Aprimoramento do módulo *crawler* em *scanners* de vulnerabilidades (*open source*) de aplicações web

Danilo Pereira Escudero

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Mestrado em Ciência da Computação
Orientador: Prof. Dr. Routo Terada

São Paulo, Abril de 2022

Aprimoramento do módulo *crawler* em *scanners* de vulnerabilidades (*open source*) de aplicações web

Esta é a versão original da dissertação elaborada pelo
candidato Danilo Pereira Escudero, tal como
submetida à Comissão Julgadora.

Agradecimentos

Ao professor Routo Terada, pela orientação, competência, paciência e dedicação. Minha admiração por ele ultrapassa o âmbito acadêmico, pois o considero um ser humano exemplar.

À professora Cíntia Borges Margi e ao professor Daniel Macedo Batista, pelo aprendizado em suas disciplinas das quais fui discente, pela participação em minha banca de qualificação e pelas contribuições muito relevantes para que este trabalho fosse concluído.

Ao Instituto Federal de Educação, Ciência e Tecnologia de Rondônia - IFRO, pelo incentivo à qualificação por meio de concessão do afastamento de minhas atividades no *Campus* para cursar as disciplinas e realizar a qualificação desta dissertação.

À Katia Kiesshau, agradeço pelo ótimo atendimento aos pós-graduandos, sempre com presteza e cordialidade.

Ao Thales Paiva, por suas contribuições em nossas reuniões do grupo de orientandos do professor Routo.

Ao meu amigo hondurenho Luis, que contribuiu para tornar os momentos no IME mais descontraídos.

Ao Rogério Theodoro Brito (*in memoriam*), pelas discussões sobre computação e por ter me ajudado em alguns momentos de dificuldade acadêmica.

Ao Aarão, por suas contribuições durante a disciplina de "Grafinhos".

Ao meu irmão Daniel, por sempre me incentivar a estudar.

Ao casal de amigos Rooger e Érica, agradeço pela resolução de vários problemas que não eram seus. Pela amizade, dedicação, atenção, preocupação e cuidados, por proporcionarem alegria e descontração em momentos emocionalmente difíceis para minha esposa e eu.

Ao meu amigo Luiz Henrique Morais Aguiar, pelas discussões sobre algoritmos, pela leveza, bom humor, camaradagem e ajuda para encontrar erros de código fonte.

Ao meu amigo Ewerton Andrade, por abrir meus olhos para a possibilidade de cursar o mestrado no IME e pelo incentivo durante o processo de admissão. Durante meu momento de luto, pela ajuda na escrita do artigo publicado durante esta pesquisa, sem suas contribuições eu não conseguiria terminá-lo antes do prazo de submissão.

À minha sogra Antonia (*in memoriam*), agradeço por ter cuidado de mim durante o tempo em que morei em São Paulo. Obrigado por ter sido uma mãe, ter me dado momentos de diversão e sempre me incentivado no mestrado.

À minha esposa Elisabete, pelo amor, paciência, carinho, incentivo e deboche; por somar alegria e abrandar os momentos de tristeza; por ser minha companheira de vida e minha inspiração.

Aos meus pais, Selma e Isaias, pelo dom da vida. Por todas as renúncias a meu favor. Pelo amor incondicional e por sempre acreditarem no meu potencial mais do que eu mesmo. Obrigado pelo constante apoio e incentivo.

Resumo

ESCUDERO, D. P. **Aprimoramento do módulo *crawler* em *scanners* de vulnerabilidades (*open source*) de aplicações web**. 2022. Dissertação - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Os *Scanners* de vulnerabilidades para aplicações web são ferramentas que auxiliam na detecção de vulnerabilidades de forma automatizada e dividem-se em três módulos: rastreador, atacante e analisador. Essas ferramentas realizam testes de segurança para encontrar vulnerabilidades de forma automatizada, sem informações privilegiadas do sistema em teste. A utilização de *scanners* de vulnerabilidade pode diminuir os custos com verificações de segurança e dar mais agilidade na execução de testes de segurança. No entanto, os *scanners open source* ignoram diversos tipos de vulnerabilidades e apresentam resultados com muitos falsos positivos, falsos negativos e, principalmente, baixa capacidade de rastreabilidade. Após fazer a análise, a ferramenta gera um relatório de segurança em alto nível que pode ser usado pela equipe de tecnologia da informação para auxiliar nas atualizações necessárias a fim de suprimir possíveis falhas de segurança. Este trabalho aprimora o módulo rastreador, melhorando sua capacidade de indexação de páginas web e a visualização correta e renderizada dessas páginas. O módulo rastreador é um dos principais limitadores da eficácia dos *scanners*: se a ferramenta não é capaz de acessar todas as funcionalidades de um sistema web, muitas páginas vulneráveis não serão testadas. Este estudo realiza também um comparativo do módulo rastreador de *scanners* de código aberto, propondo um novo módulo rastreador para atingir melhores resultados, batizado de "Roudan". O Roudan apresentou cobertura superior às demais ferramentas testadas, pois os experimentos realizados nesta pesquisa demonstram que, ao adotar o Roudan como *crawler* desses *scanners* testados, a quantidade de falsos negativos diminui significativamente, visto que a ferramenta é capaz de testar mais funcionalidades do sistema em análise. Além disso, a capacidade de realizar requisições autenticadas nas aplicações também foi superior às demais ferramentas.

Palavras-chave: *scanner* de vulnerabilidades, *crawler*, teste de segurança.

Abstract

ESCUDERO, D. P. **Enhancement of the crawler module in open source web application vulnerability scanners**. Dissertation - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Web Vulnerability Scanners are tools that help detect vulnerabilities in an automated way and are divided into three modules: crawler, attacker and analyzer. These tools perform security tests to find vulnerabilities in an automated way, without privileged information from the system under test. The use of vulnerability scanners can reduce the costs of security scans and speed up the execution of security tests. However, open source scanners ignore several types of vulnerabilities and present results with many false positives, false negatives and mainly low crawability. After doing the analysis, the tool generates a high-level security report that can be used by the information technology team to assist in the necessary updates in order to suppress possible security breaches. This work improves the crawler module, improving its ability to index web pages and the correct and rendered visualization of these pages. The crawler module is one of the main limitations on the effectiveness of scanners: if the tool is not able to access all the functionality of a web system, many vulnerable pages will not be tested. This study also makes a comparison of the open source scanners crawler module, proposing a new crawler module to achieve better results, called "Roudan". Roudan presented better coverage than the other tested tools, therefore, the experiments carried out in this research conjecture that, by adopting Roudan as a crawler for these tested scanners, the number of false negatives would significantly decrease, since the tool would be able to test more system functionalities under scan. In addition, the ability to perform authenticated requests in applications was also superior to other tools.

Keywords: vulnerability scanner, crawler, security test.

