UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Detecção Automática de Ataques de *Cross-Site Scripting* em Páginas *Web*

Thiago de Souza Rocha

Manaus - Amazonas
2013

Thiago de Souza Rocha

Detecção Automática de Ataques de *Cross-Site Scripting* em Páginas *Web*

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Informática.
Área de concentração: Redes de Computadores.

Orientador: Prof. Dr. Eduardo James Pereira Souto

Aos meus pais José Francisco e Mirlene Rocha e ao meu irmão Arnaldo Rocha.

Agradecimentos

Inicialmente a DEUS, primeiramente por ser o responsável pela minha existência e por me dar forças durante toda a minha caminhada.

Ao meu orientador, Prof. Eduardo Souto, principalmente pela paciência, e por ter acreditado em mim em momentos em que nem eu mesmo acreditava.

Aos professores, amigos e todos os integrantes do ICOMP, que contribuíram para a conclusão desse trabalho.

Ao professor Gilbert Breves Martins, pelas discussões e contribuições durante o desenvolvimento deste trabalho.

Ao meu avô Arnaldo Felisberto Imbiriba da Rocha, por ter sido o maior incentivador do meu mestrado.

A minha avó Ely Albuquerque da Rocha, por sempre me passar forças mesmo morando em outro estado.

Aos meus avós Francisco Pereira de Souza e Raimunda Pereira de Souza, pelo amor e carinho.

A todos os meus outros familiares por sempre me incentivarem.

Aos meus amigos, simplesmente por serem meus amigos, muito obrigado!

Resumo

A evolução no desenvolvimento para aplicações *web* favoreceu o surgimento de páginas dinâmicas. Tal evolução foi possível através da criação de novas tecnologias como funções de script e recursos avançados em navegadores *web* que proporcionaram a inserção de novas funcionalidades e criação de serviços interativos, tais como *Internet banking*, redes sociais, *e-commerce*, *blogs* e fóruns.

A utilização desses recursos e funcionalidades tem melhorado gradativamente a interatividade e usabilidade das aplicações *web*. Por outro lado, o uso inadequado dessas funcionalidades acarretou no surgimento de diversos ataques, entre eles, o *Cross-Site Scripting* (XSS). Este ataque tem recebido muito destaque nos últimos anos, por estar no topo de listas e relatórios das maiores ameaças para aplicações *web* nos últimos anos.

Este trabalho demonstra a viabilidade de uso de uma metodologia capaz de detectar ataques XSS em aplicações *web* através da análise de informações contidas nas aplicações. Um protótipo da metodologia, denominado de ETSSDetector, foi desenvolvido e comparado com outras ferramentas similares. Os resultados obtidos mostram que, através da análise dos campos de entrada, é possível a geração de testes mais efetivos, diminuindo a quantidade de requisições realizadas na aplicação. Além disso, a habilidade de preenchimento dos campos da aplicação apenas com informações válidas garante a submissão dos formulários das páginas, aumentando a taxa de detecção de ataques XSS.

Palavras-chave: aplicações *web*, *cross-site scripting*, testes de vulnerabilidades.

Abstract

The evolution in web applications development favored the emergence of dynamic pages. This development was made possible through the creation of new technologies like script functions and web browser advanced features that provided the insertion of new features and creation of interactive services, such as Internet banking, social networks, e-commerce, blogs and forums. The use of these new resources and features has gradually improved the interactivity and usability of web applications. Moreover, the inappropriate use of these features resulted in the emergence of several attacks, including, Cross-Site Scripting (XSS) that is highlighted at the top of lists and reports of the greatest threats to web applications in recent years. This works demonstrates the feasibility of using a methodology that is capable to detect XSS attacks by analyzing the information contained in applications. A prototype of the methodology, called ETSSDetector, was developed and compared with similar tools. The results show that by analyzing the input fields, it is possible to generate more effective tests, decreasing the amount of requests made in the application. Furthermore, the ability to fill the fields with only valid information ensures the submission of forms on pages, increasing the detection rate of XSS attacks.

Keywords: web applications, cross-site scripting, vulnerability tests.

Sumário

INTRODUÇÃO **2**

1.1. Motivação **4**

1.2. Objetivos & Metodologia de Pesquisa **5**

1.3. Organização do Trabalho **6**

CONCEITOS BÁSICOS **8**

2.1.1. DOM **10**

2.1.2. A Linguagem *Javascript* **11**

2.1.3. O Protocolo HTTP **12**

2.2. Cross-Site Scripting **14**

2.2.1. Classificação **15**
2.2.1.1 XSS Não-Persistente 15
2.2.1.2 XSS Persistente 16
2.2.1.3 XSS DOM 18

2.3. Considerações Finais **20**

TRABALHOS RELACIONADOS **22**

3.1 Técnicas que modificam o lado servidor **22**
3.1.1 Context Sensitive String Evaluation (CSSE) 22
3.1.2 Cross Site Scripting Detection System (XSSDS) 23
3.1.3 XSS Guard 24
3.1.4 BLUEPRINT 25
3.1.5 Pixy 25

3.2 Técnicas que modificam o lado cliente **26**
3.2.1 Noxes 26
3.2.2 Automatic Detection/collection system for XSS 26
3.2.3 xJS 27

3.3 Scanners de vulnerabilidades web **27**
3.3.1 Acunetix 28
3.3.2 N-Stalker 29
3.3.3 Rational AppScan 29
3.3.4 WebCruiser 30
3.3.5 SecuBat 30
3.3.6 PowerFuzzer 31
3.3.7 WebSecurify 31
3.3.8 WebXSSDetector 32

3.4 Considerações Finais **32**

DETECÇÃO DE VULNERABILIDADES XSS EM APLICAÇÕES WEB **34**

4.1. Metodologia Proposta **34**

4.2. Fase de Extração e Preparação **36**
4.2.1. Extrator de Informações 36
4.2.2. Qualificador de Pontos de Vulnerabilidade 37

4.3. Fase de Testes **38**
4.3.1. Injeção de Código 38
4.3.2. Geração de Código Malicioso 40

4.4. Fase de Análise **41**
4.4.1. Detecção de Ataques 41

4.5. Base de Dados **42**

4.6. ETSSDetector **42**
4.6.1. Simulador 43
4.6.2. Módulo de Extração 44
4.6.3 Módulo de qualificação. 46
4.6.4 Módulo de Teste 47
4.6.5 Módulo de Análise 49
4.6.6 Módulo de Relatórios 49
4.6.7 Base de Dados 50

4.7 Considerações Finais **51**

RESULTADOS **53**

5.1. Protocolo Experimental **53**

5.2. Ambiente de Experimentação **54**

5.3. Cenário de Testes **54**
5.3.1. Primeiro Cenário 54
5.3.2. Segundo Cenário 57
5.3.3. Terceiro Cenário 59

5.4. Considerações Finais **60**

CONCLUSÕES & TRABALHOS FUTUROS **61**

6.1 Trabalhos Futuros **62**

REFERÊNCIAS **63**

Lista de Figuras

Figura 1.1: Distribuição de vulnerabilidades reportadas desde 2004.. 3
Figura 1.2: Vulnerabilidades no período 2010-2012. 4
Figura 2.1: Arquitetura de referência de um navegador... 9
Figura 2.2: Exemplo de uma estrutura DOM. 11
Figura 2.3:Exemplo de delimitação de código Javascript. 12
Figura 2.4: Interação entre o navegador e a aplicação *web*. 12
Figura 2.5:Exemplo de ataque no lado cliente. 14
Figura 2.6: Exemplo de ataque no lado servidor. 14
Figura 2.7:Procedimento de um Ataque XSS não persistente. 15
Figura 2.8:Exemplo real de ataque XSS não persistente. 16
Figura 2.9: Procedimento de um ataque XSS persistente. 16
Figura 2.10: Tela do *Adobe Flash Player* utilizada para distrair a vítima. 17
Figura 2.11: Spam sendo realizado no Facebook. 18
Figura 2.12: Exemplo de código de página vulnerável a um ataque do tipo DOM. 18
Figura 2.13: Exemplo de ataque do tipo DOM. 19
Figura 3.1: Arquitetura do CSSE. 22
Figura 3.2: Arquitetura do XSS Guard.. 24
Figura 3.3: Utilização do BLUEPRINT. 25
Figura 3.4: Tela inicial do Acunetix.. 28
Figura 3.5: Tela inicial do N-Stalker. 29
Figura 3.6: Tela inicial do AppScan. 29
Figura 3.7: Tela inicial do WebCruiser. 30
Figura 3.8: Arquitetura do SecuBat. 30
Figura 3.9: Tela inicial do PowerFuzzer. 31
Figura 3.10:Tela inicial do WebSecurify. 32
Figura 4.1: Metodologia. 35
Figura 4.2: O processo de extração de informações na página *web*. 36
Figura 4.3: Análise do elemento *input*. 38
Figura 4.4: Processo da fase de injeção. 39
Figura 4.5: Processo de detecção. 41
Figura 4.6: Arquitetura do ETSSDetector. 43
Figura 4.7: Tela de configuração inicial. 44
Figura 4.8: Exemplo de endereço com parâmetros. 45
Figura 4.9: Progresso do ETSSDetector. 49
Figura 4.10: Exemplo de relatório do ETSSDetector. 49
Figura 5.1: Tela inicial da aplicação de teste do Acunetix... 55
Figura 5.2: Ataque XSS na página “criar usuários”. 57
Figura 5.3: Campos da aplicação para teste do processo de qualificação. 57
Figura 5.4: Exemplo de ataque do PowerFuzzer. 58
Figura 5.5: Ataque XSS detectado pelo ETSSDetector. 60

Lista de Tabelas

Tabela 3.1: Modificações realizadas pelos trabalhos relacionados. 33
Tabela 4.1: Exemplos de variações de ataque com a tag javascript. 40
Tabela 4.2:Categorias de casos de teste. 48
Tabela 4.3: Tabela webpage. 50
Tabela 4.4: Tabela form. 51
Tabela 4.5: Tabela input. 51
Tabela 4.6: Tabela querystring. 51
Tabela 4.7: Tabela qualifier. 51
Tabela 5.1: Resultados do primeiro cenário de teste. 55
Tabela 5.2: Resultados do segundo cenário de teste. 58
Tabela 5.3: Resultados do terceiro cenário de teste. 59

Lista de Abreviaturas

AJAX – *Asynchronous Javascript and XML;*

CSS – *Cascading Style Sheets*;

CSRF – *Cross-Site Request Forgery*;

DNS – *Domain Name Service*;

DOM – *Document Object Model*;

FTP – *File Transfer Protocol*;

HTML – *HyperText Markup Language*;

HTTP – *HyperText Transfer Protocol*;

IPA – *Information Technology Promotion Agency*;

OWASP – *Open Web Application Security Project*;

SQL – *Structured Query Language;*

SSL – *Secured Socket Layer;*

XML – *eXtensible Markup Language*

XSS – *Cross-Site Scripting*;

Capítulo 1

Introdução

As aplicações *web* têm sido utilizadas como um dos principais canais de comunicação entre os provedores de serviço e seus usuários. Portanto, garantir a segurança dessas aplicações se tornou uma tarefa prioritária e indispensável. A estrutura de sites dinâmicos, constituída por um conjunto de objetos, tais como *tags* de HTML (*HyperText Markup Language*), funções de script, hiperlinks e recursos avançados em navegadores *web*, proporcionaram a inserção de novas funcionalidades e criação de serviços interativos, tais como *e-commerce*, *Internet banking*, redes sociais, *blogs* e fóruns.

A adição desses novos recursos e funcionalidades tem melhorado, cada vez mais, a interatividade e usabilidade das aplicações *web*. Por outro lado, o uso inadequado dessas funcionalidades tem proporcionado o surgimento de diversos ataques. Conforme Shelly [1], o número de vulnerabilidades encontradas atualmente nas aplicações *web* são maiores que as encontradas nos sistemas operacionais. Tal afirmação, explica a razão das aplicações *web* serem os alvos preferenciais dos atacantes.

A OWASP (*Open Web Application Security Project*) [2], uma entidade sem fins lucrativos e de reconhecimento internacional que contribui para a melhoria da segurança de softwares e aplicativos, disponibiliza anualmente um relatório que lista as vulnerabilidades mais encontradas em aplicações *web*. O relatório de 2013, por exemplo, destaca os seguintes ataques [3]:

- Ataques de injeção como SQL *Injection* e LDAP;
- Quebra de autenticação e controle de sessão;
- *Cross-Site Scripting* (XSS);
- Referência insegura direta a um objeto;
- Falta de configuração de segurança;
- Exposição de dados importantes;

- Falta de funções de controle de acesso;
- *Cross-Site Request Forgery* (CSRF);
- Uso de componentes vulneráveis;
- Redirecionamentos e encaminhamentos inválidos.

Embora a lista inclua apenas as 10 vulnerabilidades mais críticas, existem outras vulnerabilidades, como estouro de *buffer*, *phishing* e execução de arquivos maliciosos, que também são comumente empregas pelos atacantes. Como o número de vulnerabilidades nas aplicações *web* é extremamente grande, apenas as vulnerabilidades mais relevantes são tratadas nos relatórios da OWASP.

Um dos pontos centrais que possibilita a existência de vulnerabilidades é a ausência ou falha na validação e interpretação dos campos de entrada das aplicações. A maior parte das linguagens de programação *web* não fornece, por padrão, uma passagem de dados segura para o cliente, conforme destacam Wasserman e Su [4]. A falta desse procedimento pode viabilizar um dos ataques mais frequentes em aplicações *web*, o *Cross-Site Scripting* (XSS) [5]. Desde 2007, este ataque é relacionado como um dos três principais ataques nos relatórios anuais da OWASP. O relatório de 2012 da agência IPA (*Information Technology Promotion Agency*) [6] mostra várias falhas de vulnerabilidades reportadas em aplicações *web* desde 2004. Entre essas falhas, 52% são ataques do tipo XSS como mostrado na Figura 1.1.

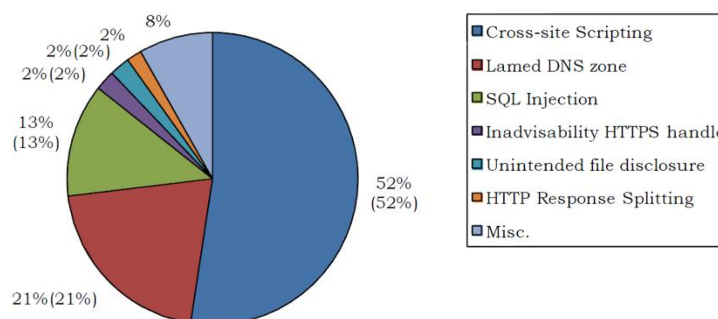

Figura 1.1: Distribuição de vulnerabilidades reportadas desde 2004 [6].

Esse mesmo relatório destaca ainda a quantidade de vulnerabilidades reportadas no período de outubro de 2010 a setembro de 2011. O gráfico da Figura 1.2 mostra que 86% das vulnerabilidades associadas a aplicações *web* são relacionadas a ataques XSS, SQL *injection* e Falhas de DNS (*Domain Name Service*).

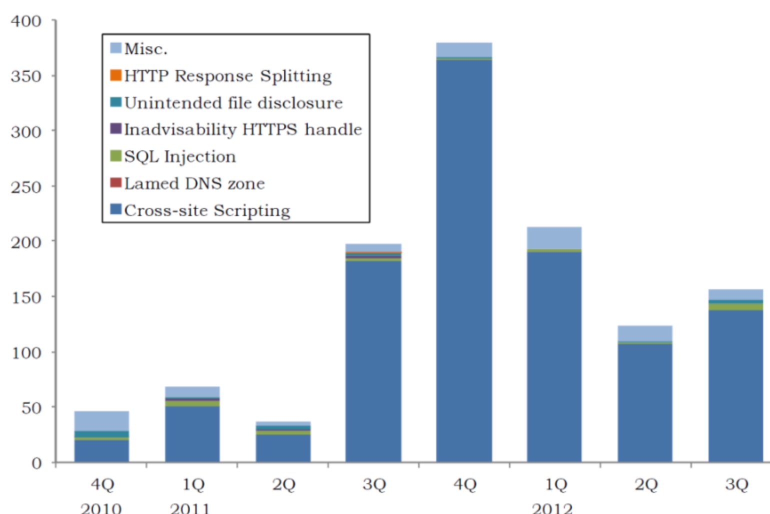

Figura 1.2: Vulnerabilidades no período 2010-2012 [6].

Outras pesquisas também indicam que ataques XSS se mantêm no topo das listas das maiores vulnerabilidades em aplicações *web* nos últimos anos, conforme as estatísticas divulgadas em Christey et al. [7], *Web Application Security Statistics Project* 2007 [8] e *Website Security Statistic Report* [9].

Ataques XSS são fáceis de executar, porém difíceis de detectar. Uma das causas é a grande flexibilidade do HTML, que oferece muitas possibilidades aos atacantes. Mesmo usuários de grandes aplicações web como Twitter e Facebook têm sido vítimas de ataques XSS [10], [11], embora os desenvolvedores dessas aplicações garantam que os serviços *web* oferecidos seguem um rígido padrão de segurança.

Para mitigar o problema de ataques XSS, diferentes abordagens e técnicas têm sido propostas. Entre essas abordagens, os *scanners* de vulnerabilidades *web,* que realizam testes caixa preta, têm sido os mais empregados por profissionais de segurança para realizar os testes de vulnerabilidade em aplicações *web*. Tal escolha é devido a diversos fatores como: *i)* não é necessário ter acesso ao código fonte da aplicação para realizar os testes de vulnerabilidade (testes de caixa preta); *ii)* são fáceis de usar e, portanto, realizam os testes rapidamente; *iii)* são capazes de identificar uma grande variedade de vulnerabilidades [1].

1.1. Motivação

A detecção de ataques XSS em aplicações *web* pode ser feita de forma manual ou automática. Na forma manual, o analista de segurança realiza testes manualmente em cada página da aplicação na busca por campos que possibilitem a injeção de códigos maliciosos, o

que torna esse processo inviável principalmente considerando a quantidade e a variabilidade dos ataques. Assim, a solução mais adequada é fazer uso de ferramentas automatizadas para realizar os testes de vulnerabilidade. Entretanto, pesquisas recentes mostram que muitas das ferramentas de testes automatizadas (*scanners* de vulnerabilidades) têm problemas com a detecção de ataques XSS nas aplicações web. O trabalho de Bau et al. [12], por exemplo, que apresenta uma análise sobre um conjunto de *scanners*, concluiu que a detecção dos ataques XSS persistentes através dessas ferramentas é de apenas 15%. Conclusões similares sobre a efetividade dessas ferramentas também foram obtidas em [13]. Kosuga [14] observa que a maioria dos *scanners* aplica todos os testes possíveis em todos os campos de entrada da aplicação, gerando um esforço computacional desnecessário. Por último, Mcallister et al. [15] observa que a maioria dos programas falha ao testar partes das aplicações quando se envolve o preenchimento de formulários complexos. Essa falha pode fazer com que a ferramenta automatizada não consiga coletar todas as páginas da aplicação que está sendo avaliada. Tal fato influencia na cobertura dos testes na aplicação, pois as páginas que não foram coletadas não serão testadas, podendo gerar falsos negativos.

