM users WHERE username = $1 AND password = $2`;
    const result = await db.query(query, [username, password]);
    if (result.rowCount !== 0) {
      res.json({ message: "Login bem-sucedido" });
    } else {
      res.status(401).json({ message: "Usuário ou senha inválidos" });
    }
  } catch (e: any) {
    res.status(500).json({ message: "Erro no servidor" });
  }
});
```

```

    }
  });

  app.use(function (req: Request, res: Response) {
    res.status(404).json({message: "Rota não encontrada"});
  });

```

## II. SQL Injection

SQL Injection, ou Injeção de SQL, é uma vulnerabilidade de segurança que ocorre quando comandos SQL são inseridos de forma maliciosa em campos de entrada de dados da aplicação, com o objetivo de manipular as consultas feitas ao BD. Essa falha é uma das mais conhecidas e exploradas na área de segurança da informação e frequentemente ocupa posições de destaque em relatórios da OWASP (Open Worldwide Application Security Project).

A origem da vulnerabilidade está na construção dinâmica de comandos SQL a partir de dados fornecidos pelo usuário, sem o devido tratamento ou validação. Quando a aplicação insere diretamente os dados recebidos nas consultas, o atacante pode injetar trechos de comandos SQL arbitrários que alteram a lógica original da consulta.

Exemplos:

Imagine um sistema de login com a seguinte consulta:

```

const query = `SELECT * FROM users WHERE username = '${username}' AND
               password = '${password}'`;

```

Se um atacante inserir `admin' OR '1'='1` como nome de usuário, a consulta se torna:

```

SELECT *
FROM users
WHERE username = 'admin' OR '1'='1' AND password = 'qualquer';

```

Como `'1'='1'` é sempre verdadeiro, o atacante pode acessar o sistema sem credenciais válidas.

Essa falha pode ser explorada para diversos fins, tais como:

- a) Obter acesso não autorizado

Objetivo: acessar o sistema sem conhecer a senha do usuário.

Entrada maliciosa:

```

{
  "username": "admin' --",
  "password": "qualquer"
}

```

Consulta gerada:

```

SELECT * FROM users
WHERE username = 'admin' --' AND password = ''

```

Explicação: o operador `--` comenta o restante da consulta, ignorando a verificação de senha. O atacante acessa a conta do admin com sucesso.

b) Exfiltrar dados sensíveis

Objetivo: obter dados de outros usuários da tabela.

Entrada maliciosa:

```
{
  "username": "' OR '1'='1' --",
  "password": "qualquer"
}
```

Consulta gerada:

```
SELECT * FROM users
WHERE username = ' OR '1'='1' --' AND password = 'qualquer';
```

Explicação: a cláusula `OR '1'='1'` é sempre verdadeira. Isso faz com que todos os registros da tabela `users` sejam retornados, permitindo visualizar logins e senhas armazenadas em texto claro.

c) Modificar ou deletar registros

Objetivo: apagar usuários da base de dados.

Entrada maliciosa:

```
{
  "username": "admin'; DELETE FROM users; --",
  "password": "qualquer"
}
```

Consulta gerada:

```
SELECT * FROM users
WHERE username = 'admin'; DELETE FROM users; --' AND password = 'qualquer';
```

Explicação: essa injeção executa duas instruções: a primeira faz a consulta e a segunda deleta todos os registros da tabela `users`.

d) Executar comandos administrativos.

Objetivo: elevar privilégios, alterar estruturas de banco, ou criar novos usuários.

Entrada maliciosa:

```
{
  "username": "admin'; INSERT INTO users(username,password) VALUES('teste',
  '123'); --",
  "password": "qualquer"
}
```

Consulta gerada:

```
SELECT * FROM users
WHERE username = 'admin'; INSERT INTO users(username,password) VALUES('teste',
'123'); --' AND password = 'qualquer';
```

Explicação: insere um usuário.

- e) Comprometer totalmente o sistema de banco de dados.

Objetivo: executar comandos que comprometem a integridade do sistema ou extraem informações administrativas.

Entrada maliciosa:

```
{  
  "username": "admin"; DROP TABLE IF EXISTS users; --",  
  "password": "qualquer"  
}
```

Consulta gerada:

```
SELECT * FROM users  
WHERE username = 'admin'; DROP TABLE IF EXISTS users; --' AND password =  
'qualquer';
```

Explicação: deleta a tabela users.

A injeção de SQL é considerada uma vulnerabilidade crítica porque:

- É relativamente fácil de ser explorada;
- Os impactos podem ser catastróficos;
- Frequentemente está presente em aplicações legadas ou com validações superficiais.

### III. Boas práticas para prevenção de SQL Injection

#### Utilização de consultas parametrizadas

Uma das formas mais eficazes de evitar ataques de SQL Injection é a utilização de consultas parametrizadas (também conhecidas como consultas preparadas ou *prepared statements*). Essa abordagem consiste em separar o código da consulta SQL dos dados fornecidos pelo usuário, evitando que entradas maliciosas alterem a lógica da instrução SQL.

#### Por que separar código e dados evita ataques?

Quando os valores fornecidos pelo usuário são tratados como parâmetros, o SGBD os interpreta apenas como dados — e não como parte da lógica SQL. Isso impede que comandos maliciosos, como `OR '1'='1'` ou `; DROP TABLE IF EXISTS users; --`, sejam executados, pois o SGBD não os interpreta como instruções, mas como strings literais.

A separação entre o código da consulta e os dados é feita utilizando marcadores de posição (placeholders) no SQL, como `$1`, `$2`, entre outros, dependendo da linguagem ou biblioteca utilizada. Esses marcadores são substituídos de forma segura pelas bibliotecas de acesso ao banco, que cuidam da escapagem de caracteres especiais e da tipagem correta dos dados. O código da rota `/login-seguro` faz uso de parâmetros para submeter o comando SQL para o SGBD:

```
const query = `SELECT * FROM users WHERE username = $1 AND password = $2`;
```

```
const result = await db.query(query, [username, password]);
```

Neste exemplo, os dados de entrada (`username` e `password`) são enviados separadamente do SQL. Com isso, mesmo que um usuário mal-intencionado envie comandos perigosos, eles não serão executados como parte da consulta.

### ORMs e Query Builders: Segurança por padrão

Muitas aplicações modernas utilizam ORMs (Object-Relational Mappers) ou Query Builders, que abstraem o uso direto de SQL. Esses frameworks já utilizam consultas parametrizadas por padrão, o que ajuda a reduzir significativamente o risco de SQL Injection.

Exemplos de ORMs:

- Sequelize (Node.js);
- TypeORM (Node.js);
- Prisma (Node.js);
- Hibernate (Java);
- Entity Framework (.NET).

Utilizar consultas parametrizadas é uma prática essencial para proteger aplicações web contra ataques de SQL Injection. Essa abordagem simples e eficiente deve ser adotada sempre, mesmo em sistemas de pequeno porte ou em fase de desenvolvimento. Além disso, ao utilizar frameworks modernos como ORMs, o desenvolvedor conta com essa proteção por padrão, reduzindo o risco de vulnerabilidades críticas.

## IV. Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

Introdução e exercícios 1 a 5: <https://youtu.be/K7mjwnOrYKM>

Exercício 6: [https://youtu.be/XTiSqt3g5\\_8](https://youtu.be/XTiSqt3g5_8)

Exercício 7: <https://youtu.be/vUilRxrG1hw>

Exercício 8: <https://youtu.be/tKTr1HervhU>

**Exercício 1** – Fazer os comandos CURL para testar os endpoints `/login-inseguro` e `/login-seguro` passando os seguintes parâmetros pelo corpo da requisição:

```
{
  "username": "admin' --",
  "password": "qualquer"
}
```

**Exercício 2** – Fazer os comandos CURL para testar os endpoints `/login-inseguro` e `/login-seguro` passando os seguintes parâmetros pelo corpo da requisição:

```
{
  "username": "' OR '1'='1' --",
  "password": "qualquer"
}
```

**Exercício 3** – Fazer os comandos CURL para testar os endpoints `/login-inseguro` e `/login-seguro` passando os seguintes parâmetros pelo corpo da requisição:

```
{
  "username": "admin'; DELETE FROM users; --",
  "password": "qualquer"
}
```

**Exercício 4** – Fazer os comandos CURL para testar os endpoints `/login-inseguro` e `/login-seguro` passando os seguintes parâmetros pelo corpo da requisição:

```
{
  "username": "admin'; INSERT INTO users(username,password) VALUES('teste',
  '123'); --",
  "password": "qualquer"
}
```

**Exercício 5** – Fazer os comandos CURL para testar os endpoints `/login-inseguro` e `/login-seguro` passando os seguintes parâmetros pelo corpo da requisição:

```
{
  "username": "admin'; DROP TABLE IF EXISTS users; --",
  "password": "qualquer"
}
```

**Exercício 6** – Alterar o código sugerido no exemplo para incluir uma camada de validação da entrada usando a biblioteca zod (<https://www.npmjs.com/package/zod>).

Requisitos:

- a) Organize o projeto na seguinte estrutura:

```
src/
├── controllers/
│   ├── auth.controller.ts
│   └── db.ts
├── routes/
│   └── auth.routes.ts
└── schemas/
```



```
|   └─ login.schema.ts
└─ index.ts
```

b) Codifique no arquivo login.schema.ts as seguintes validações:

- deve receber exatamente as propriedades `username` e `password`. Exemplo de resposta:

Objeto JSON enviado pelo body da requisição:

```
{
  "username": "abc",
  "role": "admin"
}
```

Resposta do servidor:

```
{
  "errors": [
    {
      "code": "invalid_type",
      "expected": "string",
      "received": "undefined",
      "path": [ "password" ],
      "message": "Required"
    },
    {
      "code": "unrecognized_keys",
      "keys": [ "role" ],
      "path": [],
      "message": "Unrecognized key(s) in object: 'role'"
    }
  ]
}
```

- username:
  - precisa ser string;
  - mínimo de 3 caracteres;
  - máximo de 20 caracteres;
  - deve conter apenas letras, números, ponto, hífen ou sublinhado (Dica: use regex).
- password:
  - precisa ser string;
  - mínimo de 6 caracteres;
  - máximo de 10 caracteres.

Dicas:

- Sugestão de código para o arquivo auth.controller.ts:

```
import { Request, Response } from "express";
import { loginSchema } from "../schemas/login.schema";
import db from "../db";

export async function loginController(req: Request, res: Response): Promise<void> {
  try {
    // Primeiro validamos a entrada com zod
    const validatedData = loginSchema.parse(req.body);

    const { username, password } = validatedData;
```

```
// Consulta parametrizada para evitar SQL Injection
const query = `SELECT * FROM users WHERE username = $1 AND password = $2`;

const result = await db.query(query, [username, password]);

if (result.rowCount !== 0) {
  res.status(200).json({ message: "Login realizado com sucesso!" });
} else {
  res.status(401).json({ message: "Usuário ou senha inválidos" });
}
} catch (error) {
  if (error instanceof Error && "errors" in error) {
    // Caso o erro seja de validação do Zod
    res.status(400).json({ errors: (error as any).errors });
  } else {
    // Caso o erro seja de execução da query ou outro
    res.status(500).json({ message: "Erro interno no servidor" });
  }
}
}
```

- Sugestão de código para o arquivo auth.routes.ts. Veja que está sendo criado o endpoint `/login`.

```
import express from 'express';
import { loginController } from '../controllers/auth.controller';

const router = express.Router();

router.post('/login', loginController);

export default router;
```

- Use o método `strict` do objeto `Zod` para impedir propriedades adicionais.

**Exercício 7** – Alterar o código do Exercício 6 para substituir o armazenamento e comparação de senhas em texto plano por uma abordagem segura, usando *hashes* e *salting*.

Requisitos:

- Altere a lógica do controller para considerar os seguintes pontos:
  - Durante o cadastro de um usuário, a senha deve ser *hashada* com `bcrypt` (<https://www.npmjs.com/package/bcrypt>);
  - Durante o login, o sistema deve comparar o hash armazenado no banco com a senha fornecida utilizando `bcrypt.compare()`.
- Implementar uma rota para inserir registro na tabela `users`. Adicione o endpoint `/register` no arquivo `auth.routes.ts`.

**Exercício 8** – Alterar o código do Exercício 7 para registrar tentativas suspeitas de login, como forma de identificar possíveis ataques de força bruta ou SQL Injection.

Requisitos:

- Toda tentativa de login malsucedida deve ser registrada em um arquivo de log, contendo:
  - Data e hora da tentativa;
  - IP de origem (extraído de req.ip);
  - username informado;
  - Motivo da falha: "Usuário inexistente", "Senha incorreta", "Erro de validação" e adicionar um identificador simples de tentativa potencialmente maliciosa no `username` contendo ' `OR` ou `--`.
- Codificar essas regras no endpoint `/login`.

A seguir tem-se um exemplo de conteúdo do arquivo `access.log` após os seguintes parâmetros passados pelo body da requisição `/login`:

<pre>{   "username": "maria",   "password": "123456" }</pre>	<pre>{   "username": "maria",   "password": "123456",   "role": "admin" }</pre>
<pre>{   "username": "maria",   "password": "123456x" }</pre>	<pre>{   "username": "' OR",   "password": "123456" }</pre>

Conteúdo do arquivo `access.log`:

```
[2025-04-25T22:28:36.083Z] IP: ::ffff:127.0.0.1 | Username: marias | Motivo: Usuário inexistente
[2025-04-25T22:28:41.781Z] IP: ::ffff:127.0.0.1 | Username: maria | Motivo: Senha incorreta
[2025-04-25T22:28:58.871Z] IP: ::ffff:127.0.0.1 | Username: maria | Motivo: Erro de validação
[2025-04-25T22:30:49.638Z] IP: ::ffff:127.0.0.1 | Username: ' OR | Motivo: Possível tentativa de
SQL Injection
```

Dicas:

- Crie o arquivo `logger.ts` na estrutura do projeto:

```
src/
├── controllers/
│   ├── auth.controller.ts
│   └── db.ts
├── routes/
│   └── auth.routes.ts
├── schemas/
│   └── login.schema.ts
└── utils/
```

```
|   |— access.log <- esse arquivo será criado pelo código do logger.ts
|   |— logger.ts
|   |— index.ts
```

- Sugestão de código para o arquivo `utils/logger.ts`:

```
import fs from "fs";
import path from "path";

const logFile = path.join(__dirname, "./access.log");

export function logSuspicious(data: Record<string, any>) {
  try {
    const logEntry = `[${new Date().toISOString()}] IP: ${
      data.ip
    } | Username: ${data.username} | Motivo: ${data.reason}\n`;

    fs.appendFileSync(logFile, logEntry);
  } catch (e: any) {
    console.log("erro:", e.message);
  }
}
```

- Exemplo de uso da função `logSuspicious` no endpoint `/login`:

```
if (result.rowCount === 0) {
  logSuspicious({
    ip: req.ip,
    username,
    reason: "Usuário inexistente",
  });
  res.status(401).json({ message: "Usuário inexistente" });
}
```