Sumário

Lista de Abreviaturas **xi**

Lista de Figuras **xiii**

Lista de Tabelas **xv**

1 Introdução **1**
1.1 Motivação 2
1.2 Objetivo 2
1.3 Contribuições 2
1.4 Organização do texto 3

2 Conceitos **5**
2.1 Teste de segurança 6
2.1.1 Técnicas de teste de segurança 7
2.1.2 *Frameworks* para testes de segurança 9
2.2 Vulnerabilidades mais frequentes 10
2.3 *Scanner* de vulnerabilidades 11
2.3.1 *Crawler* 12
2.4 Propriedades de segurança 12
2.5 Considerações 13

3 Roudan **15**

4 Estudo de caso **21**
4.1 Método e descrição dos testes 21
4.2 *Scanners* de vulnerabilidades testados 22
4.3 Sistemas Web testados 23
4.4 Métricas de desempenho adotadas 24
4.5 Resultados e discussão dos experimentos 25
4.5.1 WackoPicko 26
4.5.2 OWASP *Juice Shop* 29
4.6 Considerações 30

5 Trabalhos relacionados **31**
5.1 Considerações 33

6 Conclusões **35**

Referências Bibliográficas **37**

Lista de Abreviaturas

CVE	*Common Vulnerabilities and Exposures*
ERRC	*scola Regional de Redes de Computadores*
HTTP	*ypertext Transfer Protocol*
IPS	*Intrusion Prevention System*
NIST	*National Institute of Standards and Technology*
OWASP	*Open Web Application Security Project*
OWASP SAMM	*Open Software Assurance Maturity Model*
Pentest	Teste de intrusão *Penetration Test*
PTES	*Penetration Testing Execution Standard*
SDL	Ciclo de Vida do Desenvolvimento Seguro (*Security Development Lifecycle*)
SQL	*Standard Query Language*
XML	*eXtensible Markup Language*
XSS	*Cross Site Scripting*
WAF	*web application firewall*
WRSeg	*Workshop Regional de Segurança da Informação e de Sistemas Computacionais*

Lista de Figuras

2.1 Etapas necessárias para a execução de um teste de segurança. 6
2.2 Projetos de testes de segurança. 6
2.3 Fases de um teste de intrusão. Adaptado de: [SS08]. 8
2.4 Técnicas de Teste de Segurança no Ciclo de Vida de Desenvolvimento de software. Adaptado de: [FBJ+16] 8
2.5 Proporção do esforço de realização de testes em cada fase do ciclo de desenvolvimento de software. Adaptado de: [OWA20] 9
2.6 Módulos de um *scanner* de vulnerabilidades web. 11

3.1 Fluxograma de execução do Roudan. 19

4.1 Infraestrutura de testes. 22

Lista de Tabelas

3.1 Links e redirecionamentos HTML e JavaScript. 16
3.2 Versões da linguagem Python, das bibliotecas e do geckodriver utilizados. 17

4.1 Limitações de renderização de JavaScript e varredura autenticada. 26
4.2 Visão geral dos resultados obtidos nos testes realizados nessa pesquisa. 27
4.3 Páginas não vulneráveis com resposta HTTP 200 de cada ferramenta. 28
4.4 Páginas vulneráveis com resposta HTTP 200 de cada ferramenta. 28
4.5 Porcentagem de falsos negativos em decorrência do mal funcionamento do módulo *crawler*. 29

Capítulo 1

Introdução

Proteger informações pessoais é uma preocupação para a sociedade. Desde o surgimento da internet, ela tem se tornado ainda maior, pois é necessário proteger os dados em um meio público. A segurança da informação proporciona ferramentas e métodos que tentam proteger informações confidenciais em ambientes públicos, porém, ainda é necessário melhorar os esforços na área de segurança da informação [PZK16], mesmo para vulnerabilidades bem conhecidas, como a *Cross Site Scripting* (XSS) [RTFB20]. Os sistemas Web geralmente possuem uma grande quantidade de dados confidenciais e são sistemas que podem ser acessados por qualquer pessoa. Por conter dados sensíveis, é indispensável que técnicas de desenvolvimento de software seguro sejam adotadas [DCV10]. Não existem sistemas Web totalmente seguros, ou seja, toda transação realizada na Internet pode ser capturada por pessoas mal-intencionadas que usam os dados para benefício próprio.

É importante tomar medidas preventivas a fim de evitar o acesso indevido a dados pessoais dos usuários. Uma forma eficiente de fazer isso é submeter os sistemas Web a análises de segurança, pelas quais haverá tentativas exaustivas de encontrar falhas de segurança no sistema analisado [DCV10]. Um dos tipos de teste de segurança é o teste de intrusão, também conhecido como *pentest* [DZ17]. Para o *pentest* em sistemas Web, existem algumas ferramentas que tentam encontrar vulnerabilidades de forma automatizada e sem informações privilegiadas, apresentando um relatório com os resultados obtidos. Essas ferramentas são conhecidas como *scanners* de vulnerabilidade de caixa preta [FBJ+16]. No entanto, as ferramentas existentes atualmente apresentam defeitos: ignoram vários tipos de vulnerabilidades, apresentam muitos falsos positivos e falsos negativos, e não conseguem detectar todas as páginas que existem no sistema, deixando de testar algumas funcionalidades [DCV10], [JK16] e [DTPP18]. Para Deepa *et al.* (2018), apesar de os *scanners* de vulnerabilidade para aplicações Web terem sua eficácia aprimorada nos últimos anos, ainda há muitas lacunas a serem preenchidas [DTK+18].

Há estudos [DTPP18, AAA+17] que apontam a capacidade de indexação dos *crawlers* como a principal limitação dos *scanners* atuais. Os *crawlers* são responsáveis por identificar todas as páginas existentes em um sistema, encontrar onde há formulários, identificar onde é possível realizar *upload* de arquivos e identificar as páginas que necessitam de autenticação. Neste trabalho, foi adotado o *crawler* como módulo rastreador.

Um *scanner* de vulnerabilidade Web, teoricamente, é dividido em três módulos: módulo rastreador, módulo atacante e módulo analisador [DCV10], sendo o primeiro o responsável por uma das maiores limitações dessas ferramentas [Sut10]. Se esse módulo falhar, o *scanner* não será capaz de procurar vulnerabilidades em todo o aplicativo submetido a teste, tendo uma visão incompleta do

sistema.

O *crawler* proposto nesta pesquisa foi desenvolvido com o objetivo de aprimorar a técnica de rastreabilidade de páginas Web, para aumentar a eficácia na cobertura de *scanners* de vulnerabilidade de aplicações Web. Foi atribuído o nome "Roudan" para a ferramenta desenvolvida.

1.1 Motivação

Durante os testes iniciais deste trabalho, foram identificados alguns problemas nos *scanners* de vulnerabilidades para aplicações Web de código aberto ainda sem a utilização do Roudan. A tabela 4.2 apresenta uma visão geral das limitações identificadas com resultados não confiáveis e muitos falsos negativos. É desejável que esses *scanners* automatizem e facilitem os testes de segurança de forma que testadores possam descobrir vulnerabilidades com mais agilidade e possam corrigi-las previamente. Para que a análise da ferramenta seja útil, o testador não pode gastar muito tempo com falsos positivos e falsos negativos devido à baixa cobertura do *crawler*.

1.2 Objetivo

O objetivo do presente estudo é apresentar um aprimoramento do módulo de rastreamento (*crawler*) de *scanners* de vulnerabilidades (*open source*) de aplicações Web para testes de intrusão automatizados, realizando um experimento comparativo entre a ferramenta criada e *scanners* de código aberto existentes.