Na tentativa de minimizar estes problemas, este trabalho propõe uma metodologia para detectar ataques XSS de forma automática através da análise de informações contidas nas aplicações *web*.

1.2. Objetivos & Metodologia de Pesquisa

Este trabalho tem como objetivo principal propor e avaliar uma metodologia para detectar ataques de *Cross-Site Scripting* através da análise de informações contidas em aplicações web.

Para atingir esse objetivo será feita a identificação, análise e caracterização dos campos que compõem a página web analisada. Com base nessa análise será possível aplicar técnicas para qualificar e preencher os campos de entrada com valores válidos possibilitando a submissão correta dos mesmos. A qualificação dos campos de entrada permitirá a obtenção de todas as páginas da aplicação, possibilitando uma análise mais ampla e o aumento da taxa de detecção de ataques XSS.

A caracterização das informações pela metodologia permitirá a geração de casos de testes mais adequados em cada campo de entrada, possibilitando a redução no tempo computacional gasto pelas ferramentas de testes de vulnerabilidades.

Para validar a metodologia proposta um protótipo, chamado ETSSDetector, baseado nas etapas definidas na metodologia será construído e comparado com outras ferramentas de testes de vulnerabilidades.

Como resultado, a utilização da ferramenta ETSSDetector beneficiará desenvolvedores e administradores de rede que, através dos testes automatizados, poderão obter informações detalhadas sobre os pontos de vulnerabilidades existentes em suas aplicações.

1.3. Organização do Trabalho

O restante deste trabalho está estruturado da seguinte maneira:

- O Capítulo 2 fornece os principais conceitos relacionados ao entendimento deste trabalho. Inicialmente serão detalhados os conceitos relacionados a arquitetura básica da web mostrando uma arquitetura de referência de um navegador e as definições de tecnologias que são utilizadas no mesmo como HTML (*HyperText Markup Language*), CSS (*Cascading Style Sheets*) e DOM (*Document Object Model*). Posteriormente, para facilitar o entendimento de um ataque *Cross-Site Scripting,* são apresentados sua definição, categorização, vulnerabilidades associadas e formas de prevenção.
- O Capítulo 3 apresenta uma visão geral sobre alguns trabalhos relacionados ao tema dessa dissertação. Para melhorar a organização, os trabalhos foram divididos em três categorias: *i)* trabalhos que modificam o lado cliente, onde são tratadas as ferramentas que modificam o navegador do usuário; *ii)* trabalhos que modificam o lado servidor, onde são tratados os trabalhos que modificam o servidor onde a aplicação vulnerável esta hospedada; e *iii) scanners* de vulnerabilidades *web,* ferramentas que não realizam nenhuma modificação no lado cliente e lado servidor.
- O Capítulo 4 mostra a metodologia proposta para atingir os objetivos propostos neste trabalho. A metodologia proposta é dividida nas fases de: extração e

preparação, testes e análise dos resultados. Por fim, o scanner ETSSDetector, uma implementação da metodologia proposta, é apresentado.

- O Capitulo 5 aborda os experimentos realizados. Foram criados diversos cenários de testes abordando os problemas expostos neste capítulo. Através desses cenários de testes, comparações do ETSSDetector com outras ferramentas de vulnerabilidades *web* são realizadas. Por fim, os resultados são discutidos e apresentados.
- O Capítulo 6 finaliza este trabalho apresentando as considerações finais e trabalhos futuros.

Capítulo 2

Conceitos Básicos

Este Capítulo introduz os conceitos básicos necessários para o entendimento deste trabalho. O capítulo inicia com uma breve explanação sobre os componentes e tecnologias que compõem uma arquitetura web e que estão relacionados com o escopo do trabalho. Em seguida, os principais conceitos relacionados à ataques *Cross-Site Scripting* são apresentados.

2.1. Arquitetura Web

Para entender os ataques de XSS deve-se conhecer o ambiente que o mesmo atua e seus principais vetores de propagação como *links*, campos e parâmetros das páginas. Neste trabalho, tais vetores serão nomeados como pontos de vulnerabilidade. A seguir serão definidos os conceitos relacionados a arquitetura *web* cliente-servidor.

A *web* pode ser definida como uma arquitetura baseada no modelo cliente-servidor, composta por *softwares*, protocolos e convenções que têm por objetivo disponibilizar serviços, informações e documentos multimídia [16].

Os documentos tipicamente são escritos na web através da linguagem HTML, uma linguagem de marcação utilizada para exibir as páginas enviadas por um servidor para o navegador do usuário. Utilizando essa linguagem é possível criar páginas *web*, adicionar *links* para outros documentos ou para outros domínios na sua própria página [17]. Embora seja possível criar páginas apenas com a utilização da linguagem HTML, outras tecnologias podem ser utilizadas para melhorar a aparência visual e a interatividade dos usuários. Entre as quais, destacam-se:

1. **CSS**: Tecnologia utilizada para permitir que os desenvolvedores adicionem informações de apresentação e estilo nas páginas *web*.

2. **Código *Scripts***: Códigos *scripts* são utilizados realizar ações no navegador do usuário e proporcionar conteúdo dinâmico às páginas web. Um *script* (trecho de código) é executado por um interpretador (por exemplo, um interpretador *JavaScript*) que interpreta e executa o código.
3. **DOM**: Interface de programação de aplicativos (API) para documentos HTML e XML (*eXtensible Markup Language*) que representa a estrutura lógica dos documentos e o meio pelo qual um documento é acessado e manipulado. Maiores detalhes sobre esse recurso são apresentados na seção 2.1.1.

Tais recursos são usados pelos desenvolvedores de aplicações para tornar as páginas *web* mais dinâmicas. Tais páginas permitem que o seu conteúdo seja modificado no próprio navegador do cliente através de chamadas assíncronas que executam *scripts*.

Grosskurth e Godfrey [17] realizaram um estudo e através da comparação de dois navegadores definiram uma arquitetura básica de referência. A Figura 2.1 mostra os principais componentes que compõem essa arquitetura.

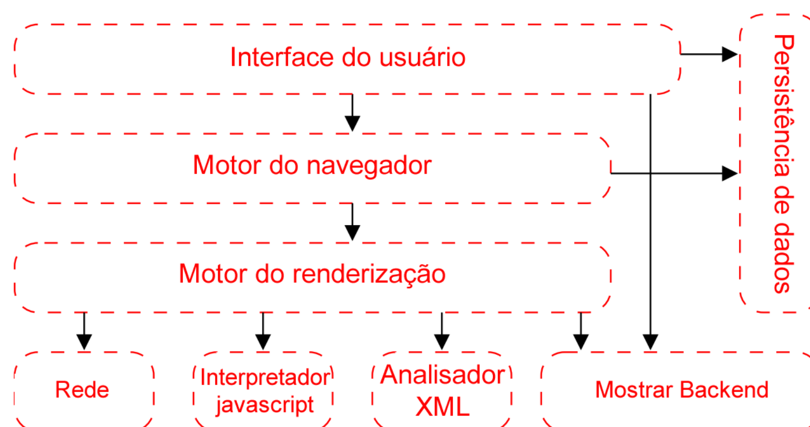

Figura 2.1: Arquitetura de referência de um navegador [22].

1. **Interface do usuário**: Camada que fica localizada entre o usuário e o motor do navegador. Em geral, a interface do usuário fornece algumas funcionalidades como barra de progresso visual para carregamento de páginas e downloads inteligentes.
2. **Motor do navegador**: Este componente fornece uma interface com o motor de renderização. Responsável por carregar os documentos e suportar algumas ações do navegador como voltar, passar e recarregar as páginas.

3. **Motor de renderização**: Este componente é responsável por produzir a representação visual dos documentos. Em geral, tais motores são capazes de renderizar páginas com conteúdo HTML e XML.
4. **Rede**: A camada de rede é responsável por implementar os protocolos de transferência de dados como HTTP (*HyperText Transfer Protocol)* e FTP (*File Transfer Protocol*), e por interpretar os diferentes tipos de *charsets* (biblioteca de *caracteres*) e *mime types* (tipo de documentos) que existem.
5. **Interpretador *Javascript***: Utilizados para executar os códigos escritos na linguagem *Javascript*. Alguns recursos *Javascript* como abrir janelas *popup*, podem ser desabilitados pelo motor do navegador ou pelo motor de renderização por aspectos de segurança.
6. **Analisador XML:** Analisadores XML são responsáveis por transforma os documentos XML em uma árvore do tipo DOM.
7. **Mostrar Backend:** Este componente interage com o sistema operacional para prover funcionalidades para a criação de desenhos e janelas.
8. **Persistência de dados:** Esta camada responsável por salvar no disco os dados associados com a sessão de navegação do usuário. Esses dados podem ser desde configurações de barras do navegador a *cookies* e certificados de segurança.

2.1.1. DOM

A W3C [18] define DOM como uma API para documentos HTML e XML que representa a estrutura lógica dos documentos e o meio pelo qual um documento é acessado e manipulado. Através da *interface* DOM, programas e *scripts* acessam e atualizam dinamicamente o conteúdo, a estrutura e estilos de documentos.

A *interface* DOM possibilita que uma página seja manipulada, permitindo o acesso direto a qualquer elemento da mesma e apresentando a página HTML como uma árvore organizada hierarquicamente, onde cada elemento é um nó [19]. Por exemplo, a Figura 2.2 mostra a representação de uma estrutura DOM a partir de um pequeno trecho de código HTML. O elemento que representa a tag HTML *head* está hierarquicamente acima do elemento que representa a *tag* HTML *title*.

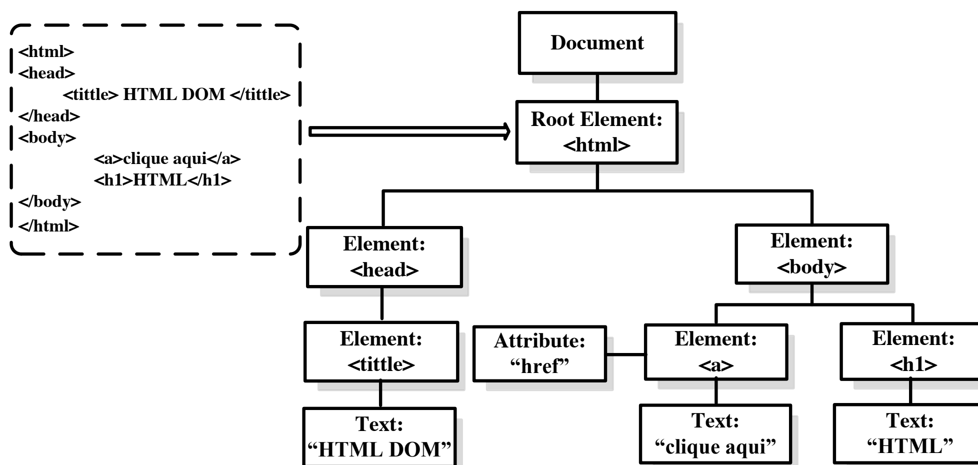

Figura 2.2: Exemplo de uma estrutura DOM.

Com a utilização da interface DOM é possível criar documentos, navegar pela estrutura de um documento qualquer e incluir, alterar ou apagar nós do documento de forma dinâmica. A utilização indevida desses recursos pode acarretar na execução de *scripts* maliciosos e, consequentemente, no roubo de informações dos usuários.

2.1.2. A Linguagem *Javascript*

A utilização de linguagens *scripts* tem estimulado e proporcionado o desenvolvimento de aplicações *web* mais dinâmicas e interativas. Conforme pesquisa da RedMonk [20], a linguagem *JavaScript* é a linguagem mais utilizada atualmente no desenvolvimento de páginas *web*.

Inicialmente conhecida como *LiveScript*, a linguagem *Javascript* foi desenvolvido pela empresa Netscape para tornar o navegador mais poderoso e funcional, proporcionando aos usuários uma experiência mais rica na web [19]. Atualmente, todos os navegadores suportam *Javascript*, apesar de haver diferenças de sintaxe entre um navegador e outro, especialmente entre o navegador *Internet Explorer* e os demais. Yue e Wang [21] destacam que 96,9% das páginas *web* mais populares do mundo fazem uso de *scripts* com base na linguagem *JavaScript*.

Um código *Javascript* permite a execução de funções no navegador do cliente que antes, obrigatoriamente deviam ser processadas pelo servidor, como validação de campos, abertura de novas janelas, criação e manipulação de camadas, cálculos, entre outras.

O potencial dos *scripts* aliado a popularidade da linguagem *Javascript*, tornou esse

recurso o principal vetor de disseminação de código malicioso na *web*. Basicamente, existem duas formas de se executar *scripts*: diretamente ou como resposta a eventos. Na execução direta, na maioria das vezes, as *tags* <script> e </script> são usadas para delimitar a inserção de código *scripts* em páginas HTML, como mostrada na Figura 2.3.

```
<script type="text/javascript">...</script>
```

Figura 2.3:Exemplo de delimitação de código Javascript.

Entretanto, isso é apenas uma parte do problema, a maioria dos navegadores *web* permite acesso via *script* a estrutura DOM da página *web*. Tal fato possibilita que o *script* mude qualquer ou todos aspectos da página. A função `eval(s)` do *Javascript*, por exemplo, cujo argumento é uma cadeia de caracteres (*string*), permite que esta seja avaliada e executada com privilégio do chamador. Assim, um código malicioso pode ser passado com argumento para a função `eval(s)` e ser executado para ativar um ataque.

Na execução como resposta a eventos, o código é inserido dentro de atributos de determinadas *tags* HTML. Tais atributos representam as ações praticadas pelo usuário e são chamados de manipuladores de eventos. Eventos são disparados por tarefas como clicar de um botão do mouse, movimentar o mouse, selecionar um texto, entre outras. Apesar dos navegadores proporcionarem a execução de conteúdo dinâmico, é necessário um meio de comunicação com o servidor para baixar as páginas e interagir com as aplicações *web*. Esse meio de comunicação ocorre através do protocolo HTTP com a geração de requisições e respostas, como mostra a Figura 2.4.

Figura 2.4: Interação entre o navegador e a aplicação *web*.

2.1.3. O Protocolo HTTP

A W3C define o HTTP como um protocolo de nível de aplicação para sistemas de hipermídia, colaborativos e distribuídos [22]. É um protocolo genérico, sem estado e orientado a objetos que pode ser usado para diversas tarefas, tais como servidores de nomes e sistemas de gerenciamento de objetos distribuídos, através da extensão de seus métodos de

requisição.

O HTTP é definido como um protocolo sem estado, pois não guarda informações de estado entre as requisições ao servidor. Caso um servidor receba várias requisições ao mesmo tempo, o mesmo não saberá separar as requisições por cliente de forma a considerar informações de uma requisição que influenciem na outra. Portanto é impossível desenvolver uma aplicação, por exemplo, de comércio eletrônico utilizando apenas o protocolo HTTP, outros recursos são utilizados para manter as informações como sessões e *cookies*.

Os *cookies* proporcionam que o servidor seja capaz de trocar informações de estado com o navegador do usuário. Tecnicamente é uma pequena quantidade de informação armazenada temporariamente pelo navegador. Os *cookies* podem ser de dois tipos, os persistentes que ficam armazenados no HD (*Hard Drive*) do usuário e os de sessão que são deletados quando o usuário fecha o navegador [23].

Os *cookies* deveriam ser utilizados apenas para guardar informações irrelevantes e temporárias. Contudo muitos desenvolvedores de aplicação os utilizam também para manter informações pessoais como senhas de usuários e *logins*. Por este motivo, os *cookies* tornam-se alvos constantes de ataques em aplicações web.

As sessões têm um principio similar aos *cookies*, porém o armazenamento do estado é feito no servidor *web* e não no navegador, proporcionando uma segurança maior [23]. A maioria das aplicações utiliza as sessões para manter as informações entre as requisições durante certo período de tempo. Uma sessão é sempre identificada com um identificador único e por ser outra forma de guardar informações também é alvo de constantes ataques.

Como mostrado na Figura 2.4, cada conexão que o navegador cria possui uma requisição e uma resposta. Os métodos GET e POST do protocolo HTTP são utilizados para gerar as requisições [24].

No método GET, as informações preenchidas pelos usuários são adicionadas nos endereços das páginas e são conhecidas como parâmetros de *querystring*. No método POST, as informações preenchidas não aparecem no endereço da página. Uma mensagem é criada e os dados preenchidos são enviados no corpo dessa mensagem.

Claramente pode-se perceber que o método GET é mais perigoso que o método POST, pois as informações são passadas através do endereço da página. Informações importantes nunca deveriam ser passadas através do método GET.

Essa interação entre o usuário e a aplicação através dos métodos GET ou POST do protocolo HTTP proporciona a execução de ataques como *phishing, spam e* XSS.

2.2. Cross-Site Scripting

Um ataque *Cross-Site scripting* (XSS) ocorre quando o atacante explora falhas de validação em campos de entrada de páginas *web*, permitindo a injeção de código malicioso. Esse código pode ser interpretado e executado a partir dos navegadores dos usuários. Esse tipo de ataque pode levar ao roubo de vários recursos da aplicação como *cookies* e sessões de usuários, dados pessoais, números de cartões de crédito, senhas, *logins* e outras informações sensíveis ou críticas.

Grossman [5] define XSS como um vetor de ataque causado por *scripts* maliciosos no lado cliente ou servidor, onde os dados de entrada do usuário não são adequadamente validados, podendo permitir o roubo de informações confidenciais, sequestro de sessões do usuário, comprometimento do navegador cliente e da integridade do sistema sob execução.

Os ataques XSS podem ser executados no lado cliente (navegador de um usuário) ou no lado servidor (servidor da aplicação). Um exemplo de ataque no lado cliente é mostrado na Figura 2.5.

```
http://localhost/XSS/get.php?name=>'"><script>alert('XSS')</script>
```

Figura 2.5:Exemplo de ataque no lado cliente.

Neste tipo de ataque um código *Javascript* é enviado através do método GET e executado no navegador do usuário. Neste caso, um *script* exibindo um *alert* com o texto "XSS" é exibido no navegador do usuário. O fato de um atacante conseguir exibir um simples *alert* na tela do usuário significa dizer que qualquer ataque XSS também pode ser executado. Nos ataques XSS executados no lado servidor o atacante tenta fazer o usuário executar algum código malicioso que o atacante possua hospedado em outra página como mostra a Figura 2.6.

```
http://localhost/XSS/get.php?url=http://www.hacker.com/cookie.php
```

Figura 2.6: Exemplo de ataque no lado servidor.

No exemplo da Figura 2.6, o atacante passa por parâmetro outra página que contenha um código malicioso inserido. Quando o usuário clica no *link* o código malicioso do atacante é adicionado na requisição e executado no servidor fazendo com que o atacante obtenha informações confidenciais do usuário. Existem diferentes tipos de ataques XSS como detalhados a seguir.

2.2.1. Classificação

Os ataques *Cross-Site Scripting* podem ser categorizado em [25]: não persistente, persistente e do tipo DOM.

2.2.1.1 XSS Não-Persistente

Os ataques XSS do tipo não persistente são os mais comuns [26]. Nesse tipo de ataque o código não é salvo na aplicação. O atacante analisa a aplicação procurando por campos de entrada que possam ser manipulados. Por exemplo, caixas de texto em aplicações de busca são um dos vetores mais utilizados nesse tipo de ataque. Após conseguir manipular a página *web*, o atacante envia a URL maliciosa para o usuário através de e-mails, mensagens instantâneas, ou *links* embutidos em outras páginas [27]. Figura 2.7 mostra um exemplo do fluxo de um ataque XSS não persistente.

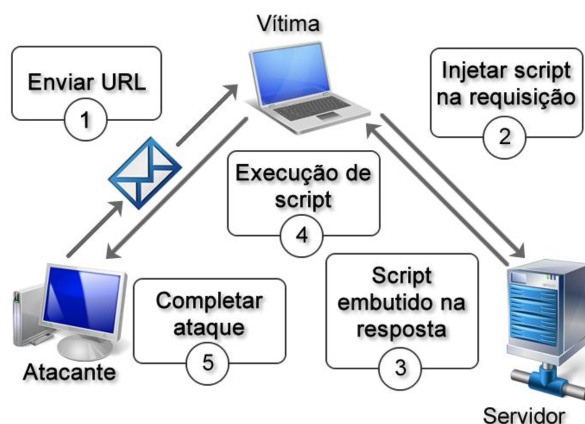

Figura 2.7:Procedimento de um Ataque XSS não persistente.

No passo 1, o atacante envia um e-mail para vítima contendo um *link* para uma aplicação *web* vulnerável. Quando o usuário clica no *link* enviado por e-mail (passo 2), uma requisição HTTP é criada. Com isso o servidor *web* recebe essa requisição que gera uma resposta com o código malicioso embutido (passo 3). Em seguida (passo 4), o código é executado enviando as informações confidenciais que o atacante requisitou finalizando o

ataque no passo 5.

Muitos ataques XSS do tipo não persistente têm sido reportados na literatura, como uma vulnerabilidade que ocorre no provedor de páginas 000WebHost, um dos provedores de hospedagem gratuita mais famosos que existe. A vulnerabilidade foi descoberta no dia 30 de Janeiro de 2013 [28]. A aplicação utilizada pelo usuário para requerer o serviço de hospedagem, especificamente os campos *domain, subdomain, name e email* estão vulneráveis a ataques XSS. Um exemplo do ataque efetuado é mostrado na Figura 2.8.

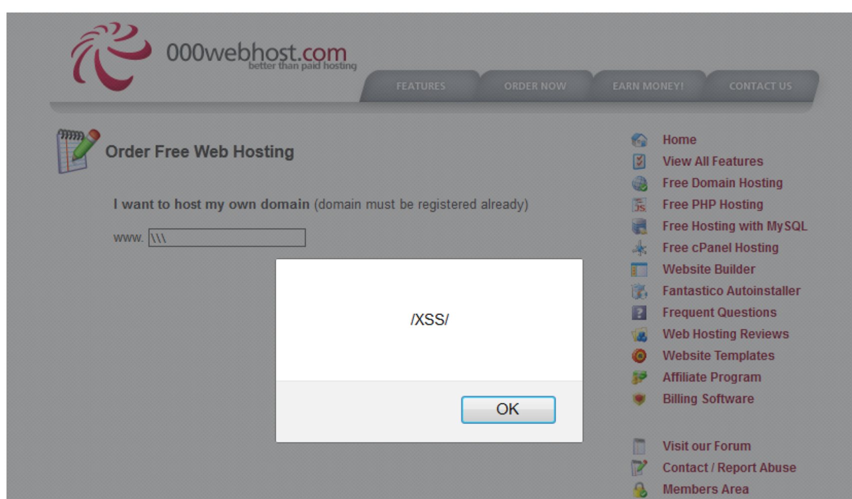

Figura 2.8:Exemplo real de ataque XSS não persistente.

2.2.1.2 XSS Persistente

Figura 2.9: Procedimento de um ataque XSS persistente.

Ataques XSS persistentes ocorrem, na maioria das vezes, em fóruns ou aplicações de *Webmail* e não precisam de páginas com código embutido para serem executados [5]. O ataque só é completado quando o código salvo na aplicação está sendo acessado e utilizado [29].

O atacante acessa a aplicação *web* vulnerável e salva um código malicioso no repositório da aplicação. Quando um usuário acessa a parte vulnerável da aplicação o *script* malicioso é carregado e executado obtendo informações confidenciais, a Figura 2.9 mostra um exemplo do fluxo de um ataque persistente.

Como mostrado na Figura 2.9, o atacante corrompe a aplicação vulnerável, localizada em um servidor, adicionando um *script*. Assim, o código malicioso será executado por todo usuário que acessar a página que o contenha. Comentários de blogs, mensagens de fóruns e mensagens em chats são os vetores mais usados nesse tipo de ataque. Por exemplo, ao clicar em um *link* disponibilizado no fórum, o navegador do usuário irá automaticamente executar o código malicioso. Tal fato faz com que o ataque XSS persistente seja mais perigoso que os outros tipos de ataque XSS, pois o usuário não tem como se defender de um ataque que ele não sabe que foi executado [5].

Um exemplo real de ataque XSS persistente foi reportado recentemente pela Symantec [30] e ocorreu no Facebook. Nesse ataque quando o usuário clica no *link* malicioso dentro do Facebook, um *script* é executado mostrando uma tela de atualização do *Adobe Flash Player* para distrair a vítima, como mostra a Figura 2.10.

Figura 2.10: Tela do *Adobe Flash Player* utilizada para distrair a vítima [34].

Enquanto a tela da Figura 2.10 é mostrada o *script* realiza requisições AJAX (*Asynchronous Javascript and XML*) para obter informações do perfil do Facebook da vítima. De posse dessas informações, uma postagem é realizada com *links* que realizam *spams* de produtos de perda de peso e de promoções para entrega de *ipads* grátis como mostra a Figura 2.11. Além de postar essas imagens no perfil da vítima, o mesmo envia mensagens com *links* maliciosos para os amigos objetivando a propagação do ataque.

Figura 2.11: Spam sendo realizado no Facebook [34].

2.2.1.3 XSS DOM

Enquanto os tipos de ataques XSS anteriores ocorrem no lado servidor, um ataque XSS do tipo DOM proporciona a execução de *scripts* no navegador da vitima [31]. Um ataque do tipo DOM requer que a página vulnerável permita que o usuário informe os dados que serão utilizados em *scripts* que utilizam o DOM e consigam modificar algum aspecto na página de forma insegura como *document.location*, *document.URL* ou *document.referrer* [32]. O objeto *document* do DOM representa a maioria das informações de uma página que é utilizada no navegador. Para facilitar o entendimento de um ataque do tipo DOM, considere o trecho de código da Figura 2.12.

```
 1 <HTML>
 2 <TITLE>Welcome!</TITLE>
 3 Hi
 4 <SCRIPT type="text/javascript">
 5 var pos=document.URL.indexOf("name=")+5;
 6 document.write(unescape(document.URL.substring(pos,document.URL.length)));
 7 </SCRIPT>
 8 <BR>
 9 Welcome to our system
10 …
11 </HTML>
```

Figura 2.12: Exemplo de código de página vulnerável a um ataque do tipo DOM.

Figura 2.13: Exemplo de ataque do tipo DOM.

Um exemplo real de ataque DOM *based* ocorreu no Yahoo. A vulnerabilidade foi encontrada por Shahin Ramezany em 2013 [33] que publicou um vídeo no site de compartilhamentos Youtube [34] mostrando como executar o ataque. Nesse vídeo, o atacante envia um e-mail para vítima com um *link* que possui o código malicioso. O vídeo mostra que a vítima ao acessar o e-mail e clicar no link envia os seus *cookies* para o atacante. De posse dos *cookies* da vítima, o atacante altera os valores dos *cookies* no seu navegador e atualiza a página passando a estar conectado com as credenciais da vítima.

2.2.2. Vulnerabilidades associadas a ataques XSS

Ataques XSS apresentam muitas variações que podem ser utilizadas nas aplicações Web com diferentes objetivos, somente para citar alguns [35] [36]:

1. O usuário pode, sem o seu consentimento, executar códigos maliciosos quando acessar páginas dinâmicas com dados fornecidos por um atacante.
2. O atacante pode finalizar uma sessão do usuário antes que o *cookie* de sessão expire.
3. O atacante pode fazer com que os usuários se conectem a um servidor malicioso da sua escolha.
4. O atacante pode obter privilégios de um usuário de uma aplicação Web.
5. As conexões SSL (*Secured Socket Layer*) podem ser expostas.
6. O atacante pode empregar scripts para roubar *cookies*.
7. Ataques de negação de serviço podem ser executados a partir de *scripts* maliciosos inseridos em páginas *web*
8. O atacante pode alterar o comportamento de formulários das páginas HTML.

Entre essas variações, as mais comuns são [24]:

- Roubo de *cookies* e de contas: ataques XSS são bastante empregados para a obtenção de informações armazenadas nos *cookies*. Essas informações podem conter *login* e senha de usuários que podem ser utilizadas pelos atacantes para se passar pelo usuário na aplicação.
- Roubo de outras informações: além do roubo de *cookies*, os atacantes comumente utilizam códigos maliciosos injetados em páginas *web* para espionar o comportamento do usuário, obtendo estatísticas, como o histórico de páginas visitadas. De posse dessa informação, o atacante emprega um ataque *phishing* fazendo com que o usuário envie informações para o atacante inconscientemente [37].
- Negação de serviço: Ataques XSS são também empregados em ataques de negação de serviço, tornando uma aplicação indisponível ou o navegador do usuário inoperante. Uma forma de realizar esse ataque é através de redirecionamentos infinitos para outros sites.
- Exploração de navegador: Nessa variação o navegador do usuário pode ser direcionado para uma página desenvolvida pelo atacante. Então, o atacante pode obter o controle do computador do usuário através de falhas de segurança que permitem que o atacante execute comandos arbitrários, possibilitando a instalação de cavalos de Tróia [38] entre outras ameaças.

2.3. Considerações Finais

Como mostrado neste Capítulo, existem várias formas de se explorar um ataque XSS e a prevenção desses ataques é um problema complexo que está longe de ser resolvido [5]. Diferente da maioria dos problemas de segurança não existe nenhuma solução rápida para os ataques XSS que seja aceita pela maioria das suas variações. O primeiro impedimento é que um navegador foi criado apenas para fazer requisições e processar resultados, e não tem a capacidade de decidir se um código está realizando alguma atividade maliciosa. O segundo problema é que os desenvolvedores não estão projetando as aplicações de forma segura. Isso faz com que os atacantes possam encontrar uma forma de adicionar códigos maliciosos no navegador do usuário.

Shalini e Usha [39] descrevem que existe uma grande variedade de técnicas para tentar combater XSS, incluindo os seguintes aspectos: análise estática, análise dinâmica, testes caixa-preta, testes caixa-branca e detecção de anomalias. Tadeuz [29] enfatiza que a maioria dessas técnicas depende do trabalho do desenvolvedor, fazendo com que elas sejam passíveis de erros. Alguns trabalhos de pesquisa que utilizam essas técnicas serão estudados no próximo capítulo.

Capítulo 3

Trabalhos Relacionados

Este capítulo mostra pesquisas recentes relacionadas aos ataques XSS que aplicam as técnicas expostas no capitulo anterior. Para realizar essa tarefa de forma mais organizada, as pesquisas foram divididas em: técnicas que modificam o lado servidor da aplicação, técnicas que modificam o lado cliente e *scanners* de vulnerabilidades Web. Entretanto, existem trabalhos que modificam tanto o lado cliente e o lado servidor.

3.1 Técnicas que modificam o lado servidor

As soluções aplicadas no lado servidor tem a vantagem de serem capazes de descobrir um número maior de vulnerabilidades e o benefício de que, caso uma vulnerabilidade seja resolvida, sua solução é automaticamente propagada para todos os usuários [27]. A maioria das soluções utilizadas no lado servidor é baseada na utilização de *proxies,* utilizados para classificar qualquer conteúdo fornecido pelo usuário.

3.1.1 Context Sensitive String Evaluation (CSSE)

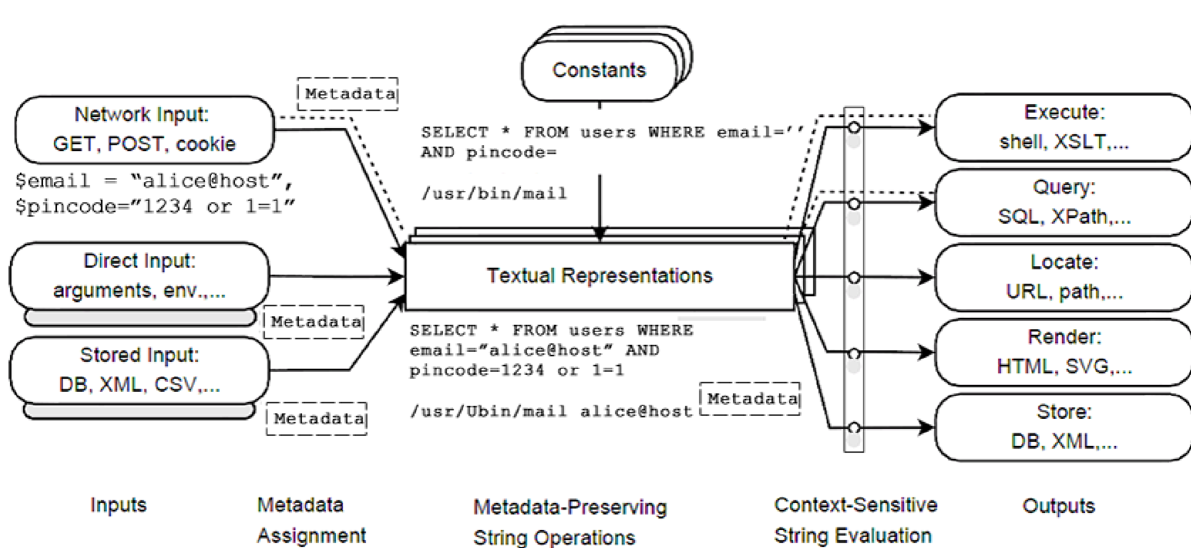

Figura 3.1: Arquitetura do CSSE [33].

O CSSE [29] trabalha no descobrimento da causa raiz de ataques XSS através da

serialização do conteúdo passado pelos usuários nos campos de entrada. Funciona modificando o interpretador da linguagem de programação para fazer análises baseadas em contexto. O CSSE é divido em fases, a Figura 3.1 mostra a arquitetura do CSSE.

Pode-se observar na Figura 3.1 que o CSSE possui três fases chamadas de *Metadata Assignment*, *Metadata-Preserving String Operations* e *Context-Sensitive String Evaluation*. Na primeira fase todas as variáveis são marcadas utilizando metadados com informações sobre a sua origem. A segunda fase é utilizada para checar e atualizar os metadados caso alguma operação seja realizada em alguma variável. A última fase do CSSE é utilizada quando os dados de alguma variável são utilizados em algum contexto seguro. Por exemplo, caso o valor de alguma variável seja utilizado para montar uma consulta SQL (*Structured Query Language*), então testes serão realizados no contexto de SQL para garantir o uso do valor desta variável.

A principal limitação do CSSE ocorre pelo fato de ser um trabalho que altera o interpretador da linguagem de programação no caso do trabalho o interpretador da linguagem PHP foi alterado. Essa tarefa não é facilmente replicada a outras linguagens de programação para Web como a linguagem Java, por exemplo.

3.1.2 Cross Site Scripting Detection System (XSSDS)

No XSSDS são desenvolvidas duas abordagens que podem ser utilizadas separadas ou em conjunto [40]. A primeira apresenta um detector para encontrar ataques XSS persistentes e a outra para ataques XSS não persistentes. Ambos são baseados em monitoramento de tráfego HTTP que dependem da correlação entre os parâmetros que são coletados. O XSSD funciona como um *proxy* e fica localizado entre o servidor e a aplicação *web*.

A primeira abordagem, responsável por detectar os ataques XSS não persistentes, é baseada na relação direta que esse tipo de ataque possui com os valores fornecidos nos campos de entrada e os *scripts* que são injetados. O método desenvolvido trabalha aplicando a técnica de casamento de *strings* entre a resposta da requisição e o *script* injetado para detectar se um ataque ocorreu.

Na abordagem de detecção de ataques persistentes, o lado servidor da aplicação é alterado para adicionar um componente que mantém uma lista de *scripts* conhecidos, um tipo de lista “branca”. Desta forma, qualquer *script* executado que não esteja nessa lista será

considerado um ataque XSS persistente.

A maior limitação desse trabalho é o grande número de falsos positivos gerado, principalmente, pelo segundo detector. Além disso, a utilização dos dois detectores em conjunto pode causar problemas de desempenho e escalabilidade.

3.1.3 XSS Guard

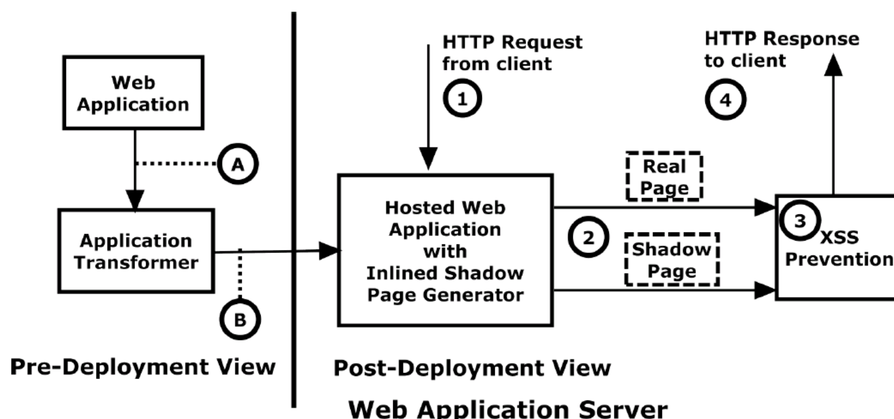

Figura 3.2: Arquitetura do XSS Guard [45].

O XSS Guard [41] funciona aprendendo dinamicamente o conjunto de *scripts* que uma aplicação *web* cria para uma requisição HTML. O XSS Guard trabalha identificando os *scripts* no lado servidor e removendo qualquer *script* na resposta da requisição que não esteja identificado a Figura 3.2 mostra a arquitetura do XSS Guard.