1.3 Contribuições

Este trabalho traz como contribuição o aprimoramento do módulo rastreador de *scanners* de vulnerabilidades de aplicações Web. Os experimentos demonstram que, ao adotar esse aprimoramento proposto, os *scanners* melhoraram seu desempenho e capacidade de detecção de vulnerabilidades, visto que o módulo rastreador é responsável por encontrar e indexar todas as funcionalidades de um sistema em teste. Caso adotem os aprimoramentos desenvolvidos neste trabalho, essas ferramentas apresentarão menos falsos negativos. O código fonte do *crawler* proposto nesta pesquisa encontra-se disponível em https://github.com/escuderoDP/crawler.

Até a finalização da dissertação, não foi identificado nenhuma outra análise focada no módulo *crawler* dessas ferramentas, como é a proposta aqui. Por ser focada no módulo *crawler*, ela permite ter uma visão mais ampliada dos defeitos apresentados nesse módulo para que seja possível sugerir aprimoramentos direcionados. Além disso, sua utilização pode tornar os sistemas Web mais seguros, visto que a maioria das vulnerabilidades é advinda de configurações inadequadas, falta de atualizações ou problemas básicos de programação. A maioria das vulnerabilidades não é causada de forma intencional e pode passar despercebida por usuários e gestores de sistemas.

Parte desta pesquisa foi aprovada e apresentada no 5º Workshop Regional de Segurança da Informação e de Sistemas Computacionais (WRSeg 2020) - evento integrante da Escola Regional de Redes de Computadores (ERRC) - que proporciona um fórum para discussão e apresentação de trabalhos científicos e técnicos nas áreas de segurança da informação e de sistemas computacionais. O

trabalho aprovado foi um estudo comparativo do módulo rastreador de *scanners* de vulnerabilidade Web de código aberto.

1.4 Organização do texto

2 Conceitos

Nesta seção, são apresentados os conceitos necessários para entender o presente trabalho.

2.1 Teste de segurança

Neste tópico, são apresentadas definições sobre teste de segurança, tais como: as etapas necessárias para a execução de um teste de segurança; projetos de teste de segurança; técnicas de teste de segurança; e *frameworks* para testes de segurança.

2.1.1 Técnicas de teste de segurança

Aqui, são definidas as técnicas e fases de teste de segurança, sendo o teste de intrusão e a análise dinâmica a técnica adotada nesta pesquisa.

2.1.2 *Frameworks* para testes de segurança

Neste ponto, são apresentados os principais *frameworks* para testes de segurança com suas respectivas descrições.

2.2 Vulnerabilidades mais frequentes

São citadas e definidas as vulnerabilidades mais frequentes em sistemas Web.

2.3 *Scanner* de vulnerabilidades

Apresenta-se, aqui, a especificação desse tipo de ferramenta e seus três módulos: rastreador, atacante e analisador.

2.3.1 *Crawler*

Neste tópico, é definido o módulo *crawler*, também conhecido como módulo rastreador, de um *scanner* de vulnerabilidades. O módulo *crawler* foi estudado e aprimorado nesta pesquisa.

2.4 Propriedades de segurança

Neste item, são apresentadas as propriedades e requisitos necessários para garantir a segurança da informação.

2.5 Considerações

A partir dos conceitos elencados, são apresentadas algumas considerações.

3 Roudan

Neste capítulo, é definida uma nova abordagem de módulo *crawler* para *scanners* de vulnerabilidades para sistemas Web, ou seja, trata-se de uma contribuição deste trabalho ao desenvolver um novo *crawler*, batizado de Roudan, que apresenta melhorias significativas em relação aos *crawlers* existentes.

4 Estudo de caso

Nesta seção, são descritos os experimentos realizados nesta pesquisa e exibidos os resultados obtidos.

4.1 Método e descrição dos testes

Neste tópico, é descrito, de forma geral, o método utilizado e a descrição dos testes, bem como as configurações físicas e virtuais adotadas, especificando o ambiente de teste utilizado.

4.2 *Scanners* de vulnerabilidades testados

Neste item, são especificados os *scanners* de vulnerabilidades para aplicações Web testados e as versões utilizadas.

4.3 Sistemas Web testados

Aqui, são apresentados os sistemas Web utilizados no estudo de caso.

4.4 Métricas de desempenho adotadas

Neste ponto, são elencadas e definidas todas as métricas de desempenho avaliadas neste estudo.

4.5 Resultados e discussão dos experimentos

Nesta subseção, são apresentados os resultados obtidos nos testes realizados por meio de tabelas e discussões.

4.5.1 WackoPicko

Aqui, são apresentados os resultados obtidos para o sistema WackoPicko, desenvolvido em PHP.

4.5.2 OWASP *Juice Shop*

Aqui, são apresentados os resultados obtidos para o sistema OWASP *Juice Shop*, desenvolvido em Node.js, Express e Angular.

4.6 Considerações

Neste tópico, são apresentadas algumas considerações a respeito dos resultados obtidos nos experimentos.

5 Trabalhos relacionados

Neste capítulo, são discutidos alguns trabalhos relacionados com a presente pesquisa.

5.1 Considerações

Neste item, são apresentadas algumas considerações em relação aos trabalhos relacionados.

6 Conclusões

Apresentam-se, aqui, as conclusões obtidas após a realização deste trabalho.

Capítulo 2

Conceitos

Os sistemas web são alvos frequentes de pessoas mal-intencionadas que podem usar os dados acessados para fins criminosos, logo, é necessário a realização de testes de segurança para identificar vulnerabilidades e corrigi-las o mais rápido possível. Não se pode afirmar que um sistema seja livre de vulnerabilidades, mas para garantir que os requisitos de segurança de um sistema sejam atendidos, necessita-se da aplicação de técnicas de teste de segurança apropriadas, que sejam eficazes e eficientes [FBJ+16].

O protocolo utilizado para comunicação com servidores web é o HTTP (*Hypertext Transfer Protocol*) ou HTTPS, quando adicionados protocolos criptográficos [IET00]. O HTTP é um protocolo sem estados, isso significa que cada comunicação com um servidor web é tratada de forma independente e informações de transações diferentes também são independentes. Devido à natureza sem estado do protocolo HTTP [LX11], as variáveis de sessão são explicitamente definidas nos aplicativos da web para manter o estado de uma sessão na web. As variáveis de sessão podem ser mantidas no lado cliente ou no lado servidor.

Uma vulnerabilidade é um comportamento não esperado capaz de comprometer a segurança de um sistema. A exploração de vulnerabilidades de segurança pode causar custos elevados, tais como um período de tempo inativo ou a modificação e perda de dados. A maior parte dos incidentes de segurança é provocada por invasores que exploram falhas de segurança já conhecidas [FBJ+16]. Aplicar técnicas de teste de segurança é uma medida eficiente e eficaz de identificar vulnerabilidades e garantir que os requisitos de segurança de um sistema sejam atendidos.

É necessário fazer uma avaliação do estado de segurança de forma contínua para entender os riscos que existem em um sistema. Essa avaliação é geralmente realizada por meio de testes de segurança [DZ17]. Logo, é importante entender as técnicas e ferramentas existentes para o auxílio da avaliação de segurança a fim de minimizar os riscos existentes em um sistema.

Segundo Felderer *et al.* (2016), os requisitos de segurança de um sistema podem ser positivos e funcionais, definindo explicitamente a funcionalidade de segurança esperada de um mecanismo de segurança; ou negativa e não funcional, especificando o que o aplicativo não deve fazer. Os testes realizados em um sistema dependem diretamente de como são classificados esses requisitos. Para requisitos de segurança positivos, podem ser aplicadas técnicas de teste clássicas, enquanto que para requisitos de segurança negativos devem-se adotar medidas adicionais, como análises de risco, testes de intrusão ou base de conhecimento de vulnerabilidades.