Para alcançar seus objetivos o XSS Guard possui dois componentes (*Pre-Deployment View* e *Post-Deployment View*). O primeiro prepara a aplicação para criar cópias das páginas geradas em cada requisição. Essas cópias são chamadas de *shadow pages* e são uma cópia original das páginas geradas por cada requisição. A única diferença é que as mesmas possuem apenas *scripts* autorizados.

No passo final do XSS Guard, a página original é comparada com sua respectiva *shadow page*. Caso algum ataque seja encontrado a resposta é modificada e enviada para o cliente com o ataque removido. A principal limitação do XSSGuard é o fato de ser aplicado apenas em aplicações desenvolvidas na linguagem Java e não pode detectar alguns tipos de ataques XSS.

3.1.4 BLUEPRINT

O BLUEPRINT [42] é uma ferramenta que também trabalha marcando as informações que não são confiáveis. É composto por um componente que fica no lado servidor e uma biblioteca de *scripts* que fica no lado cliente.

No lado servidor é criada uma árvore para analisar as informações que não são confiáveis e no lado cliente essa árvore é transmitida para o navegador sem caminhos vulneráveis. A Figura 3.3 mostra um exemplo de uso do BLUEPRINT.

```
1  // Code for trusted blog content above^^.
2  // Code to emit untrusted comments below:
3
4  <?php foreach ($comments as $comment): ?>
5      <li>
6          <?php echo($comment); ?>
7      </li>
8  <?php endforeach; ?>
9
10 // Code for trusted footer follows...
```

```
1  // Code for trusted blog content
2  // appears untransformed above^^.
3  <?php foreach ($comments as $comment): ?>
4      <li>
5          <?php
6  $model = Blueprint::cxPCData($comment);
7  echo($model);
8          ?>
9      </li>
10 <?php endforeach; ?>
```

Figura 3.3: Utilização do BLUEPRINT [15].

O lado esquerdo da Figura 3.3 mostra um exemplo de código que possui vulnerabilidade e o direito mostra outro exemplo de código, com a mesma função do anterior, protegido através da utilização do BLUEPRINT. A proteção ocorre na linha 6 onde o desenvolvedor da aplicação utilizou o BLUEPRINT para verificar o valor da variável “comment”.

As principais limitações do BLUEPRINT são que todos os navegadores teriam que atualizar suas bibliotecas de código para utiliza-lo e o tamanho da página também iria aumentar muito devido a codificação de texto adicionada a página a ser analisada.

3.1.5 Pixy

Pixy [43] é um protótipo *open source* desenvolvido em Java que usa análise estática para detectar vulnerabilidades XSS em páginas escritas na linguagem PHP. O Pixy trabalha marcando o caminho dos dados e checando se os dados fornecidos pelos usuários são utilizados em algum campo de entrada sem passar por validações.

Uma limitação do Pixy é que algumas variações de ataques XSS podem conseguir ultrapassar filtros facilmente. A diversidade dos algoritmos que fazem renderização no navegador e o desejo dos desenvolvedores de prover aplicações com conteúdo mais rico faz

com que o trabalho de filtragem, característica do Pixy, seja bem difícil [44].

3.2 Técnicas que modificam o lado cliente

As soluções no lado cliente, em sua maioria, também atuam como um *proxy,* geralmente criando regras manuais para mitigar as tentativas de ataques XSS. As soluções no lado cliente protegem eficientemente contra o vazamento de informações do ambiente do usuário [36]. Entretanto, definir uma solução no lado cliente não é uma tarefa fácil por causa da dificuldade em identificar *scripts* que sejam maliciosos.

3.2.1 Noxes

Noxes [27] é um *web proxy* que atua como *firewall* pessoal para aplicações *web* analisando as requisições HTTP, permitindo as conexões com base na política de segurança definida por meio de regras automáticas ou manuais e com a utilização de lista negra. O usuário pode criar regras para requisições de páginas *web* e inseri-las de forma interativa, cada vez que um pedido de conexão é realizado.

Para mitigar o envio de informações para outros domínios, a abordagem incorpora a análise de *links* internos e externos contidos na página, que são extraídos e analisados conforme as regras especificadas. Isso permite ao usuário acompanhar a análise durante em tempo de execução.

A ferramenta analisa as requisições de páginas em várias etapas, dentre as quais, verifica se é um *link* interno. Caso não seja, o Noxes verifica se há alguma regra temporária ou permanente para a conexão, que executa o bloqueio ou submete a decisão ao usuário. Conforme os próprios autores, um dos grandes problemas dessa abordagem corresponde à geração de um grande número de alertas, exigindo a intervenção do usuário.

Outro ponto a ser considerado é que a ferramenta proposta regula a atividade que ocorre ao longo da rede, mas não aborda o comportamento local do código *JavaScript* fazendo com que o Noxes não seja capaz de detectar alguns tipos de ataques XSS como o ataque de negação de serviços e os ataques que modificam o conteúdo das páginas.

3.2.2 Automatic Detection/collection system for XSS

Este trabalho [45] apresenta um conjunto de *proxies* localizado no lado cliente que executam um processo de análise das informações trocadas entre o servidor da aplicação e o

navegador. Esse processo é projetado para detectar requisições feitas pelo atacante para o usuário.

Quando o primeiro *proxy* detecta alguma vulnerabilidade, informações sobre a vulnerabilidade como nomes do *host* são coletadas e enviadas para um servidor de banco de dados XSS que compartilha essas informações com outros *proxies*.

Caso uma requisição maliciosa seja detectada, os caracteres dessa requisição são codificados na tentativa de evitar o sucesso do ataque. As principais limitações dessa abordagem é que a mesma só pode ser utilizada para detectar ataques XSS não persistentes e confia na configuração feita pelos usuários, que é uma tarefa passível de erros.

3.2.3 xJS

O xJS [46] propõe uma metodologia que modifica o navegador para isolar o código das aplicações *web* de códigos maliciosos. A proposta do xJS é trabalhar randomizando os ataques de uma forma que não possam ser executados.

O xJS é baseado no conceito de operadores de isolamento e políticas baseadas em ações. O xJS modifica o navegador e o servidor para adicionar chaves que irão remover o isolamento do código e das politicas de segurança. Um *script* será executado apenas se conseguir produzir um resultado que satisfaça uma política e se as chaves para retirar o isolamento funcionarem. O xJS não foi projetado para detectar os ataques XSS persistentes.

3.3 Scanners de vulnerabilidades web

Pode-se observar que todos os trabalhos citados anteriormente modificam algum aspecto no lado servidor ou no lado cliente da aplicação para combater os ataques XSS. Tal fato proporcionou o surgimento de ferramentas automatizadas que não precisam modificar nenhum aspecto da aplicação. Essas ferramentas são chamadas de *scanners* de vulnerabilidades *web*.

Um *scanner* de vulnerabilidades *web* é um programa automatizado que examina as aplicações *web* na busca por vulnerabilidades [47]. Alguns *scanners* procuram também por outras vulnerabilidades que não são especificas da *web*, como *buffer overflow*.

Um *scanner* explora a aplicação obtendo informações das suas páginas e aplicando testes de penetração, que nada mais são que uma análise ativa da aplicação através de ataques.

Esta análise envolve a geração de dados de entrada adequados e o estudo da resposta gerada pela requisição HTTP.

Existem vários tipos de teste aplicados por *scanners* de vulnerabilidades. Um dos testes mais comuns, chamado de *fuzzing*, consiste em um método de testes de software em que a aplicação testada é bombardeada por casos de testes gerados por outro programa [48].

Atualmente existem muitos *scanners* disponíveis, alguns comercias como Appscan [49], Acunetix [31], N-Stalker [50], WebCruiser [51] e outros de código aberto como o Secubat [52], PowerFuzzer [53], WebSecurify [54] e o WebXSSDetector [24]. Alguns dos trabalhos citados serão utilizados nesse trabalho e serão explicados a seguir, porém a maioria das ferramentas possui pouca documentação fazendo com que apenas algumas características e um *screenshot* da tela inicial sejam apresentados.

3.3.1 Acunetix

O Acunetix é o mais famoso *scanner* de vulnerabilidade comercial que checa diferentes tipos de ataques web, incluindo ataques XSS e SQL injection. O Acunetix pode ser utilizado executando uma tecnologia denominada de AcuSensor que efetua os testes caixa-preta e também analisa o código fonte de aplicações .NET e PHP através da inserção de sensores, supostamente identificando rapidamente as vulnerabilidades e apresentando um número baixo de falsos positivos. A versão grátis do Acunetix detecta apenas ataques XSS e será utilizada nesse trabalho. A Figura 3.4 mostra a tela inicial do Acunetix.

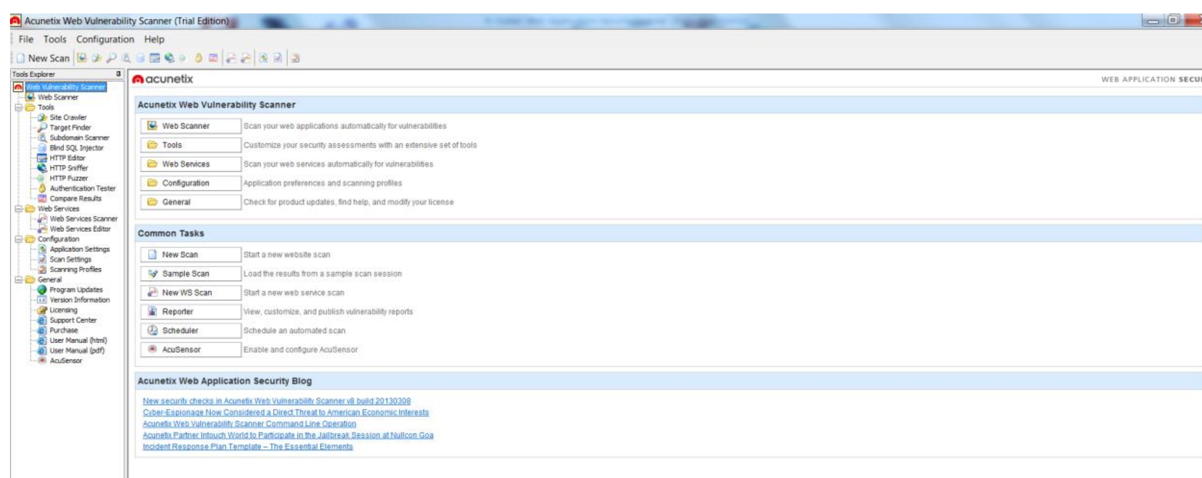

Figura 3.4: Tela inicial do Acunetix.

3.3.2 N-Stalker

O N-Stalker também é um *scanner* comercial e provem um conjunto de checagens para aprimorar a segurança das aplicações *web*. Essa ferramenta permite que os usuários criem as suas próprias políticas e requisitos. A versão para testes do N-Stalker possui uma licença que dura apenas sete dias. A Figura 3.5 mostra a tela inicial do N-Stalker.

Figura 3.5: Tela inicial do N-Stalker.

3.3.3 Rational AppScan

O AppScan é um *scanner* da IBM liberado para testes avançados em aplicações *web*. Possui a capacidade de gerar recursos para facilitar futuras correções de vulnerabilidades. O Rational AppScan possui uma licença de tempo indeterminado para sua versão padrão. Porém com essa versão apenas um site padrão fornecido pelo software pode ser testado. A Figura 3.6 mostra a tela inicial do AppScan.

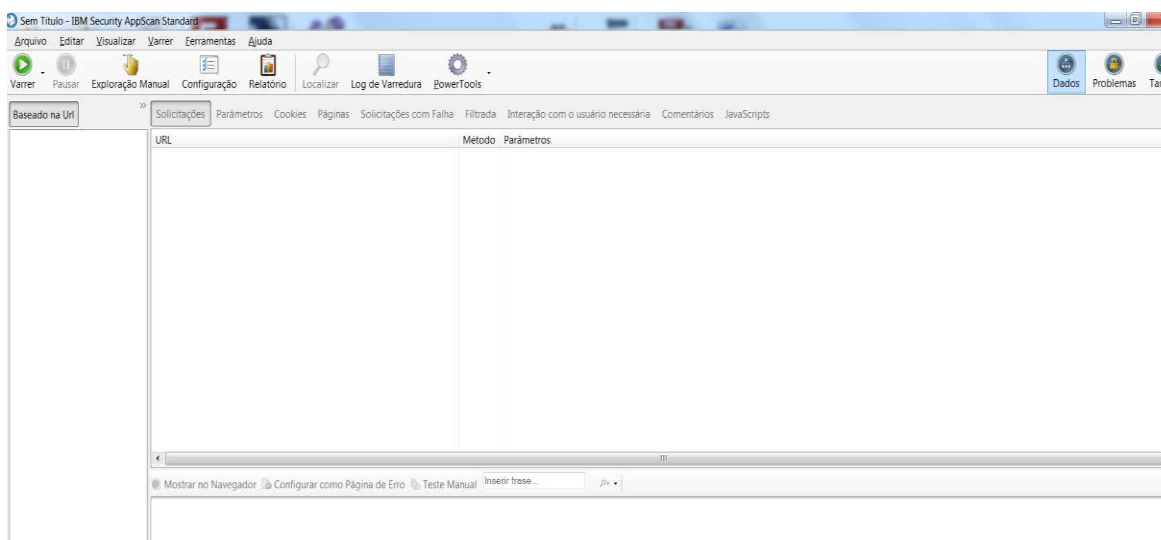

Figura 3.6: Tela inicial do AppScan.

3.3.4 WebCruiser

O WebCruiser é uma ferramenta para detecção de vulnerabilidades em aplicações *web*, incluindo ataques XSS. Uma das vantagens que o mesmo possui é que em algumas vulnerabilidades o WebCruiser disponibiliza provas de conceito (POC) sobre o ataque que foi gerado possibilitando que os profissionais de segurança reproduzam os ataques. A Figura 3.7 mostra a tela inicial do WebCruiser.

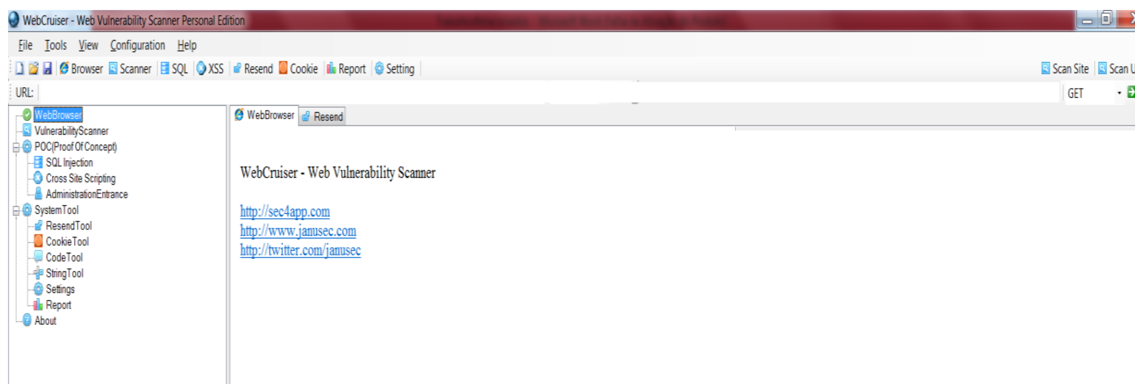

Figura 3.7: Tela inicial do WebCruiser.

3.3.5 SecuBat

O SecuBat é um *scanner* utilizado para descobrir vulnerabilidades XSS. A arquitetura em camadas do SecuBat é apresentada na Figura 3.8.

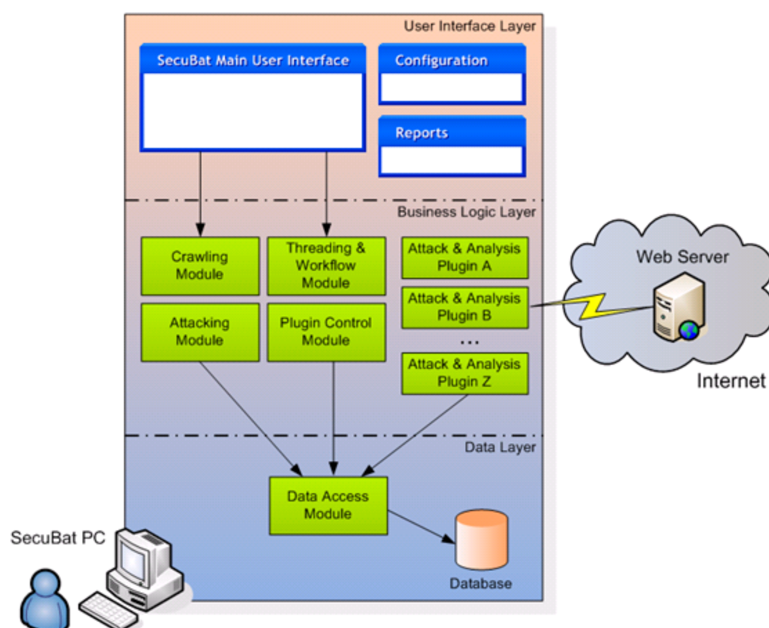

Figura 3.8: Arquitetura do SecuBat [16].

A arquitetura do SecuBat é dividida em três camadas chamadas de *User Interface Layer*, *Business Logic Layer* e *Data Layer*. A primeira camada é responsável pela configuração do escaneamento. Nela o usuário informa as informações necessárias para o SecuBat iniciar a sua execução como o endereço da aplicação que será testada. A segunda camada consiste de módulos para realizar a extração de informações da aplicação, realizar os testes e configurar os *plug-ins* de ataques. Por fim a última camada é responsável por armazenar todas as informações que são necessárias durante a execução do SecuBat. Uma das limitações do SecuBat é que o mesmo foi projetado para focar apenas na descoberta de ataques XSS não persistentes.

3.3.6 PowerFuzzer

O PowerFuzzer trata-se de uma ferramenta de código livre que realiza a busca por diversas vulnerabilidades, incluindo ataques XSS. Uma das vantagens do PowerFuzzer é que o mesmo apresenta na tela todas as requisições que realiza na aplicação testada. A Figura 3.9 apresenta a tela inicial do PowerFuzzer.

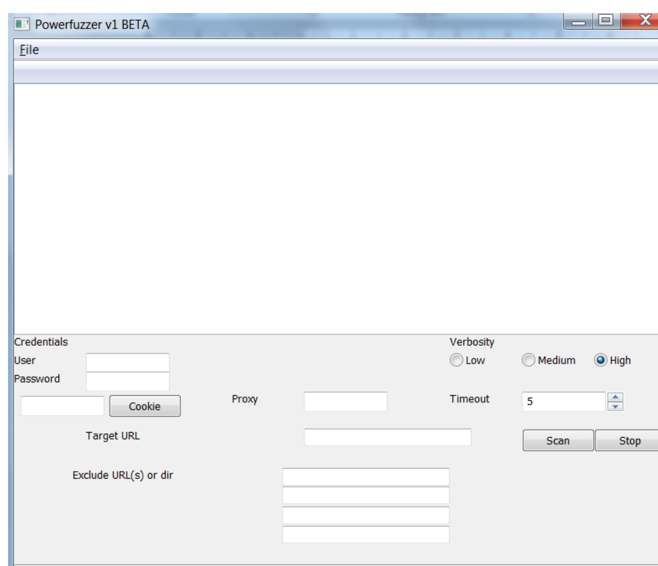

Figura 3.9: Tela inicial do PowerFuzzer.