2.1 Teste de segurança

O teste de segurança simula ataques que tentam comprometer a segurança de uma aplicação, desempenhando o papel de um atacante que tenta explorar suas vulnerabilidades. O teste de segurança em sistemas web é de fundamental importância, pois a crescente popularidade, disponibilidade, facilidade de uso e a grande base de usuários tornam os aplicativos da web atraentes para os cibercriminosos, e por isso, têm sido alvo de grandes ataques [DTPP18].

Para realizar um teste de segurança, deve-se ter um planejamento prévio e cumprir algumas etapas, conforme ilustrado na Figura 2.1. Para Felderer *et al.* (2016), o planejamento de teste é a atividade de criar ou atualizar um plano de teste. Segundo os autores, esse plano inclui os objetivos, o escopo e os métodos de teste, bem como os recursos e o cronograma das atividades pretendidas. Ao finalizar, os critérios de saída são avaliados e os resultados do teste registrados são resumidos em um relatório de teste. Após executar o teste planejado, deve-se gerar um relatório com os resultados obtidos.

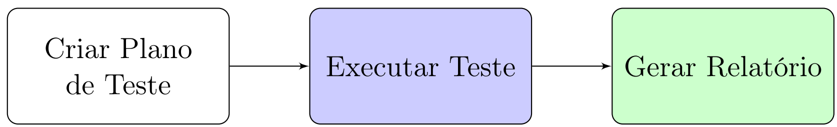

Figura 2.1: *Etapas necessárias para a execução de um teste de segurança.*

O teste de segurança adequado requer uma combinação de técnicas, pois não existe uma técnica de teste única que possa ser executada e abranja de maneira eficaz todos os testes de segurança necessários para analisar um sistema. No entanto, muitas organizações adotam apenas uma abordagem de teste de segurança [FBJ+16]. Em relação aos projetos de testes de segurança, eles podem ser classificados em: teste de caixa preta, teste de caixa branca e teste de caixa cinza [DZ17]. A Figura 2.2 mostra uma síntese dessas classificações. Dalalana Bertoglio e Zorzo (2017) afirmam que nos testes de caixa branca há o conhecimento prévio completo do sistema e os casos de teste são derivados com base nas informações sobre como o software foi projetado ou codificado; nos testes de caixa preta, os casos de teste dependem apenas do comportamento de entrada/saída do software, permitindo simular invasões; no teste da caixa cinza, a quantidade de informações sobre o alvo não está completa, mas também não é inexistente.

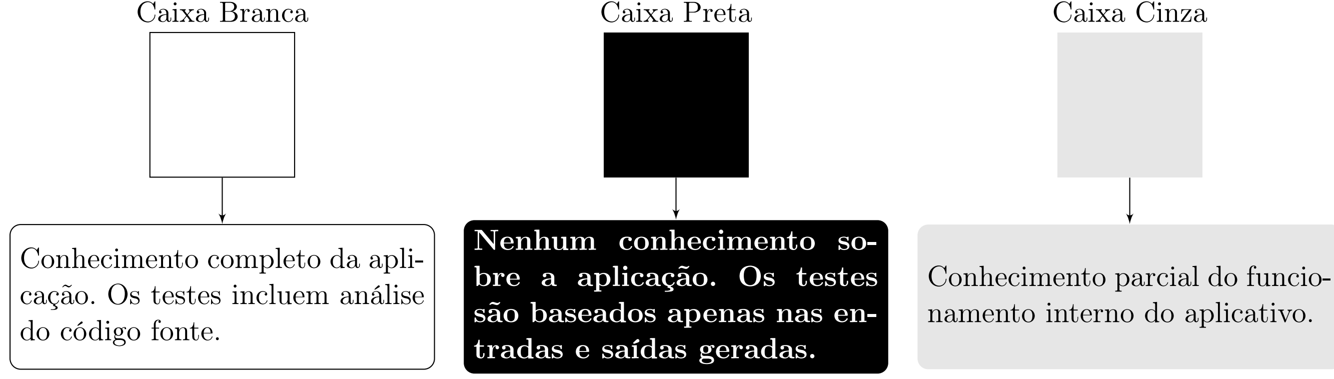

Figura 2.2: *Projetos de testes de segurança.*

2.1.1 Técnicas de teste de segurança

Felderer *et al.* (2016) classificam quatro tipos principais de técnicas de teste de segurança: teste de segurança baseado em modelo, teste baseado em código e análise estática, teste de intrusão e a análise dinâmica, e teste de regressão de segurança.

O teste de segurança baseado em modelo é fundamentado em requisitos de projeto criados durante a fase de análise e projeto do desenvolvimento da aplicação. Nesse tipo de teste, alguns algoritmos são selecionados de forma manual e geram, automaticamente e de forma sistemática, casos de teste de um conjunto de modelos do sistema em teste ou de seu ambiente. O teste baseado em modelo é o que tem menor custo para a empresa e pode encontrar vulnerabilidades de forma precoce. Uma adaptação do teste baseado em modelo origina o teste de vulnerabilidade baseado em risco, adaptando as técnicas de teste para a geração de casos de teste de acordo com riscos previamente identificados no desenvolvimento da aplicação [FBJ+16].

Os testes baseados em código e análise estática são baseados no código fonte durante a fase de desenvolvimento da aplicação, portanto, são considerados testes de caixa branca, pois possuem informações privilegiadas do sistema. Felderer *et al.* (2016) afirmam que nesse tipo de análise é possível detectar vulnerabilidades nos estágios iniciais do ciclo de desenvolvimento da aplicação sem alto custo. A análise do código fonte pode ser feita de forma manual ou de forma automatizada: quando é feito de forma manual, deve-se ler linha por linha do código fonte, o que exige alto conhecimento do profissional sobre a ferramenta, sobre a linguagem de programação utilizada e sobre segurança da informação; quando é feito de forma automatizada, é possível que a análise seja feita por desenvolvedores que não são especialistas em segurança, pois as ferramentas que executam a verificação de código tentam automatizar e simplificar o processo. No entanto, mesmo com a utilização de ferramentas de automatização, é necessário uma verificação manual posterior. Primeiro, deve-se verificar as supostas vulnerabilidades apontadas pela ferramenta e decidir se a descoberta realmente representa uma vulnerabilidade que possa ser explorada por alguém mal intencionado, para que ela seja corrigida. Caso a detecção apontada pela ferramenta não possa ser explorada por algum invasor, então tem-se um falso positivo e não é necessário realizar a correção. No teste, pode ser gerado também falsos negativos, ou seja, há vulnerabilidades que não foram detectadas [FBJ+16].