3.3.7 WebSecurify

O WebSecurify é uma ferramenta que também realiza buscas por diversas vulnerabilidades, como XSS e SQL *Injection*. Na sua tela inicial encontra-se um assistente com três ícones. No primeiro ícone o usuário informa o endereço da aplicação que deve ser testada, o segundo ícone corresponde ao processo de execução do *scanner* que mostra uma

barra de progresso para o usuário visualizar o fim do processo e o terceiro ícone é responsável pela geração dos relatórios, a Figura 3.10 mostra a sua tela inicial.

Figura 3.10:Tela inicial do WebSecurify.

3.3.8 WebXSSDetector

O WebXSSDetector é um trabalho acadêmico que propôs o desenvolvimento de um *scanner* de vulnerabilidades para combater apenas ataques XSS. A motivação dos envolvidos se deu pelo fato de pesquisas atuais constatarem que as ferramentas existentes não produzem resultados satisfatórios.

A arquitetura do WebXSSDetector é dividida em cinco módulos. O módulo de extração, injeção, detecção, base de dados e gerador de código. O primeiro módulo é responsável pela extração das informações das páginas, o segundo módulo (injeção) trabalha em parceria com o gerador de código para gerar os ataques de forma adequada e aplica-los nos campos coletados.

O módulo de detecção é responsável por verificar se um ataque XSS foi bem sucedido. Por fim a base de dados é utilizada para armazenas as informações necessárias durante a execução da ferramenta.

3.4 Considerações Finais

Com relação aos trabalhos que modificam o lado cliente e o lado servidor, Saha [26] avaliou dez importantes trabalhos propostos na literatura para detectar XSS. A análise mostra que sete metodologias propostas apresentaram baixa capacidade para resolver ou detectar ataques XSS. Dentre os quais, alguns trabalhos citados aqui como o Pixy e o CSSE.

No trabalho de Saha foram avaliados oito metodologias sob o aspecto de falso positivo, sendo que seis metodologias apresentaram taxas elevadas de falso alarme, dentre os quais a ferramenta Noxes. Além disso, muitas propostas não detectam ataques XSS persistentes e apresentam algumas limitações práticas, por dependerem de muitas alterações

nos navegadores como o XSS Guard ou alterações significativas no código fonte das aplicações, como por exemplo, o BLUEPRINT ou mesmo alterações no lado cliente e no lado servidor da aplicação como mostrado na Tabela 3.1.

Tabela 3.1: Modificações realizadas pelos trabalhos relacionados.

#	Ferramenta	Alterações	Detecta ataques persistentes
1	CSSE	Interpretador	Não
2	XSSDS	Servidor	Sim
3	XSS Guard	Servidor/Navegador	Sim
4	BLUEPRINT	Servidor/Navegador	Sim
5	Pixy	Servidor	Sim
6	Noxes	Firewall	Não
7	Automatic detection/collection	Proxy	Não
8	Xjs	Servidor/Navegador	Não

Com relação aos *scanners* de vulnerabilidades *web*, observa-se que alguns são proprietários e possuem um custo muito alto. Além disso, os *scanners* existentes não são capazes de detectar os ataques XSS com eficiência apresentando um alto número de falsos positivos como reportado em [12] e [13].

O estudo bibliográfico realizado neste trabalho demonstra que, apesar de existirem muitas metodologias propostas para mitigar o problema de XSS, muito trabalho ainda é necessário para mitigar vulnerabilidades XSS. O próximo capítulo apresenta a metodologia proposta neste trabalho para combater ataques XSS.

Capítulo 4

Detecção de Vulnerabilidades XSS em aplicações Web

Este Capítulo está dividido da seguinte forma: a primeira seção apresenta uma breve introdução sobre os requisitos que nortearam a concepção da metodologia para realizar testes de vulnerabilidades XSS em aplicações web. Uma visão geral dessa metodologia, uma descrição detalhada de suas principais fases e métodos e, por fim, os passos que foram realizados para o desenvolvimento de um protótipo que utiliza a metodologia proposta.

4.1. Metodologia Proposta

Como mencionado nos primeiros capítulos deste trabalho, um dos ataques mais comuns em aplicações *web* é o XSS. Este ataque permite que o atacante injete códigos maliciosos dentro de páginas visitadas por usuários, possibilitando o acesso a informações confidenciais.

Devido aos grandes prejuízos financeiros, tanto para os usuários como para as companhias, ferramentas automatizadas têm sido empregadas no combate a ataques XSS. Tais ferramentas procuram detectar o ataque antes que o mesmo possa ser explorado pelos atacantes. Contudo, de acordo com Bau et al. [12], o nível de detecção dos ataques XSS persistentes é de apenas 15%. Além disso, a maioria das soluções falha ao testar partes da aplicação quando essas envolvem o preenchimento de formulários complexos [15]. Uma das razões é que não existe um critério na aplicação dos testes nos pontos de vulnerabilidade [14].

Na tentativa de minimizar tais deficiências, este trabalho propõe uma metodologia para detecção de vulnerabilidades XSS em aplicações *web*. Esta metodologia é dividida em fases que serão responsáveis pela extração e análise das informações contidas nas páginas de uma determinada aplicação *web*.

Como observado, um importante requisito da metodologia proposta é garantir o

correto preenchimento dos campos de entrada da aplicação com informações válidas. Tal abordagem visa minimizar o problema de preenchimento de formulários complexos, identificado por Mcallister et al. [15], através de uma análise qualificativa nos campos de entrada da aplicação, garantindo que os testes nos formulários sejam submetidos corretamente.

Os testes são submetidos através da inserção de códigos maliciosos nos pontos de vulnerabilidade que usualmente um atacante utilizaria. O objetivo é verificar se esses códigos são executados em alguma parte da aplicação, sem precisar de detalhes do código fonte.

Para conduzir os testes de vulnerabilidades XSS em aplicações *web*, os seguintes problemas devem ser levados em consideração:

1. Como identificar todos os pontos de vulnerabilidade?
2. Como acessar os formulários da página e preenche-los?
3. Como coletar todas as páginas da aplicação?
4. Quais ataques utilizar em cada ponto de vulnerabilidade?
5. Como identificar se o ataque foi executado corretamente?

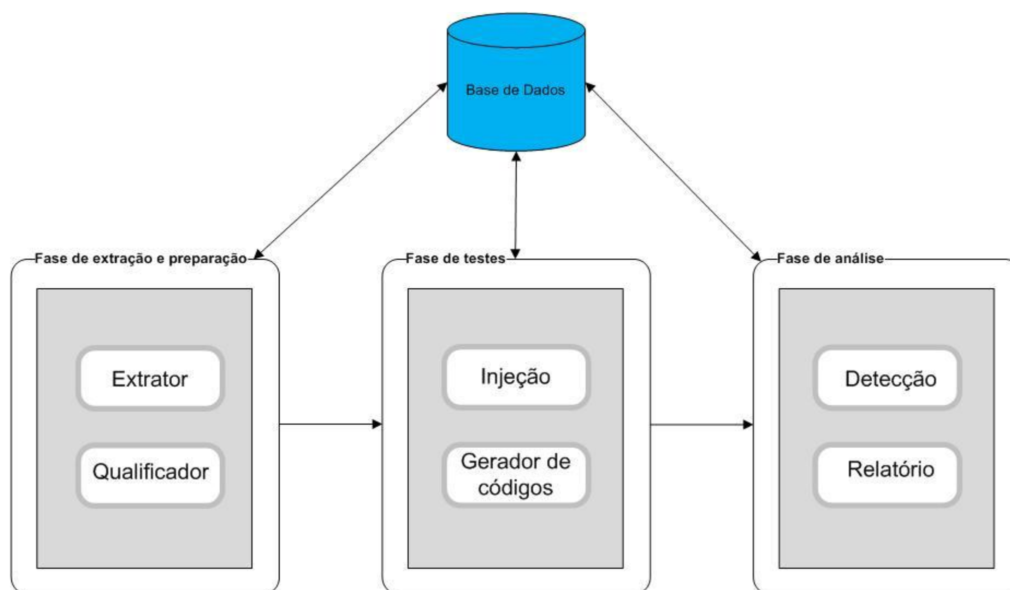

Figura 4.1: Metodologia.

Para lidar com estes questões, a metodologia proposta foi dividida em três fases: extração e preparação, responsável pela extração do conteúdo e identificação dos pontos de vulnerabilidades; testes, responsável por gerar e realizar os testes necessários de acordo com o tipo entrada apontados na fase anterior; e a análise, onde é realizada a análise dos resultados e

emissão dos relatórios. A Figura 4.1 apresenta uma visão geral das fases que compõem a metodologia proposta. As seções seguintes apresentam uma visão detalhada dos componentes e métodos empregados em cada fase.

4.2. Fase de Extração e Preparação

A Fase de Extração e Preparação é responsável pela identificação, coleta e análise das informações necessárias à avaliação da aplicação *web*. Esta fase é composta pelas etapas de extração de conteúdo e qualificação dos pontos de vulnerabilidade.

O diferencial desta fase, com relação a pesquisas anteriores, encontra-se na qualificação dos pontos de vulnerabilidade que visa preencher cada campo com apenas informações válidas para garantir a submissão de formulários complexos e mitigar o problema encontrado por Mcallister et al. [15]. As etapas da fase serão detalhadas a seguir.

4.2.1. Extrator de Informações

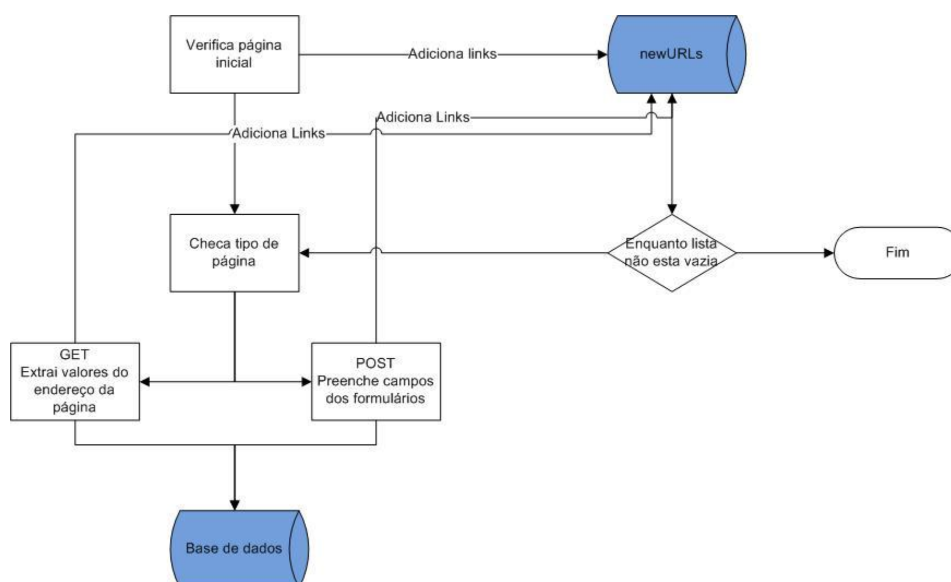

Figura 4.2: O processo de extração de informações na página *web*.

O processo de extração é responsável por coletar as informações necessárias das páginas que serão testadas tais como seus *links* e parâmetros de seus formulários, bem como identificar quais métodos que o formulário irá executar. Todas essas informações são armazenadas em uma base de dados e são usadas pelo componente de qualificação (detalhado na seção 4.2.2) para identificar os principais pontos de vulnerabilidades e aumentar a garantia de submissão dos formulários em uma página *web* na tentativa de testar a aplicação por

completo. A Figura 4.2 exemplifica o processo de extração de informações.

O processo se inicia com uma página inicial para ser visitada. O extrator então faz uma requisição a um servidor HTTP, baixa o documento e dele extrai os *links*, conteúdo e outras informações. Os endereços dos *links* extraídos são guardados para que se possa acessá-los posteriormente.

Como a maioria dos coletores de página existentes, o processo prevê dois tipos de requisições HTTP, via os métodos POST e GET. Caso a requisição inicial seja realizada via o método GET, os parâmetros do endereço da página são extraídos e inseridos na base de dados. Caso a requisição seja do tipo POST, os parâmetros dos formulários são recuperados e também armazenados na base de dados.

Durante o processo de extração de informações, novas páginas da aplicação *web* avaliada poderão ser encontradas e deverão passar pelo mesmo procedimento.

4.2.2. Qualificador de Pontos de Vulnerabilidade

O processo de qualificação de entradas é responsável por fazer uma análise das páginas no navegador do cliente, identificando cada ponto de vulnerabilidade e seus possíveis valores para qualificar o seu preenchimento durante o processo de teste. Esta análise ocorre através do estudo da árvore DOM do ponto de vulnerabilidade em questão, seguindo a ordem de caminhamento na árvore da esquerda para a direita. Dessa forma são analisados primeiros os elementos à esquerda do ponto de vulnerabilidade em análise, depois o próprio ponto de vulnerabilidade e, por fim, os elementos da direita.

O objetivo desse processo é identificar quais entradas são mais apropriadas no preenchimento de um formulário. Por exemplo, o processo de qualificação do elemento *input*, mostrada na árvore DOM da Figura 4.3 se inicia com a recuperação do valor do campo localizado a esquerda. Neste caso, o valor do campo de texto "email". Após a recuperação do valor do campo é checado se existe algum qualificador correspondente a este valor na base de dados.

Caso exista, o valor do qualificador é recuperado e atribuído ao valor do elemento que esta sendo qualificado. Caso contrário, a análise continua verificando os valores dos atributos do elemento seguinte. Nesse caso, os valores dos atributos *name* e *type* do elemento *input*. Caso não exista um qualificador correspondente aos valores desses atributos, a análise

continua verificando os campos a direita do elemento *input*. Por fim, caso não exista um qualificador correspondente a nenhum dos valores testados o campo recebe como entrada um valor padrão.

No caso da Figura 4.3, o texto *email* seria utilizado para análise e o qualificador de e-mail correspondente seria encontrado, fazendo com que o elemento *input* recebesse o valor do qualificador, por exemplo, “teste@gmail.com”.

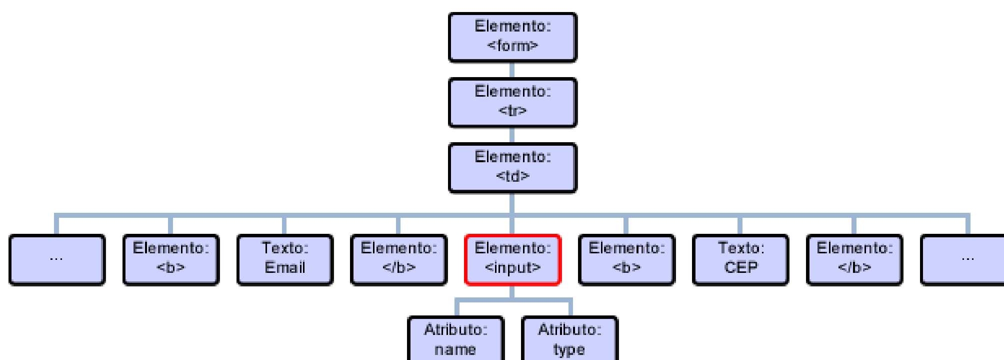

Figura 4.3: Análise do elemento *input*.

Após qualificar todos os campos do formulário da página é realizada uma submissão do formulário para identificar se a página resposta já passou pelo extrator de informações. Caso isso não tenha ocorrido, a página é marcada para passar pelo mesmo processo.

4.3. Fase de Testes

A Fase de Injeção e Testes é responsável pela injeção e execução dos códigos de ataque XSS que serão inseridos em cada ponto de vulnerabilidade da página *web* avaliada. Basicamente esta fase é composta pelos componentes de injeção e geração de código.

O diferencial desta fase encontra-se na aplicação de um pré-teste em cada ponto de vulnerabilidade no intuito de verificar as informações encontradas após a submissão do formulário. O objetivo deste pré-teste é minimizar o problema identificado por Kosuga [14]. A fase será detalhada a seguir.

4.3.1. Injeção de Código

O processo de injeção de código gerado é utilizado para ajudar na análise do comportamento da página web na presença de ataques XSS. Nesta etapa, o código gerado é inserido na página *web* e consequentemente executado. Os pontos de vulnerabilidades onde

serão inseridos os códigos em cada página foram identificados na fase de extração e preparação e armazenados na base de dados.

Diferentemente da maioria das ferramentas de teste de vulnerabilidade em aplicações *web*, a metodologia proposta propõe o uso de um processo de seleção ataques. Os testes são feitos de acordo com o tipo de ponto de vulnerabilidade que será testado. Por exemplo, caso o ponto de vulnerabilidade encontre-se no título da página apenas ataques referentes ao título serão utilizados. O objetivo dessa abordagem é evitar que um grande número de ataques inapropriados e desnecessários seja executado durante o processo de avaliação. A Figura 4.4 exemplifica o processo de injeção de código.

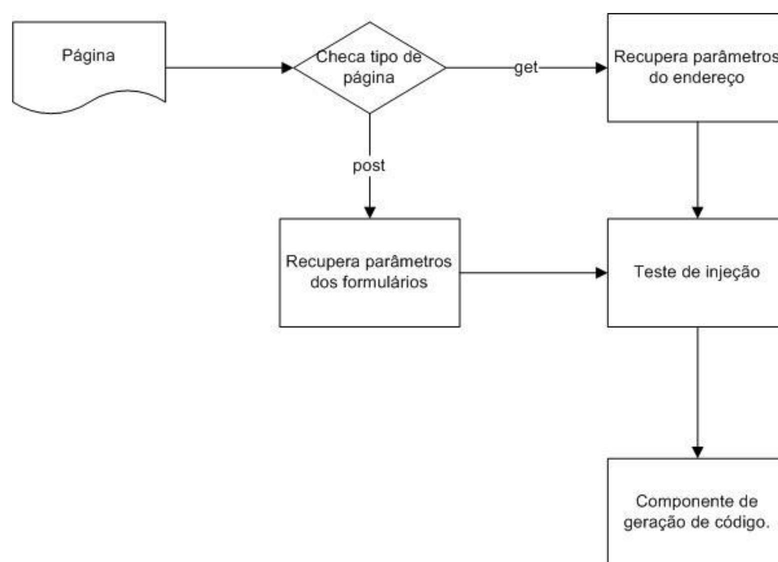

Figura 4.4: Processo da fase de injeção.

Da mesma forma que o processo de extração, o processo de injeção considera os diferentes métodos de requisição HTTP (métodos POST e GET). Os parâmetros do endereço da página (método GET) ou os campos de formulários (método POST) são recuperados da base de dados. O próximo passo é injetar um código malicioso nos pontos de vulnerabilidade recuperados da página web avaliada.

Antes de inserir o código propriamente dito, um teste de injeção simples (por exemplo, um *alert*) é adicionado para cada ponto de entrada para determinar a posição onde o código irá aparecer na página de resposta. Esta informação possibilitará ao gerador de código criar os ataques mais apropriados e ao componente de detecção saber a possível posição de um código malicioso na página. Tal abordagem também tem sido empregada na ferramenta

WebXSSDetector e é conhecida como “*dummy injection code*” [24].

4.3.2. Geração de Código Malicioso

A etapa de geração automática de código malicioso é responsável pela criação dos códigos que serão inseridos nos pontos de vulnerabilidade. Como os ataques XSS possuem uma enorme variedade de formas de serem executados, um gerador de código é necessário para simular o maior número de variações de ataques possíveis.

O processo de geração de código é simples e funciona da seguinte maneira: primeiro, é obtida uma lista inicial de ataques XSS conhecidos. Essa lista pode ser obtida, por exemplo, a partir de repositórios mantidos por grupos de segurança como OWASP [55]. Em seguida, variações dos códigos maliciosos iniciais (ver exemplo na Tabela 4.1) são geradas de acordo com as informações de posicionamento obtidas durante o processo de teste de injeção.

O gerador de código malicioso identifica cada ataque gerado através de um contador (ID Ataque). Esse identificador é utilizado para auxiliar a Fase de Detecção na procura dos ataques XSS que foram bem sucedidos. A Tabela 1 mostra exemplos de variações de um ataque XSS que utiliza a *tag javascript*.

Tabela 4.1: Exemplos de variações de ataque com a tag javascript.

#	Descrição	Variações do código geradas
1	Caso básico	javascript:alert(1)
2	Caixa alta	JaVasCriPt:alert(1)
3	Caso com o caracter especial /t	ja\tvascript:alert(1)
4	Caso com o caracter especial /n	ja\nvascript:alert(1)
5	Codificação hexadecimal	javascrip t:alert(1)
6	Codificação UTF-8	javascrip t:alert(1)
7	Codificação UTF-8 longa	javas cript :aler t(1)

As variações de ataques são geradas pelo fato dos atacantes empregarem diferentes formas de codificações para burlar os filtros que são desenvolvidos pelos programadores. Na tentativa de simular o comportamento real de um atacante, o gerador de código gera diferentes variações de um mesmo ataque.

4.4. Fase de Análise

A Fase de Análise é responsável por realizar a detecção dos ataques XSS através da análise dos resultados das fases anteriores e posteriormente gerar relatórios para que os dados possam ser consultados.

O diferencial desta fase encontra-se no fato da metodologia realizar a detecção de ataques em toda a aplicação, a maioria das pesquisas anteriores verifica apenas a resposta direta gerada pela requisição. Com essa abordagem a metodologia visa melhorar o nível de detecção de ataques XSS persistentes e mitigar o problema encontrado por Bau et al. [12], o processo de detecção será detalhado a seguir.

4.4.1. Detecção de Ataques

O processo de detecção de vulnerabilidades é responsável pela análise dos resultados da fase de injeção e testes. Uma página é dita vulnerável quando um ataque XSS é executado em pelo menos em um dos pontos de vulnerabilidade, a Figura 4.5 exemplifica o processo de detecção.

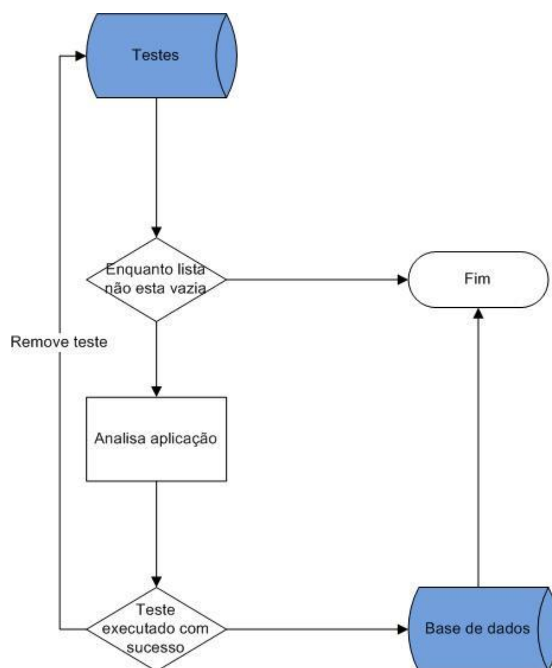

Figura 4.5: Processo de detecção.

O processo se inicia através da obtenção da lista de testes gerada na fase anterior. De posse dessa lista ocorre uma iteração onde os valores dos testes serão inseridos em cada ponto

de vulnerabilidade.

O próximo passo é executar o teste para verificar o seu resultado. Caso o resultado seja positivo ocorre uma atualização na base de dados e o processo continua no próximo ponto de vulnerabilidade, caso contrário, o processo continua até que a lista de testes fique vazia.

4.5. Base de Dados

No intuito de salvar as informações adquiridas durante as fases é necessária uma base de dados. Além de salvar as informações a utilização de uma base de dados possui outros benefícios tais como:

- Os dados podem ser facilmente recuperados através de consultas SQL.
- Depois da condução de um teste, os ataques podem ser recuperados da base de dados a qualquer momento e reconstruídos. Com isso os profissionais de segurança podem entender o ataque e tentar resolve-lo.
- Relatórios podem ser gerados com facilidade através da obtenção dos dados.

4.6. ETSSDetector

Esta seção apresenta o projeto e a implementação de uma ferramenta de teste automatizada para detecção de vulnerabilidade em aplicações web, denominada de ETSSDetector. Baseada na metodologia proposta, a ferramenta ETSSDetector é implementada através de um conjunto de módulos como mostra a Figura 4.6. Cada módulo é listado a seguir:

- Interface do usuário: Utilizada para que o usuário preencha as informações necessárias para o inicio da execução da ferramenta.
- Simulador: Responsável por reproduzir a utilização de um navegador.
- Módulo de extração: Responsável por coletar as páginas da aplicação.
- Módulo de qualificação: Utilizado para fazer a análise DOM das páginas *web* e minimizar a submissão de formulários complexos.
- Módulo de testes: Responsável por simular os ataques gerados pelo módulo de geração de código.
- Módulo de análise: Utilizado para detectar se os ataques foram executados.

- Base de dados: Módulo responsável por armazenar as informações obtidas sobre as páginas da aplicação testada.

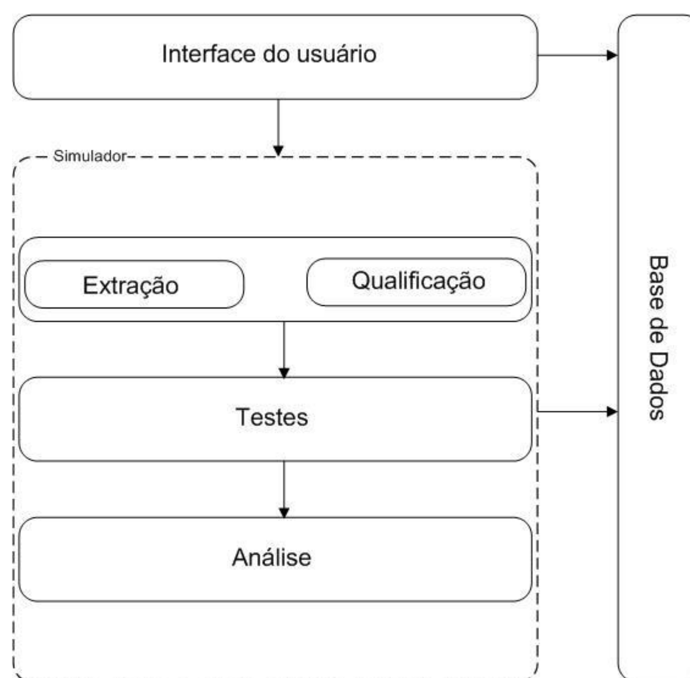

Figura 4.6: Arquitetura do ETSSDetector.

O ETSSDetector foi implementado na linguagem Java pelo fato de ser uma linguagem *open source* e independente de plataforma, ou seja, o ETSSDetector roda em qualquer computador que possua a JRE (*Java Runtime Environment*) instalada. Cada módulo será detalhado nas seções seguintes.

4.6.1. Simulador

Para detectar os ataques XSS persistentes e não persistentes, o ETSSDetector deve ser capaz de simular a utilização da aplicação *web* que estará sendo testada e realizar as mesmas ações de um usuário.

Para simular esse comportamento, o simulador de browser HTMLUNIT [56] é utilizado. Através desse simulador é possível modelar todo o documento HTML e interagir com as páginas, preenchendo formulários e clicando em *links* simulando o comportamento de um navegador real.

Essa simulação ocorre através da criação de um objeto do tipo *WebClient* para

possibilitar a interação entre o ETSSDetector e a aplicação, e objetos do tipo *HTMLPage* para representar cada página extraída. Através do objeto HTMLPage a análise da página pode ser realizada de duas formas diferentes:

- Analisando o texto da resposta HTTP: Quando o simulador executa uma ação navegando para outra página a resposta HTTP pode ser recuperada como texto e operações de texto podem ser utilizadas.
- Analisando a árvore DOM: Toda a estrutura de uma página pode ser representada em uma árvore DOM. Usando-a é possível distinguir as *tags* HTML dos textos através dos elementos. Por exemplo, é possível determinar a presença de um texto em uma página analisando a sua árvore DOM e verificando seus campos de texto.

4.6.2. Módulo de Extração

O ETSSDetector inicia recebendo algumas informações do usuário como o endereço inicial da aplicação *web* que será testada e páginas que devem ficar fora do processo de avaliação como páginas de *logout* e páginas que não possuem interação com o usuário como mostra a Figura 4.7. De posse dessas informações o ETSSDetector inicia a sua execução no módulo de extração.

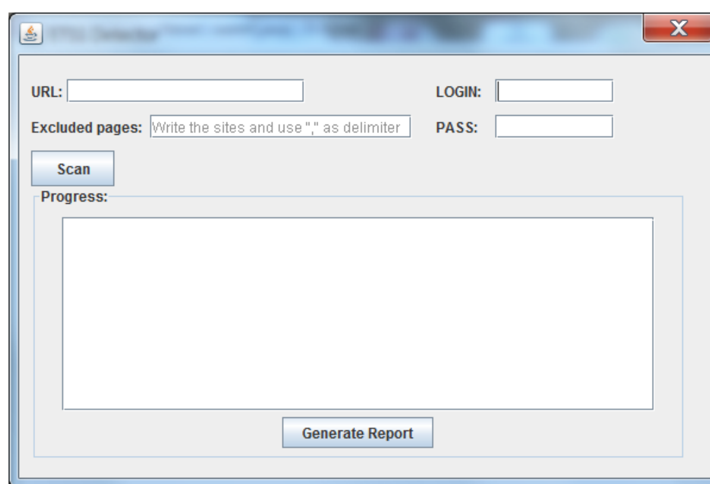

Figura 4.7: Tela de configuração inicial.

O módulo de extração conta com o auxilio de uma estrutura de dados do tipo lista chamada newURLs. Essa estrutura serve para armazenar todas as páginas da aplicação que irão ter o seu conteúdo extraído. A primeira página adicionada nesta estrutura é o endereço (URL) da aplicação. Enquanto existirem valores nessa lista o processo será executado.

A partir de uma iteração na lista de newURLs e obtenção dos endereços, o ETSSDetector inicia a extração das informações instanciando um objeto do tipo HTMLPage por vez para cada endereço. A partir desse objeto são extraidos seus *links* através de uma chamada direta ao método *getAnchors*. Todos os *links* são adicionados na lista de *newURLs,* com exceção de *links* repetidos e *links* de e-mail.

Em seguida, inicia-se um busca por parâmetros no endereço da página que esta sendo analisada. Todos os parâmetros encontrados são salvos na tabela webpage da base de dados (ver Tabela 4.3). As seções seguintes apresentam detalhes sobre o processo de extração de valores presentes na URL e nos formulários de uma página.

4.6.2.1 Extraindo valores do endereço da página.

Os parâmetros que se encontram no endereço da página, também conhecidos como *querystrings*, são um dos possíveis pontos de vulnerabilidades de ataques XSS, principalmente os ataques não persistentes. A *querystrings* é formada por pares ordenados (atributo, valor) que ficam separados no endereço pelo caracter "&". Os campos atributo e valor são separados pelo caracter "=", como mostra a Figura 4.8.

www.exemplo.com/exemplo.php?nome=teste&idade=10

Figura 4.8: Exemplo de endereço com parâmetros.

Para extrair os parâmetros do endereço das páginas foi criado um método, utilizando a função *split* da classe *String* do Java, que quebra o texto em várias partes. Assim, o valor do endereço da página é primeiramente quebrado para recuperar os parâmetros após o caracter "?". Seguindo o exemplo da Figura 4.8, um vetor com o valor "nome=teste&idade=10" é recuperado.

Em seguida, o mesmo método é utilizado para separar os pares ordenados. De acordo com o exemplo, o vetor teria tamanho dois com a posição zero possuindo o valor "nome=teste" e a posição um o valor "idade=10". Por último, esse procedimento é novamente repetido para separar os parâmetros e seus respectivos valores. Tais informações são então armazenadas na tabela querystring da base de dados (ver Tabela 4.6).

4.6.2.2 Extraindo valores dos formulários da página.

Os formulários de cada página são recuperados através de uma chamada direta ao

método *getForms* do objeto *HTMLPage*. Cada formulário recuperado é identificado (ID form) e inseridos na tabela form da base de dados (ver Tabela 4.4).

A partir dos formulários da página salvos inicia-se a extração dos campos (pontos de vulnerabilidades) que compõe o formulário. O método *getHtmlElementDescendants* é usado para essa finalidade. Os campos extraídos são:

1. *Text*: campo de texto que possuem apenas uma linha.
2. *Password*: campo do tipo senha.
3. *Hidden*: campo de texto configurado para não aparecer na página.
4. *Textarea*: campo do tipo texto que possuem mais de uma linha.
5. *Checkbox*: campo do tipo caixa de seleção.
6. *Radio*: campo do tipo botão de seleção.

Após a obtenção dos campos do formulário inicia-se o processo de qualificação (ver seção 4.6.3). O objetivo é atribuir a cada campo valores apropriados quando o formulário for submetido na fase de testes. Os pontos de vulnerabilidade do tipo *Hidden, Checkbox e Radio* não precisam passar pelo módulo de qualificação. Isso ocorre pelo fato do ponto de vulnerabilidade do tipo *Hidden* ser um campo auxiliar para o desenvolvedor, sendo invisível para o usuário do sistema. Os campos *Radio* e *Checkbox* não são qualificados por serem apenas campos de seleção.

Após a qualificação os pontos de vulnerabilidade dos formulários são inseridos na tabela input da base de dados (ver Tabela 4.5).

4.6.3 Módulo de qualificação.

O módulo de qualificação trabalha com a lista dos campos extraída dos formulários da página durante o processo de extração. De posse dessa informação, o módulo de qualificação realiza uma pesquisa verificando os elementos à esquerda e à direita do ponto de vulnerabilidade em análise, bem como os seus atributos, para determinar o valor mais adequado de preenchimento. Os valores de preenchimento usado pelos qualificadores ficam armazenados na tabela qualifer da base de dados (ver Tabela 4.7) e possuem um identificador único, nome e valor.

Caso exista algum qualificador correspondente, o seu valor é retornado e atribuído ao ponto de vulnerabilidade. Caso contrário, uma cadeia de caracteres padrão (por exemplo,

“12345”) é atribuída ao campo em análise.

Após todos os pontos de vulnerabilidades serem qualificados é realizada uma submissão da página para verificar se a página da resposta já passou pelo módulo de extração. Caso isso não tenha ocorrido, o endereço da página é adicionado na lista de newURLs para ser verificado. O processo de extração encerra quando a lista de newURLs está vazia.

4.6.4 Módulo de Teste

No ETSSDetector, o módulo de teste recebe como entrada o identificador da página inicial. De posse desse identificador é realizada uma consulta SQL na tabela webpage da base de dados para recuperar os parâmetros necessários para simular a página avaliada.

Os testes são conduzidos de acordo com o parâmetro cs_query_string que determina o tipo de requisição que será feita durante o teste (GET ou POST). Caso a requisição seja do tipo GET, um método para recuperar os parâmetros do endereço da página é chamado.

Basicamente esse método realiza uma consulta SQL para recuperar todos os parâmetros da tabela querystring da base de dados e faz uma requisição ao componente de injeção. Caso a requisição seja do tipo POST, um método para recuperar os parâmetros dos formulários da página é executado. Esse método recupera os parâmetros da tabela form da base de dados e faz uma requisição ao componente de injeção.

O componente de injeção aplica a técnica de teste de injeção para determinar a posição do ataque. A atual implementação do ETSSDetector usa o script alert(ID)
 para: i) determinar a posição do ponto de vulnerabilidade na página resposta e, ii) gerar os ataques apropriados a cada ponto de vulnerabilidade com o auxilio do componente de geração de código.

No componente de geração de código cada ataque é gerado com um identificador único. Esse ID é utilizado para auxiliar o módulo de detecção na procura dos ataques XSS que foram bem sucedidos. Os ataques são gerados de acordo com as categorias que estão listadas na Tabela 4.2 e serão explicadas a seguir:

Tabela 4.2:Categorias de casos de teste.

Categorias	Descrição
NonBracketTestCases	Utilizado caso o campo possua filtros
TitleTestCases	Utilizado caso o texto aparece no título da página
CSSTestCases	Utilizado quando o texto aparece dentro de um atributo CSS
HeaderTestCases	Utilizado quando o texto aparece no cabeçalho da página
TestCasesForComent	Utilizado quando o texto aparece em comentários
TestCasesInScriptTags	Utilizado quando o texto aparece em tags de script
TestCasesForTextArea	Utilizado quando o texto aparece em campos do tipo textarea
BracketTestCases	Utilizado quando o texto em outros campos e não possui filtro

- **NonBracketTestCases:** O ETSSDetector possui a capacidade de verificar se determinado ponto de vulnerabilidade possui filtros contra ataques XSS como trocas do caracter "<" por "lt". Esses filtros apresentam alguma eficiência contra ataques XSS, porém não impedem totalmente que um ataque aconteça. Esta categoria é responsável por gerar os ataques XSS que conseguem sobrepor esse tipo de filtro.
- **TitleTestCases:** Essa categoria gera ataques que serão utilizados nos títulos das páginas que estão sendo testadas. Os testes utilizam a *tag* <title> do HMTL.
- **CSSTestCases:** Categoria que gera ataques que serão utilizados apenas nos atributos CSS. Nesse caso os ataques são gerados com *tags* CSS como "background-image".
- **HeaderTestCases:** Categoria que gera os ataques que serão utilizados no cabeçalho da página que esta sendo testada.
- **TestCasesForComent:** Essa categoria gera os ataques que serão utilizados dentro de blocos de comentários.
- **TestCasesInScriptTags:** Categoria responsável por gerar os ataques que aparecem dentro de *tags* de script.
- **TestCasesForTextArea: :** Essa categoria gera ataques que serão utilizados apenas em campos do tipo *textarea*, utiliza a *tag* <textarea> do HMTL.
- **BracketTestCases:** Essa categoria é utilizada quando o ponto de vulnerabilidade é de um tipo que não foi especificado anteriormente e quando a aplicação web não possui nenhum tipo de filtro.