O teste de intrusão e a análise dinâmica são baseados em sistemas em execução e não dependem de informações privilegiadas, sendo classificados como testes de caixa preta. Um teste de intrusão, também conhecido como *pentest* [DZ17], é um mecanismo muito utilizado em sistemas web, em uma configuração bem similar ao de um atacante. Dalalana Bertoglio e Zorzo (2017) dividem o processo de *pentest* nas seguintes atividades: coleta de dados do sistema alvo; escanear o sistema alvo para identificar os serviços e protocolos disponíveis; identificar sistemas e aplicativos existentes que estão sendo executados no sistema alvo; identificar e explorar as vulnerabilidades conhecidas nos sistemas e aplicações. Já o NIST (*National Institute of Standards and Technology*) publicou um guia técnico no qual divide o teste de intrusão em quatro fases distintas: planejamento, descoberta, ataque e relatório [SS08]. Segundo o NIST, no planejamento não ocorre nenhum teste, analisa-se as condições e limites, definindo o escopo do teste e documentando os procedimentos a serem tomados nos testes; na fase da descoberta, primeiramente é feita a descoberta e enumeração de todas as interfaces externas acessíveis do sistema em teste. Após realizar sistematicamente a enumeração de todas as interfaces acessíveis, é feita uma identificação de classes de vulnerabilidades aplicáveis a

cada interface. Isso é feito para direcionar o testador no momento do ataque. Somente na terceira fase, no ataque, as interfaces identificadas são testadas por meio de uma sequência de tentativas de ataque: os testadores tentam comprometer o sistema, enviando cargas de ataque. Se houver vulnerabilidades de segurança encontradas, elas serão exploradas para obtenção de mais informações sobre o sistema, ampliar os privilégios de acesso do testador e outros componentes do sistema, podendo comprometer até mesmo interfaces não documentadas no planejamento do ataque. Nesse caso, é necessário retornar à fase de descoberta, conforme observado na figura 2.3; a última fase, relatório, deve ocorrer simultaneamente com as outras três fases, documentando todas as descobertas e especificando a gravidade estimada. Na maioria das vezes, um teste de intrusão não é feito exclusivamente com o uso da ferramenta automatizada, combinando também com configurações e análises manuais para garantir maior eficácia no teste [SS08].

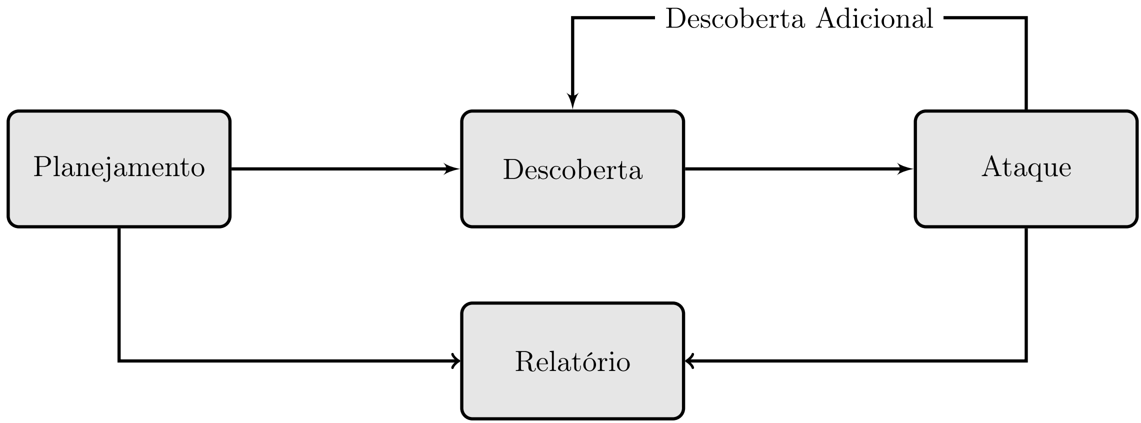

Figura 2.3: *Fases de um teste de intrusão. Adaptado de: [SS08].*

Por fim, o teste de regressão de segurança é executado durante manutenções do sistema. Essas manutenções de sistema precisam ser bem acompanhadas e verificar se novas vulnerabilidades não são criadas. O teste de segurança em todo o sistema não é considerado uma boa prática nesta fase, é preciso selecionar o que será testado e acompanhar as manutenções realizadas. Em geral, as partes selecionadas são as que sofreram modificações [FBJ+16]. A figura 2.4 mostra quando cada técnica de teste é utilizada em um ciclo de vida de desenvolvimento de um software. Uma única técnica não consegue abranger todas as fases.

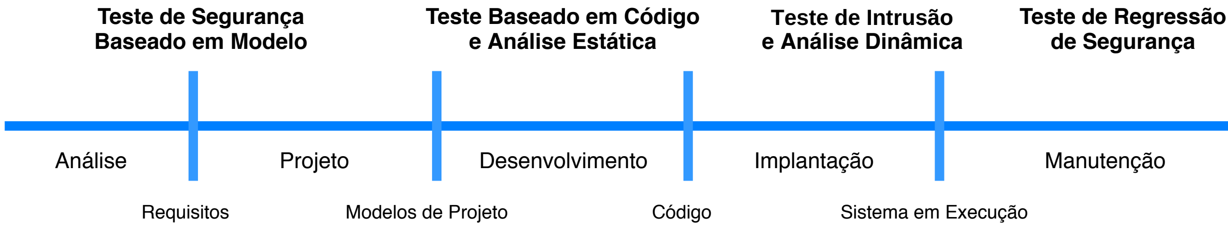

Figura 2.4: *Técnicas de Teste de Segurança no Ciclo de Vida de Desenvolvimento de software. Adaptado de: [FBJ+16]*

.

2.1.2 *Frameworks* para testes de segurança

Existem diversos *frameworks* a fim de auxiliar na elaboração de planos de teste, como por exemplo, o SDL (*Security Development Lifecycle*) da Microsoft [HL06]. O SDL é um conjunto de práticas que visam à garantia de segurança para projetos de desenvolvimento de software, no qual a segurança é um atributo de qualidade integrado que afeta todo o ciclo de vida do software. Os sistemas de software são construídos de forma a minimizar o dano potencial causado pelos invasores. Com esse conjunto de práticas, a implantação de um sistema é feita com acompanhamento de ferramentas e orientações as quais dão suporte aos usuários e administradores [HL06].

A *Open Web Application Security Project* (OWASP) publicou um guia técnico especificando a proporção do esforço a ser feito em cada etapa no ciclo de vida do desenvolvimento de software. A Figura 2.5 ilustra todo o ciclo, que é dividido em cinco fases genéricas: análise, projeto, desenvolvimento, implantação e manutenção [OWA20], como também a proporção do esforço a ser dedicado em cada fase. Isso deixa evidente que a utilização de apenas uma técnica para realização de testes de segurança é insuficiente e que os projetos de testes mostrados na figura 2.2 devem ser utilizados de forma combinada.

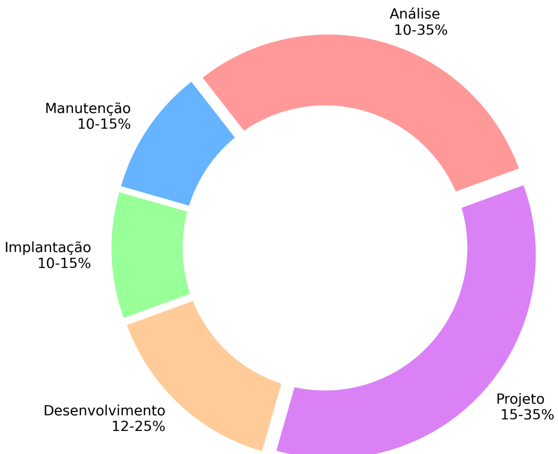

Figura 2.5: *Proporção do esforço de realização de testes em cada fase do ciclo de desenvolvimento de software. Adaptado de: [OWA20]*

Há outros *frameworks* que auxiliam no processo de teste de segurança e no desenvolvimento, implementação ou governança de software de forma segura, tais como o PTES (Penetration Testing Execution Standard) [PTE14] e OWASP SAMM (Open Software Assurance Maturity Model) [OWA19b] e o SDL (Security Development Lifecycle) [HL06].