Com o código malicioso inserido no ponto de vulnerabilidade ocorre uma submissão da página para verificar se o ataque será executado, essa ação finaliza o módulo de teste.

4.6.5 Módulo de Análise

O módulo de análise se inicia após a página que possui o ataque ser submetida, com isso o componente de detecção utiliza um recurso do HTMLUNIT chamado AlertHandler para coletar todos os alertas causados pelos ataques testados naquela página.

Após ocorrer uma iteração na lista de alertas coletada pelo *AlertHandler* e caso o valor do código gerado seja encontrado, um ataque XSS foi executado. Com isso o componente atualiza os valores nas tabelas input (caso o ponto de vulnerabilidade localize-se em um formulário) ou querystring (caso o ponto de vulnerabilidade localize-se no endereço da página) da base de dados inserindo o ataque correspondente. Esse procedimento continua até que todos os pontos de vulnerabilidade da página sejam checados. A Figura 4.9 mostra o ETSSDetector no final da sua execução.

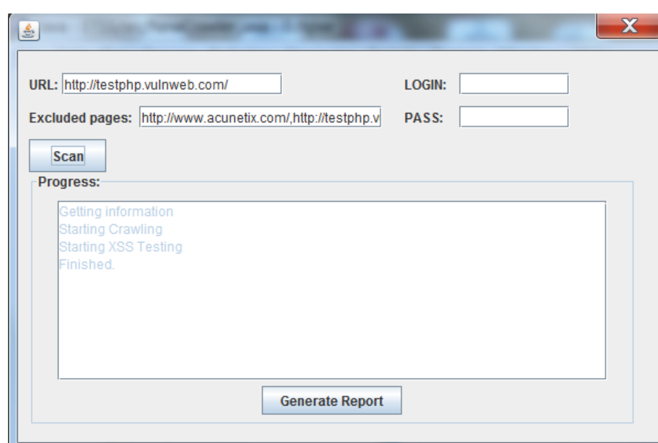

Figura 4.9: Progresso do ETSSDetector.

4.6.6 Módulo de Relatórios

O componente de relatório pode ser utilizado para gerar os relatórios a partir das informações das páginas e resultados da análise salvas na base de dados. Os relatórios são gerados através de uma consulta SQL nas tabelas webpage, form, input e querystring. A Figura 4.10 mostra um exemplo de relatório.

url	method	field	attack
http://localhost/XSS/get.php?name=12345	GET	name	>'"><script>alert(2)</script>
http://localhost/XSS/submit.php	get	name	>'"><script>alert(1)</script>

Figura 4.10: Exemplo de relatório do ETSSDetector.

Atualmente as informações mostradas são a página afetada, o método de execução do

formulário, o ponto de vulnerabilidade e qual ataque foi efetivo. O ataque pode ser facilmente reproduzido por um profissional de segurança através dos seguintes passos:

- Acessar a aplicação.
- Adicionar o código malicioso no ponto de vulnerabilidade.
- Submeter a requisição.
- Navegar até a página vulnerável e verificar o alerta.

De posse dessas informações profissionais de segurança podem identificar quais campos devem ser corrigidos na aplicação e como podem ser corrigidos.

4.6.7 Base de Dados

O ETSSDetector mantém todas as informações coletadas durantes as etapas de processamento de uma página *web* em uma base de dados. O uso de um repositório persistente possibilita a ferramenta acesso aos recursos em análise (*links*, formulários, parâmetros, e outros) em qualquer instante de tempo.

Por razões de simplicidade, o projeto do ETSSDetector usa o servidor de banco de dados MySQL. O MySQL é um sistema de gerenciamento de banco de dados de código-fonte aberto popular geralmente utilizado em aplicações *web* devido à sua velocidade, flexibilidade e confiabilidade. O MySQL emprega a linguagem SQL, ou *Structured Query Language*, para acessar e processar os dados contidos em bancos de dados.

A base de dados desenvolvida possui cinco tabelas que serão detalhadas a seguir. Para facilitar o entendimento, uma padronização nos nomes dos campos foi adotada. Os campos do tipo chave primária começam com o valor "id", os campos do tipo *string* começam com "tx", os campos do tipo *boolean* começam com "cs" e os campos do tipo inteiro com "nb".

Tabela 4.3: Tabela webpage.

Campo	Descrição
id_webpage	Campo que serve como identificador único da tabela webpage
tx_url	String responsável por armazenar a URL da página coletada
tx_title	String responsável por armazenar o título da página coletada
cs_query_string	Booleano que irá informar se a URL possui query string ou não
nb_start_url	Inteiro que armazena o id da página onde os testes foram iniciados

Tabela 4.4: Tabela form.

Campo	Descrição
id_form	Campo que serve como identificador único da tabela form
tx_method	String responsável por armazenar o valor da tag method do form
tx_action	String responsável por armazenar o valor da tag action do form
webpage_id_webpage	Chave estrangeira que serve de ligação entre a tabela webpage e a tabela form
tx_name	String para salvar o atributo name do form

Tabela 4.5: Tabela input.

Campo	Descrição
id_input	Campo que serve como identificador único da tabela input
tx_name	String responsável por armazenar o valor da tag name do campo
tx_type	String responsável por armazenar o valor da tag type do campo
cs_vulnerable	Booleano que identificará se o campo é vulnerável ou não para a geração dos relatórios
tx_attack	String responsável por armazenar o ataque que foi executado na aplicação
form_id_form	Chave estrangeira que serve de ligação entre a tabela form e a tabela vulnerable points
tx_attacked_page	String responsável por armazenar a página que executou o ataque
nb_search_id	Id da pesquisa para auxiliar nos relatórios
tx_value	String para salvar o valor inserido no input após a qualificação
tx_id	String para salvar o parâmetro id do input

Tabela 4.6: Tabela querystring.

Campo	Descrição
id_query_string	Campo que serve como identificador único da tabela querystring
tx_param	String responsável por armazenar o nome dos parâmetros utilizados na URL
tx_attack	Campo responsável por armazenar qual ataque foi executado
webpage_id_webpage	Campo responsável por armazenar o id da página que possui esses parâmetros

Tabela 4.7: Tabela qualifier.

Campo	Descrição
id_qualifier	Campo que serve como identificador único da tabela ataque
tx_name	String responsável por armazenar o nome do qualificador
tx_value	String responsável por armazenar o valor do qualificador

4.7 Considerações Finais.

Através da avaliação da metodologia é possível observar que a mesma possui diversas características que não existem na maioria das pesquisas anteriores. Tais características foram adicionadas no intuito de mitigar os problemas encontrados por Bau [12], Kosuga [14] e

Mcallister [15].

O próximo capítulo apresenta uma avaliação de desempenho do protótipo desenvolvido (ETSSDetector) através de uma análise comparativa em diversos cenários de testes com outras ferramentas disponíveis no mercado.

Capítulo 5

Resultados

Este Capítulo apresenta os resultados obtidos através da avaliação de desempenho do protótipo desenvolvido para a metodologia, o ETSSDetector. Para validar e comparar o protótipo desenvolvido foram realizados vários testes sobre cenários experimentais e cenários reais. Os resultados são apresentados na forma de tabelas e discussões. No intuito de enriquecer o estudo foram escolhidas e analisadas outras ferramentas de testes de vulnerabilidade para análise de características e comparação de resultados.

5.1. Protocolo Experimental

Para avaliar a metodologia proposta foram realizados vários experimentos e seus resultados comparados com as ferramentas Acunetix [31], PowerFuzzer [57], WebSecurify [58], NStalker [50], WebXSSDetector [24] e WebCruiser [59]. Além dessas, houve a tentativa de utilizar outras ferramentas, como o APPScan, porém impedimentos diversos impossibilitaram a sua utilização, no caso do APPScan o mesmo não possui uma versão *free*.

Os critérios para escolha foram primeiramente a disponibilidade para download da ferramenta de testes de vulnerabilidade (*scanners*), a similaridade de objetivos e a percepção de utilização da ferramenta no mercado. No conjunto das ferramentas o Acunetix é considerado um dos melhores *scanners* de vulnerabilidades do mercado conforme os estudos citados anteriormente neste trabalho. Com isso, é possível concluir que a escolha das ferramentas é adequada para atingir os objetivos de avaliação do protótipo da metodologia proposta. O único problema relacionado ao Acunetix é que a versão gratuita da ferramenta só pode ser utilizada nas páginas desenvolvidas pela empresa que possui a licença do produto. Tal fato possibilitou o uso do Acunetix apenas no primeiro cenário de teste, onde uma aplicação web desenvolvida pela empresa que possui o Acunetix foi utilizada.

As métricas utilizadas para avaliar as ferramentas nos experimentos foram:

- Quantidade de ataques XSS detectados.

- Quantidade de testes gerados na aplicação.
- Tempo de execução das ferramentas.

Tais métricas foram escolhidas por serem utilizadas em pesquisas anteriores e por verificarem a resolução dos problemas encontrados na motivação.

5.2. Ambiente de Experimentação

As ferramentas foram executadas em uma máquina com o sistema operacional Windows 7 que possui 6 *gigabytes* de memória e um processador Intel Core i7 de 1.73 *GigaHertz*. Por razões de compatibilidade com o sistema operacional Windows 7, o N-Stalker foi executado em uma máquina com configuração inferior que possui sistema operacional Windows Vista, 2 *gigabytes* de memória e processador Intel Core 2 duo de 1.73 *GigaHertz*. O N-Stalker foi utilizado apenas no primeiro cenário de testes devido a diferença de *hardware* e sistema operacional.

5.3. Cenário de Testes

Todas as ferramentas são executadas sobre os mesmos cenários de testes no intuito de avaliar o funcionamento de cada *scanner* e realizar análises comparativas através da coleta de informações sobre o funcionamento de cada ferramenta. A coleta ocorre através da análise dos comportamentos das ferramentas, dos relatórios gerados e, quando possível, dos *logs* obtidos no servidor onde se encontra a aplicação que esta sendo testada.

Para a criação dos cenários de testes foram utilizadas aplicações vulneráveis conhecidas (primeiro e terceiro cenário) e foi desenvolvida uma aplicação (segundo cenário), utilizando a linguagem PHP, onde vulnerabilidades específicas foram adicionadas de forma proposital para realizar as comparações.

5.3.1. Primeiro Cenário

No primeiro cenário de teste foi utilizada uma aplicação desenvolvida pelos proprietários da ferramenta Acunetix. A aplicação encontra-se no endereço *testphp.vulnweb.com,* como mostrado na Figura 5.1.

Trata-se de uma aplicação desenvolvida com diversas vulnerabilidades para auxiliar os pesquisadores na obtenção de conhecimento sobre as mesmas. Cada ferramenta foi executada apenas uma vez na aplicação, essa decisão foi adotada pelo fato de alguns fatores

influenciarem no tempo de execução das ferramentas. Como, por exemplo, a velocidade da conexão com a Internet, a Tabela 5.1 mostra os resultados.

Figura 5.1: Tela inicial da aplicação de teste do Acunetix.

O primeiro ponto a ser analisado neste cenário de teste é que a maioria das ferramentas tiveram o seu tempo de execução entre 2 e 5 minutos, com exceção da ferramenta PowerFuzzer que terminou a sua execução em 14 minutos e 51 segundos e o WebSecurify que foi executado em apenas 37 segundos.

Tabela 5.1: Resultados do primeiro cenário de teste.

Ferramenta	Ataques XSS encontrados	Tempo
Acunetix	17 ataques	2 minutos e 2 segundos
PowerFuzzer	12 ataques	14 minutos e 51 segundos
N-Stalker	9 ataques	5 minutos
WebSecurify	4 ataques	37 segundos
WebXSSDetector	15 ataques	4 minutos e 9 segundos
WebCruiser	8 ataques	2 minutos e 30 segundos
ETSSDetector	17 ataques	2 minutos e 37 segundos

A razão da ferramenta PowerFuzzer ter sido executada em 14 minutos e 51 segundos é que a mesma possui vários tipo de ataques (não somente XSS) e efetua todos esses ataques em cada ponto de vulnerabilidade, fazendo com que o seu tempo de execução seja

diretamente associado aos tipos e quantidade de ataques testado e pontos de vulnerabilidade da aplicação. Por outro lado, não foi possível determinar a razão da ferramenta WebSecurify ter sido executado em apenas 37 segundos. A ferramenta não fornece acesso aos *logs* do servidor onde a aplicação de teste está instalada. Porém, pelo fato da ferramenta ter encontrado apenas 4 ataques XSS pode-se concluir que o grupo de testes contra ataques XSS que a ferramenta possui deve ser bastante limitado.

Com relação aos ataques de XSS, o Acunetix e o ETSSDetector detectaram o maior número de vulnerabilidades, 17 ataques cada, sendo 15 deles os mesmos. O Acunetix encontrou dois ataques XSS que foram executados através de *cookies,* funcionalidade que o ETSSDetector ainda não possui, impossibilitando a sua detecção. Contudo, o ETSSDetector encontrou dois outros ataques não detectados pelo Acunetix. As únicas ferramentas capazes de encontrar esses dois ataques foram o ETSSDetector e o WebXSSDetector.

Esses ataques ocorrem na página da aplicação que serve para o usuário criar contas http://testphp.vulnweb.com/signup.php, em um dos ataques possíveis nessa página o atacante pode utilizar os campos "*password*" e "*retype password*" para inserir *scripts* maliciosos. O *script* inserido deve ser o mesmo nos dois campos, como ambos são vulneráveis a ferramenta identificou como pontos de vulnerabilidade. Para provar que esses ataques não são falsos positivos os mesmos foram reproduzidos manualmente seguindo os seguintes passos:

1. Acessar a página http://testphp.vulnweb.com/signup.php.
2. Preencher os campos "*password*" e "*retype password*" com um ataque XSS, por exemplo, >"'><script>alert('XSS')</script>.
3. Preencher os outros campos.
4. Clicar no botão *signup*.

Após esses passos o ataque pode ser verificado como mostra a Figura 5.2, o navegador utilizado foi o Mozilla Firefox na versão 18.0.

Para finalizar a análise de resultados no primeiro cenário de teste as ferramentas PowerFuzzer e WebCruiser detectaram uma vulnerabilidade XSS na página da aplicação http://testphp.vulnweb.com/showimage.php, que mostra imagens. Essa vulnerabilidade foi testada manualmente e não foi possível reproduzi-la, de posse dessa informação e pelo fato de nenhuma das outras 5 ferramentas terem detectado esse ataque, o mesmo foi considerado um

falso positivo.

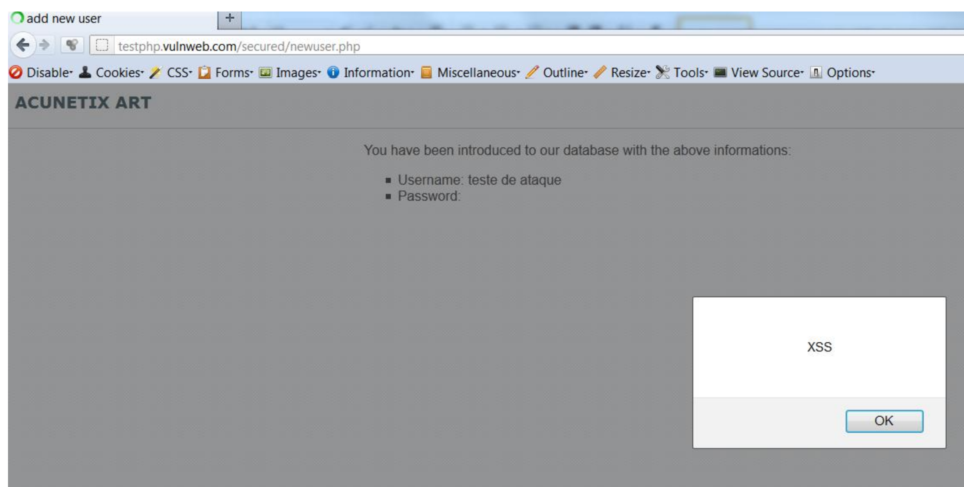

Figura 5.2: Ataque XSS na página “criar usuários”.

5.3.2. Segundo Cenário

No segundo cenário de teste foi desenvolvida uma página simples para testar o processo de qualificação e a geração de ataques. Essa página possui quatro campos como mostra a Figura 5.3.

nome:
sobrenome:
Email:
CPF:
Submit

Figura 5.3: Campos da aplicação para teste do processo de qualificação.

Entre esses campos dois possuem vulnerabilidades (nome e sobrenome) e dois precisam de validação para o formulário ser submetido (Email e CPF). Propositalmente o campo nome é exibido no título da página de resposta e o campo sobrenome é exibido como atributo de uma *tag* HTML. Esse procedimento foi feito no intuito de testar se o ETSSDetector gera menos ataques (menos requisições) para cada ponto de vulnerabilidade através da aplicação da técnica de testes de injeção.

Como explicado anteriormente o Acunetix, na versão *free*, não executa em outras aplicações que não sejam as páginas fornecidas pela empresa que o desenvolveu, enquanto que o N-Stalker não possui uma interface funcional para ser utilizada no Windows 7. Assim para este cenário de teste e os seguintes serão utilizadas apenas as ferramentas PowerFuzzer,

ETSSDetector, WebSecurify, WebCruiser e o WebXSSDetector. A Tabela 5.2 exibe os resultados do segundo cenário.

Tabela 5.2: Resultados do segundo cenário de teste.

Ferramenta	Quantidade de requisições	Quantidade de ataques detectados
PowerFuzzer	64 requisições	Nenhum
WebSecurify	37 requisições	Todos
WebXSSDetector	Erro em tempo de execução	Nenhum
WebCruiser	60 requisições	Todos
ETSSDetector	15 requisições	Todos

Através da Tabela 5.2 é possível verificar que duas das cinco ferramentas não foram capazes de detectar os ataques XSS. O WebXSSDetector apresentou um erro em tempo de execução no seu componente utilizado para extrair as páginas. Tal fato impossibilitou a continuação do processo de escaneamento da ferramenta.