Enquanto o PTES cobre desde a comunicação inicial e raciocínio necessário em um *pentest*, através da coleta de informações sobre a organização testada e fases de modelagem de ameaças, como também a pesquisa de vulnerabilidades, exploração, pós-exploração e relatório, que captura todo o processo de uma maneira que faça sentido para o cliente [PTE14].

O OWASP SAMM, por sua vez, é um *framework* aberto que auxilia na formulação e implementação de uma estratégia de segurança de governança, construção, verificação e implantação de um software. O OWASP SAMM foi definido de forma flexível, podendo ser utilizado por organizações pequenas, médias e grandes, ou até mesmo para um projeto individual, usando qualquer estilo de desenvolvimento. Esse *framework* classifica a prática de segurança em quatro níveis, iniciando no nível zero. Cada nível possui especificações que devem ser atingidas para obter o nível desejado

[OWA19b].

Já o SDL é um conjunto de práticas que suportam a garantia de segurança para projetos de desenvolvimento de software, no qual a segurança é um atributo de qualidade integrado que afeta todo o ciclo de vida do software. Os sistemas de software são construídos de forma que o dano potencial causado pelos invasores seja minimizado. Com esse conjunto de práticas, a implantação de um sistema é feita com acompanhamento de ferramentas e orientações que dão suporte aos usuários e administradores [HL06].

Todos os *frameworks* apresentados enfatizam ser imprescindível a realização de testes para a descoberta de vulnerabilidades. O *framework* PTES pode auxiliar o testador em um *pentest*, técnica aplicada nos experimentos desta pesquisa, ajudando a descobrir vulnerabilidades e proporcionar um software mais seguro.

2.2 Vulnerabilidades mais frequentes

Apesar da análise de vulnerabilidades não ser o foco principal desta pesquisa, é importante conhecer algumas definições para melhor compreensão do trabalho. Muitas aplicações web possuem vulnerabilidades bem comuns e conhecidas. A OWASP possui uma lista com a classificação das dez vulnerabilidades mais graves para aplicações web de organizações. A última lista emitida, em 2021, relata que os ataques de injeção e autenticação quebrada são os que apresentam o maior risco, o que acontece desde 2013 [OWA21b]. A OWASP realiza essa classificação de risco com base na estimativa da probabilidade da vulnerabilidade existir, a facilidade de exploração e o impacto que pode causar em uma organização [OWA19a].

O CVE (Common Vulnerabilities and Exposures) é um sistema que fornece um método de referência para vulnerabilidades e exposições de segurança da informação conhecidas [CVE22]. Até o mês de agosto, as vulnerabilidades com o maior número de registros em 2022 no CVE, foram de execução de código, XSS (*Cross Site Scripting*) e DoS (*Denial of Service*). As vulnerabilidades XSS permitem que um invasor altere o contexto dos dados para o contexto do código usando caracteres especiais, podendo executar códigos JavaScript maliciosos [NCS17]. Já a vulnerabilidade SQL *Injection* permite manipular, criar e executar consultas SQL arbitrárias.

O principal motivo de haver tantas aplicações web com vulnerabilidades consideradas graves é o fato dos desenvolvedores geralmente terem um curto prazo para o desenvolvimento do sistema, além de não receberem treinamentos suficientes sobre desenvolvimento de software seguro, com exceção de grandes empresas [AAZ+17]. Alzahrani *et al.* (2017) afirmam que as vulnerabilidades mais comuns são proteção insuficiente da camada de transporte, vulnerabilidade de vazamento de informações, XSS e SQL *Injection*. Segundo os autores, a proteção insuficiente da camada de transporte significa a não utilização de criptografia ou a utilização de algoritmos considerados inseguros; já a vulnerabilidade de vazamento de informações é uma fraqueza em aplicativos da web pela qual dados do sistema ou informações de depuração são revelados. Essas informações podem ser exploradas e ajudam um invasor a atacar o sistema.

Outra vulnerabilidade comum é a violação de estados em aplicativos web, que exploram vulnerabilidades lógicas e permitem o acesso a funções e dados confidenciais. Para um ataque de violação de estados, existem três tipos de vulnerabilidades possíveis: definição insuficiente de variáveis de sessão para diferenciar todos os estados possíveis; verificação insuficiente de variáveis de sessão em

pontos de programa apropriados; verificação errônea de variáveis de sessão que podem ser ignoradas [LX11].

A quantidade de vulnerabilidades tem aumentado muito [MK15] e [CVE22], portanto, verificar todas as vulnerabilidades da web manualmente é difícil e demorado. Assim sendo, é necessário um *scanner* de vulnerabilidades de aplicações web para automatizar as varreduras, visto que essas ferramentas são capazes de encontrar vulnerabilidades comuns conhecidas.

2.3 *Scanner* de vulnerabilidades

A realização de testes de segurança permite descobrir novas vulnerabilidades e se antecipar aos riscos [PSM15]. Uma das técnicas mais utilizadas em aplicações web é o *pentest* [HCO11], sua realização em aplicativos web é frequente na utilização de *scanners* de vulnerabilidades de caixa preta, que operam lançando ataques contra o aplicativo e observando sua resposta a esses ataques [sec].

Os *scanners* de vulnerabilidades são ferramentas que tentam explorar vulnerabilidades de forma automática, com pouca ou nenhuma interação humana. Essas ferramentas acessam as aplicações da mesma forma que usuários comuns acessam, portanto, o *scanner* é independente da tecnologia particular usada para implementar a aplicação em teste [DCV10]. Por exemplo, em sistemas web, a ferramenta deve ser capaz de realizar testes de segurança independentemente da linguagem de programação adotada.

Uma possível abordagem para os *scanners* de aplicativos da web é que eles consistem em três módulos principais: módulo rastreador (*crawler*), um módulo atacante e um módulo analisador. No módulo rastreador, o *scanner* fará uma varredura completa do sistema, tentando encontrar todas as URLs acessíveis. Já no módulo atacante, cada URL descoberta será analisada e são verificados os pontos de entrada para enviar ataques que podem acionar vulnerabilidades do sistema. Por fim, o módulo analisador verifica as páginas retornadas pela aplicação web em resposta aos ataques lançados pelo módulo atacante para detectar possíveis vulnerabilidades e fornecer *feedback* aos outros módulos [DCV10].

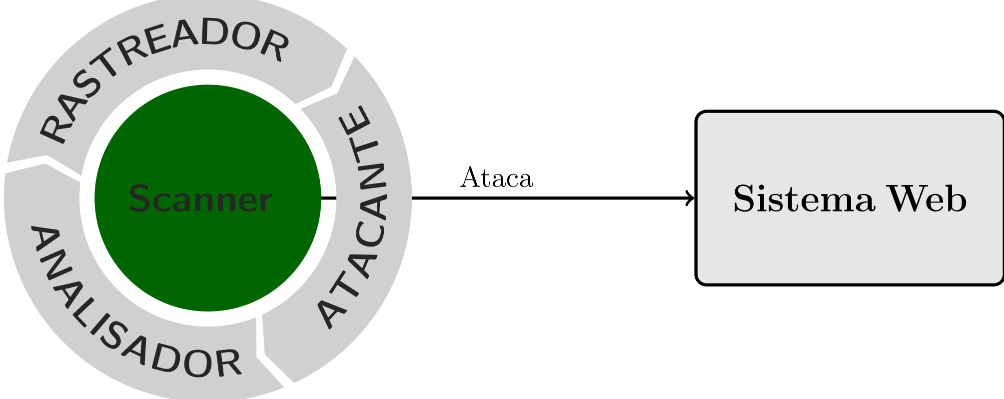

Figura 2.6: *Módulos de um scanner de vulnerabilidades web.*

Na figura 2.6, atacar o sistema web significa atacar o servidor web e analisar as respostas obtidas. Porém, um servidor web pode conter mais de uma aplicação e o teste de segurança pode ser realizado

em uma aplicação específica.

Os *scanners* de vulnerabilidades de caixa preta são ferramentas que podem ser usadas para identificar problemas de segurança em aplicações web [DCKV12]. Essas ferramentas também devem ser capazes de acessar e testar os vários componentes do aplicativo, que muitas vezes estão ocultos por trás de formulários como links gerados por JavaScript [BBGM10].

2.3.1 *Crawler*

Para este trabalho, adotou-se 'rastreador' como tradução da palavra *crawler*, portanto, ambas são sinônimas. É imprescindível que esse módulo seja eficaz para que todas as funcionalidades de um sistema sejam testadas. Kalso *et al.* (2006) apontam que o *crawler* percorre a árvore de links, coletando todas as páginas e formulários da web incluídos durante o processo. Para Liu *et al.* (2015), o módulo rastreador é a função básica mais importante da ferramenta de teste automatizado da web: começa a varredura a partir de um URL inicial, identifica todos os hiperlinks na página e os adiciona à lista de URLs a visitar [xss].

Alguns estudos [FBJ+16, AAA+17, DTPP18, Sha20] apontam que a principal limitação dos *scanners* atuais é a capacidade de indexação dos *crawlers*. Os *crawlers* são responsáveis pela capacidade de indexação de páginas dos *scanners*. A ferramenta precisa ser capaz de percorrer todas as páginas de um sistema e de realizar transações legítimas, acessando todas as funcionalidades do sistema que um usuário comum acessaria. Caso o *scanner* não consiga percorrer todas as páginas da aplicação, haverá funcionalidades não testadas que podem conter vulnerabilidades.

2.4 Propriedades de segurança

Os aspectos de segurança podem ser classificados em três níveis: infraestrutura de rede, sistema operacional e a própria aplicação. Todos os níveis estão suscetíveis a testes de segurança[FBJ+16], estes verificam e validam os requisitos da aplicação relacionados a propriedades de segurança, que são: confidencialidade, integridade, disponibilidade, autenticação, autorização e não repúdio [dJJM15].

Felderer *et al.* (2016) definem que a confidencialidade é a garantia de que as informações não sejam divulgadas a indivíduos, processos ou dispositivos não autorizados; já a integridade é a garantia de que os dados não são modificados ou destruídos de sua origem; enquanto que a disponibilidade é a garantia de acesso oportuno e confiável a dados e serviços para usuários autorizados; a autenticação é a garantia da validade de uma transmissão, mensagem ou origem, ou um meio de verificar a autorização de um indivíduo para receber categorias específicas de informações; a autorização, por sua vez, estabelece as permissões de acesso concedidas a um usuário ou processo; e, por fim, a irretratabilidade é a garantia de que nenhum dos participantes de uma transação possa negar ter participado dela.

A segurança pode ser estabelecida por três mecanismos de proteção: prevenção, detecção e resposta. A prevenção é o processo de tentar impedir que os intrusos tenham acesso aos recursos do sistema; a detecção ocorre quando o intruso teve sucesso ou está no processo de obter acesso ao sistema; finalmente, a resposta é um mecanismo posterior que tenta responder à falha dos dois primeiros mecanismos [DZ17].

Segundo Felderer *et al.* (2016), uma prática comum, porém não recomendada, é a não realização de testes de segurança durante o desenvolvimento da aplicação. O autor acrescenta que é

importante realizar testes de segurança durante todo o ciclo de vida de desenvolvimento da aplicação, salientando o quanto sistemas são implementados sem terem passado por testes suficientes para garantia das propriedades de segurança. Durante a fase de projeto da aplicação, as análises de risco fornecem meios eficazes para orientar os testes de segurança e, assim, detectar falhas e vulnerabilidades [FBJ+16].

2.5 Considerações

Nesse capítulo foram apresentados os conceitos fundamentais para compreender os trabalhos realizados e apresentados, disponíveis nos próximos capítulos. A utilização de *scanners* de vulnerabilidades é imprescindível para a realização de teste de segurança e para ajudar na garantia das propriedades de segurança de um sistema, além de auxiliar no desenvolvimento seguro de software. Para este trabalho, adotou-se a classificação de *scanners* de vulnerabilidades para aplicações web apresentada por Doupé *et al.* (2012), que definem três módulos para esse tipo de ferramenta: módulo rastreador (*crawler*), um módulo atacante e um módulo analisador.

Capítulo 3

Roudan

O *crawler* proposto nesta pesquisa foi desenvolvido com o objetivo de aprimorar a técnica de rastreabilidade de páginas Web, para aumentar a eficácia na cobertura de *scanners* de vulnerabilidade de aplicações Web. Foi atribuído o nome "Roudan" para o *crawler* desenvolvido.

Os estudos [LX11, DCV10] apontam que para um *crawler* ser eficiente ele precisa ser capaz de renderizar código JavaScript [Jav21], pois em sistemas Web modernos desenvolvidos nessa linguagem de programação, muitos links e redirecionamentos ficam escondidos. Para Li e Xue (2011), o método tradicional, no qual os *crawlers* são incapazes de renderizar códigos JavaScript, tem pouca cobertura de pontos de injeção ocultos. Em oposição ao método tradicional, o Roudan sempre renderiza a página utilizando o *browser* Mozilla *Firefox* [Moz21a] antes de escanear o sistema Web em busca dos links, a fim de mostrar os pontos de injeção ocultos.

O Roudan foi desenvolvido utilizando a linguagem de programação *python* [Pyt21] e, para a renderização de JavaScript, utilizou-se a biblioteca *selenium* [Sel21]. Esta proporciona a automação de navegadores e possui um `WebDriver` que permite criar testes robustos de automação de regressão baseada em navegador. O testador deve passar uma URL de entrada ao Roudan, a qual será acessada utilizando o `WebDriver` e renderizada como se o acesso fosse feito pelo *browser* Mozilla *Firefox*. Todos os links encontrados são adicionados a um conjunto de links a serem escaneados. Esse método é usado por todos os acessos realizados pela ferramenta.

Para encontrar os links na página, foi utilizada a biblioteca *python* chamada *Beautiful Soup* [Bea21]. Esta foi modelada para projetos de resposta rápida, como, por exemplo, uma captura de tela. A *Beautiful Soup* fornece um *kit* de ferramentas para analisar um documento e extrair os dados com pouco código. O *crawler* pesquisa por todos os links e redirecionamentos HTML (*HyperText Markup Language*) [RLHJ+99] ou JavaScript listados na Tabela 3.1.