O PowerFuzzer foi a ferramenta que efetuou o maior número de requisições, 64 no total, porém passou valores inválidos em todas elas fazendo com que os ataques XSS não fossem detectados, como mostra a Figura 5.4.

```
http://localhost/XSS/submitpost.php/post.php
Attacking forms (POST)...
+ http://localhost/XSS/submitpost.php/post.php
'sobrenome' => 'on'
'Email' => 'http://www.google.com/'
'CPF' => 'on'
'name' => 'on'
```

Figura 5.4: Exemplo de ataque do PowerFuzzer.

Analisando a Figura 5.4 é possível verificar que o PowerFuzzer passa como parâmetro para o campo de *email* o endereço do Google e passa como parâmetro para o campo CPF o texto "*on*" fazendo com quem o formulário não fosse submetido.

Entre as ferramentas que detectaram os ataques, o WebCruiser gerou mais de quatro vezes a quantidade de requisições geradas pelo ETSSDetector e o WebSecurify gerou mais que o dobro. Com isso, pode-se afirmar que o ETSSDetector gerou menos ataques em cada ponto de vulnerabilidade através da qualificação dos campos de entrada durante o processo de teste de injeção e detectou todos os ataques.

5.3.3. Terceiro Cenário

No terceiro cenário foram conduzidos testes em uma aplicação que efetua reserva de salas, disponibilizada em http://www.webscantest.com/crosstraining/reservation.php. Essa página possui duas vulnerabilidades XSS, ambas não persistentes. Porém, uma delas se comporta como se fosse persistente, pois não é imediatamente refletida ao usuário, a tabela 5.3 mostra os resultados.

Tabela 5.3: Resultados do terceiro cenário de teste.

Ferramenta	Quantidade de ataques detectados
PowerFuzzer	Nenhum
WebSecurify	Nenhum
WebXSSDetector	Nenhum
WebCruiser	Um
ETSSDetector	Todos

As ferramentas WebCruiser, PowerFuzzer, ETSSDetector, WebSecurify e WebXSSDetector foram utilizadas nesta aplicação e apenas o ETSSDetector foi capaz de detectar as duas vulnerabilidades. O WebCruiser detectou apenas uma das vulnerabilidades e as demais ferramentas não encontraram nenhuma. No intuito de verificar se o ataque que apenas o ETSSDetector encontrou tratava-se de um falso positivo o mesmo foi reproduzido manualmente através dos seguintes passos:

1. Acessar a página www.webscantest.com/crosstraining/reservation.php.
2. Preencher o campo “*First Name*” com um ataque de XSS, por exemplo, “>"'><script>alert(‘XSS’)</script> “.
3. Preencher os outros campos.
4. Clicar no botão *submit*.
5. Clicar no link “*start new reservation*” na página de confirmação da reserva.
6. Clicar no link “*view reservation history*” na página de reservas.

Após esses passos serem feitos o ataque será executado como mostra a Figura 5.5.

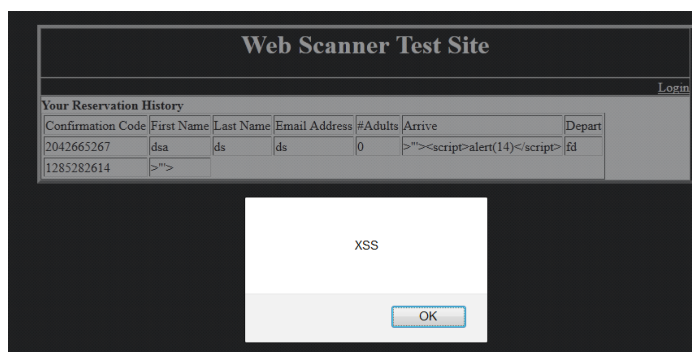

Figura 5.5: Ataque XSS detectado pelo ETSSDetector.

Os testes que são realizados pelo ETSSDetector não são baseados apenas na resposta direta gerada pela requisição. Uma análise é realizada nas outras páginas da aplicação, por esse motivo foi possível detectar os dois ataques que a aplicação de reserva de salas possui.

5.4. Considerações Finais

Através da análise dos resultados obtidos no capítulo é possível verificar que a aplicação da técnica de testes de injeção reduziu a quantidade de testes gerados nos pontos de vulnerabilidades, e a aplicação da qualificação nos pontos de vulnerabilidade proporcionou o aumento da garantia de submissão de formulários complexos.

Por fim, a verificação dos testes em todas as páginas da aplicação, em conjunto com as técnicas de testes de injeção e qualificação fez com que o nível de detecção dos ataques XSS fosse aumentado.

Capítulo 6

Conclusões & Trabalhos Futuros

Este trabalho apresentou uma metodologia para detecção automática de ataques de *Cross-Site Scripting* em páginas *web* através da análise das informações contidas nas aplicações.

Vários trabalhos têm sido propostos para mitigar os ataques XSS, porém, grande parte dos trabalhos encontrados na literatura que utilizam a técnica de testes caixa-preta englobam vários tipos de vulnerabilidades e não apresentam resultados satisfatórios. Para obter resultados melhores este trabalho focou apenas na detecção de ataques XSS persistentes e não persistentes, explorando detalhadamente o problema.

A metodologia apresentada foi dividida em três fases. A primeira fase, chamada de fase de extração e preparação, foi criada para realizar a extração do conteúdo das páginas, identificando os pontos de vulnerabilidade, e para aplicar a qualificação dos pontos preenchendo os mesmos com valores válidos para garantir a submissão dos formulários.

A segunda fase, chamada de fase de testes, foi criada para aplicar a técnica de testes de injeção no intuito de gerar apenas os ataques necessários em cada ponto de vulnerabilidade da aplicação.

A fase de análise foi concebida no intuito de realizar a análise dos resultados detectando os ataques XSS persistentes e não persistentes e para a emissão dos relatórios. Todas as fases tiveram suporte de uma base de dados para o armazenamento das informações.

Para avaliar a viabilidade da metodologia foi desenvolvido um protótipo chamado ETSSDetector que foi avaliado com outras ferramentas em diferentes cenários de teste. Na criação dos cenários foram utilizadas aplicações vulneráveis já existentes e também foram criadas aplicações vulneráveis novas.

Os resultados dos experimentos mostraram que a utilização de um protótipo que utilizou a metodologia conduziu a uma melhora na detecção de ataques XSS e redução na

quantidade de ataques gerados nos pontos de vulnerabilidades diminuindo o tempo de execução do protótipo.

Conclui-se que os resultados alcançados são promissores e demonstram a viabilidade do uso da metodologia para minimizar os problemas que foram expostos nos capítulos iniciais e o protótipo desenvolvido apresenta-se como uma alternativa para auxiliar os profissionais de segurança da informação na luta contra as vulnerabilidades que estão presentes em sistemas *web*.

6.1 Trabalhos Futuros

As pesquisas na área de detecção vulnerabilidades são dinâmicas, pois as técnicas de ataques estão em constante evolução e mudanças. Portanto as ferramentas de testes caixa-preta devem prover ao usuário uma forma de atualizar o seu componente de testes para estar sempre adicionando as novas formas de ataque.

Diante desse fato, como trabalho futuro pretende-se alterar o componente de geração de códigos do protótipo, para possibilitar que ataques sejam adicionados pelos usuários. Outra opção de trabalho futuro seria a avaliação do protótipo para adicionar mecanismos para detectar ataques XSS do tipo DOM.

Por fim, ampliar a utilização da metodologia proposta adicionando mecanismos para a detecção de outras vulnerabilidades em aplicações *web* como SQL *Injection*.

Referências

[1] D. Shelly, *Using a Web Server Test Bed to Analyze the Limitations of Web,* 2010.

[2] OWASP, “OWASP,” OWASP, [Online]. Available: http://owasp.com/index.php/Main_Page. [Acesso em 15 Julho 2011].

[3] OWASP, “OWASP TOP TEN WEB APPLICATION SECURITY RISKS,” Janeiro 2013. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[4] Zhendong Su e Gary Wassermann, “The Essence of Command Injection Attacks in Web Applications,” *33rd ACM Symposium on Principles of Programming Languages,* 2006.

[5] Jeremiah Grossman, Robert Hansen, Petko Petkov, Anton Rager e Seth Fogie, Cross site scripting attacks:XSS Exploits and defense., Syngress,Elsevier, 2007.

[6] IPA, “IPA/ISEC: IT Security,” Setembro 2012. [Online]. Available: http://www.ipa.go.jp/security/english/index.html. [Acesso em 18 Fevereiro 2013].

[7] Steve Christey e Robert A. Martin, “CWE - Vulnerability Type Distributions in CVE,” CWE, 22 Maio 2007. [Online]. Available: http://cwe.mitre.org/documents/vuln-trends/index.html#table1. [Acesso em 10 Fevereiro 2013].

[8] WebAppSec, “The Web Application Security Consortium,” WebAppSec, 2007. [Online]. Available: http://projects.webappsec.org/w/page/13246989/Web-Application-Security-Statistics. [Acesso em 18 Fevereiro 2013].

[9] WhiteHat, “WhiteHat WebSite Security Statistics Report 2011,” WhiteHat, 2011. [Online]. Available: https://www.whitehatsec.com/resource/stats.html. [Acesso em 20 Fevereiro 2013].

[10] B. Lord, “All about the onMouseOver incident,” Setembro 2010. [Online]. Available: http://blog.twitter.com/2010/09/all-about-onmouseover-incident.html. [Acesso em 23 Fevereiro 2013].

[11] J. Jean, “Facebook CSRF and XSS vulnerabilities | Destructive worms on a social network,” Outubro 2010. [Online]. Available: http://seclists.org/fulldisclosure/2010/Oct/35. [Acesso em 20 Janeiro 2013].

[12] Jason Bau e Elie Bursztein e Divij Gupta e John Mitchell, “State of the Art: Automated Black-Box Web Application,” *IEEE Symposium on Security and Privacy Vulnerability Testing,* 2010.

[13] Adam Doupe e Marco Cova e Giovanni Vigna, “Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners,” *Seventh Conference on Detection of Intrusions and Malware and Vulnerability Assessment,* 2010.

[14] Y. Kosuga, *A Study on Dynamic Detection of Web Application Vulnerabilities,* 2011.

[15] Sean McAllister e Engin Kirda e Christopher Kruegel, “Leveraging User Interactions for In-Depth Testing of Web Applications,” *ACM international symposium on Recent Advances in Intrusion Detection,* Setembro 2008.

[16] W3C, “Web Architecture - W3C,” W3C, [Online]. Available: http://www.w3.org/standards/webarch/. [Acesso em 10 Janeiro 2013].

[17] Alan Grosskurth e Michael W. Godfrey, “Architecture and evolution of the modern web

browser," *IEEE international conference on software maintenance,* 19 Junho 2006.

[18] W3C, "W3C Document Object Model," W3C, [Online]. Available: http://www.w3.org/DOM/. [Acesso em Fevereiro 15 2013].

[19] W. Soares, AJAX guia prático, São Paulo: Érica, 2007.

[20] RedMonk, "Javascript Leads the Pack as Most Popular Programming Language," RedMonk, Setembro 2012. [Online]. Available: http://www.webpronews.com/javascript-leads-the-pack-as-most-popular-programming-language-2012-09. [Acesso em 10 Novembro 2012].

[21] Chuan Yue e Hong Wang, "Charatering Insecure JavaScript Practice on the Web," *International Conference on the World Wide Web,* 2005.

[22] W3C, "HTTP Overview," W3C, [Online]. Available: http://www.w3.org/Protocols/. [Acesso em 3 Novembro 2012].

[23] K. Laube, "Entendendo os cookies e sessões," [Online]. Available: http://klauslaube.com.br/2012/04/05/entendendo-os-cookies-e-sessoes/. [Acesso em 23 Novembro 2011].

[24] X. Jia, *Design, Implementation and Evaluation of an Automated Testing Tool for Cross-Site Scripting Vulnerabilities,* 2006.

[25] Jin-Cherng Lin e Jan-Min Chen e Cheng-Hsiung Liu, "An Automatic Mechanism for Sanitizing Malicious Injection," *International Conference for Young Computer Scientists,* 2009.

[26] S. Saha, "Consideration Points: Detecting Cross-Site Scripting," *International Journal of Computer Science and Information Security,* 2009.

[27] Engin Kirda e Christopher Kruegel e Giovanni Vigna e Nenad Jovanovic, "Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks," *ACM symposium on Applied computing,* 2006.

[28] S. Selvan, "000Webhost vulnerability," Cyber Security Researcher Vedachala, 30 Janeiro 2013. [Online]. Available: http://www.ehackingnews.com/2013/01/000webhost-vulnerable-to-non-persistent.html. [Acesso em 10 Fevereiro 2013].

[29] Tadeusz Pietraszek e Chris Vanden Berghe, "Defending against Injection Attacks through Context-Sensitive String Evaluation," *Proceedings of Recent Advances in Intrusion Detection,* 2005.

[30] C. Wueest, "Persistent XSS vulnerability in Facebook," Symantec, 9 Março 2011. [Online]. Available: http://www.symantec.com/connect/blogs/persistent-xss-vulnerability-facebook. [Acesso em 15 Fevereiro 2013].

[31] Acunetix, "Website Security with Acunetix Web Vulnerability Scanner," Acunetix, [Online]. Available: http://www.acunetix.com/. [Acesso em 10 Agosto 2011].

[32] A. Klein, "DOM Based Cross Site Scripting or XSS of the Third Kind," Web Application Security Consortium, 7 Abril 2005. [Online]. Available: http://www.webappsec.org/projects/articles/071105.shtml. [Acesso em 20 Outubro 2012].

[33] S. Ramezany, "Yahoo DOM XSS," 9 Janeiro 2013. [Online]. Available: http://www.offensive-security.com/offsec/yahoo-dom-xss-0day-prevails/. [Acesso em 20 Fevereiro 2013].

[34] Youtube, "Youtube," Youtube, 9 Janeiro 2013. [Online]. Available: http://www.youtube.com/. [Acesso em 5 Fevereiro 2013].

[35] Jayamsakthi Shanmugam M. Ponnavaikko, "Cross Site Scripting-Latest developments and solutions: A survey," *Int. J. Open Problems Compt. Math.,* 2008.

[36] K. Selvamani e A.Duraisamy e A.Kannan, "Protection of Web Applications from Cross-Site Scripting Attacks in Browser Side," *(IJCSIS) International Journal of Computer Science and Information Security,* 2010.

[37] L. Muller, "O que é Phishing?," [Online]. Available: http://www.tecmundo.com.br/phishing/205-o-que-e-phishing-.htm. [Acesso em 15 Abril 2011].

[38] Symantec, "Symantec," [Online]. Available: http://www.symantec.com/region/br/avcenter/education/index.html. [Acesso em 19 Julho 2011].

[39] S. Shalini e S. Usha, "Prevention Of Cross-Site Scripting Attacks (XSS) On Web Applications In The Client Side," *IJCSI International Journal of Computer Science Issues,* 2011.

[40] J. Martin and E. Bjorn and P. Joachim, "XSSDS: Server-side Detection of Cross-Site Scripting Attacks," *IEEE Computer Security Applications Conference,* 2008 Outubro.

[41] Prithv Bisht e V.N. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," *ACM Symposium on Applied Computing,* Maio 2006.

[42] Mike Ter Louw e V.N. Venkatakrishnan, "BLUEPRINT: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers," *In: Proceedings of the IEEE Symposium on Security and Privacy,* 2009.

[43] Nenad Jovanovic e Christopher Kruegel e Engin Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," *IEEE Symposium on Security and Privacy,* 2006.

[44] T. Jim e N. Swamy e M. Hick, "Defeating script injection attacks with browser-enforced embedded policies," *International World Wide Web Conference (WWW2007),* Maio 2007.

[45] O. Ismail e M. Eto e Y. Kadobayashi e S. Yamaguchi, "A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability," *International Conference on Advanced Information Networking and Applications (AINA04),* Março 2004.

[46] Elias Athanasopoulos e Vasilis Pappas e Antonis Krithinakis e Spyros Ligouras e Evangelos P. Markatos, "xJS: Practical XSS Prevention for Web Application Development," *In: Proceedings of the 2010 USENIX Conference on Web Application Development,* 2010.

[47] Elizabeth Fong e Romain Gaucher e Vadim Okun e Paul E. Black, "Building a Test Suite for Web Application Scanners," *Hawaii International Conference on System Sciences,* 2008.

[48] Mateus Ferreira e Eduardo Feitosa e Thiago Rocha e Gilbert Martins e Eduardo Souto, "Análise de vulnerabilidades em Sistemas Computacionais Modernos: Conceitos, Exploits e Proteções," *SBSEG,* 19 Novembro 2012.

[49] AppScan, "Download IBM AppScan," IBM, [Online]. Available:

http://www.ibm.com/developerworks/downloads/r/appscan/. [Acesso em 20 Junho 2012].

[50] N-Stalker, "N-Stalker The Web Security Specialists," N-Stalker, [Online]. Available: http://www.nstalker.com/. [Acesso em 14 Abril 2012].

[51] WebCruiser, "Download WebCruiser," WebCruiser, [Online]. Available: http://sec4app.com/download.htm. [Acesso em 1 Fevereiro 2012].

[52] Stefan Kals e Engin Kirda e Christopher Kruegel e Nenad Jovanovic, "SecuBat: A Web Vulnerability Scanner," *International World Wide Web Conference (WWW2006),* Maio 2006.

[53] PowerFuzzer, "PowerFuzzer - a fuzzer that introduces powerfull and easy web fuzzing," PowerFuzzer, [Online]. Available: http://www.powerfuzzer.com/. [Acesso em 20 Julho 2012].

[54] WebSecurify, "WebSecurify Online Web Application Security Scanner and Web Security Testing Tool," WebSecurify, [Online]. Available: http://www.websecurify.com/. [Acesso em 5 Julho 2012].

[55] OWASP, "XSS Filter Evasion Cheat Sheet," OWASP, [Online]. Available: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. [Acesso em 11 Outubro 2011].

[56] Gargoyle, "HtmlUnit - Welcome to HtmlUnit," Gargoyle, [Online]. Available: http://htmlunit.sourceforge.net/. [Acesso em 10 Janeiro 2012].

[57] HACKTOOLREPOSITORY, "Hack Tool Repository - Download Powerfuzzer," HACKTOOLREPOSITORY, [Online]. Available: http://www.hacktoolrepository.com/tool/126/Powerfuzzer. [Acesso em 4 Fevereiro 2013].

[58] WEBSECURIFY, "Web Application Security Scanner and Manual," WEBSECURIFY, [Online]. Available: http://www.websecurify.com. [Acesso em 10 Janeiro 2013].

[59] SEC4APP, "Web Vulnerability Scanner, SQL Injection Tool," SEC4APP, [Online]. Available: http://sec4app.com. [Acesso em 10 Fevereiro 2013].