Após extrair todos os links, os que são externos ao endereço de entrada passado ao Roudan são descartados para que não sejam escaneados. O *crawler* desenvolvido também é capaz de identificar páginas geradas dinamicamente para que aplicações utilizando o método *GET* do HTTP não fiquem em *loop* infinito, o que poderia acontecer em páginas geradas a partir de um calendário, por exemplo.

Ao passar uma URL de entrada, a primeira etapa será descobrir a página de *login*. Logo, o Roudan acessa a URL, extrai todos os links e identifica qual é a página de *login*. Em seguida, o rastreador renderiza a página de login e identifica os nomes dos campos necessários para realizar a autenticação no sistema. Antes de escanear toda a aplicação Web, o Roudan tentará fazer a autenticação no sistema e criar uma sessão válida. Para isso, o testador pode passar como parâmetro um usuário e senha válidos para garantir que a autenticação será bem sucedida, mas, caso não seja,

Tabela 3.1: *Links e redirecionamentos HTML e JavaScript.*

Tag	**Definição**
`src`	Usado para referenciar um arquivo JavaScript ou imagem.
`href`	(*Hypertext REFerence*) é usado para criar um link para outra página.
`action`	Especifica para onde enviar os dados de um formulário.
`window.location`	Pode ser usado para obter o endereço da página atual (URL) e para redirecionar o navegador para uma nova página.
`window.open`	Abre uma URL em uma nova janela do navegador ou uma nova guia, dependendo das configurações do navegador e dos valores dos parâmetros.
`location.assign`	Carrega um novo documento.
`load`	Carrega os dados do servidor passados como parâmetros e coloca o HTML retornado nos elementos correspondentes.
`routerlink`	Link que inicia a navegação para uma rota. A navegação abre um ou mais componentes roteados em um ou mais locais na página.

o *crawler* saberá que a autenticação falhou. Se a autenticação falhar ou se o testador não passar um usuário e uma senha, será oferecido ao testador a tentativa de autenticação por meio de uma *wordlist* de usuários e senhas. A *wordlist* é um arquivo de texto com várias combinações de usuários e senhas. Caso a ferramenta consiga autenticar na aplicação, a sessão é guardada para ser usada em todos os próximos acessos.

A *wordlist* foi criada a partir da combinação das *wordlists* da ferramenta *Metasploit Framework* [Rap21a], a estrutura de *pentest* mais usada do mundo [Rap21b]. Inicialmente, foram selecionadas todas as *wordlists* com combinações de usuário e senha, removendo as combinações repetidas e combinações com usuário ou senha em branco, visto que a maioria das aplicações modernas não permitem clicar no botão de *login* se alguma entrada estiver em branco. As combinações contidas nesse documento são usuários e senhas padrões de diversas tecnologias e aplicações da Web, resultando em 2385 entradas diferentes.

É essencial criar uma sessão válida para escanear uma aplicação Web pois muitas páginas e funcionalidades são liberadas apenas após a realização da autenticação. Para realizar a autenticação, o Roudan simula uma interação humana, preenche os dados de *login* e faz um clique no botão de enviar os dados preenchidos. Ressalta-se que, para testar aplicações reais em ambiente de produção, é necessário implementar técnicas para enganar a aplicação sobre a origem das requisições de autenticação nos casos em que for usar a *wordlist*. Caso contrário, as requisições do *crawler* podem ser facilmente bloqueadas por um WAF (*Web application firewall*) [PLP15] ou um IPS (*Intrusion Prevention System*) [IPS].

Esposito *et al.* (2018) conjecturaram que *scanners* de vulnerabilidade de aplicativos da Web automatizados não possuem bom potencial e que duas das principais razões para isso são limitações com relação aos recursos de rastreamento e problemas para executar varreduras autenticadas [ERRW18]. Mesmo o Roudan não conseguindo uma sessão válida, será realizada a tentativa de extrair a maior quantidade possível de links. Nesse caso, muitas páginas e funcionalidades podem não

Listing 3.1: *Comando para executar o crawler.*

```
python3 crawler.py http://example.br username password
```

ser alcançadas. O rastreador possui um conjunto de links para escanear e outro conjunto para não escanear. É importante ter um conjunto de links para não escanear a fim de não realizar varreduras repetidas na aplicação testada.

A implementação do *crawler* descrito nesta seção, bem como a *wordlist*, instruções de uso e requisitos necessários para execução encontram-se no repositório [ET21]. Para que o Roudan funcione, é necessário copiar o `geckodriver` [Moz21b] para `/usr/local/bin` ou para `/usr/bin` com a finalidade de o *selenium* executar o `WebDriver` corretamente.

Tabela 3.2: *Versões da linguagem Python, das bibliotecas e do geckodriver utilizados.*

Recurso	Versão
Linguagem *Python*	3.8.10
Biblioteca beautifulsoup4	4.9.3
Biblioteca *selenium*	3.141.0
Geckodriver	v0.29.1-linux64

Para executar o Roudan, deve-se abrir o terminal no diretório em que foi feito o download do projeto [ET21] e executar o Comando 3.1. Para executar o *crawler* corretamente, deve-se utilizar as mesmas versões da linguagem *python* e das bibliotecas listadas na Tabela 3.2.

Os resultados obtidos são apresentados no próprio terminal ao testador. Um exemplo de *output* da execução do comando 3.1 é mostrado no *Output* 3.2. Cada item apresentado no *output* é um link que foi indexado e acessado, obtendo um código HTTP 200. Os links encontrados poderiam ser usados por um módulo atacante para realizar ataques e descobrir vulnerabilidades na aplicação.

Listing 3.2: *Output da execução do crawler na aplicação WackoPicko.*

```
The username and password  ['admin', 'admin']  is valid in the page  http
    ://127.0.0.1/admin/index.php?page=login . The Web system is vulnerable.

{'', 'users/login.php', '/pictures/view.php?picid=8', '/calendar.php?date
    =1632250766', '/calendar.php', '/users/logout.php', '/pictures/upload.
    php', 'sample.php', 'delete_preview_comment.php', '/cart/action.php?
    action=add&picid=15', '/pictures/view.php?picid=13', '/cart/action.php?
    action=add&picid=9', 'home.php', 'login.php', '/cart/action.php?action=
    add&picid=7', 'admin/', 'search.php', 'comments/', '/guestbook.php', '/
    users/login.php', '/pictures/recent.php', '/admin/index.php?page=login'
    , 'view.php', '/users/sample.php?userid=1', 'conflict.php', 'register.
    php', 'add_coupon.php', 'piccheck.php', 'view_flymake.php', 'pictures/
    search.php?query=default&x=38&y=12', '/', '/passcheck.php', '/users/
    view.php?userid=11', 'pictures/', '/users/view.php?userid=1', 'users/
    register.php', 'similar.php', 'preview_comment.php', '/admin/index.php?
```

```
page=create', 'users/home.php', 'purchased.php', 'conflictview.php', '
recent.php', 'pictures', '/users/similar.php', '/cart/action.php?action
=add&picid=14', '/users/register.php', '/cart/action.php?action=add&
picid=13', '?C=N;O=D', 'admin/index.php?page=login', 'cart', '/users/
home.php', 'pictures/search.php?query=defaultdefault&x=38&y=12', '/
pictures/search.php', 'action.php', '/piccheck.php', 'logout.php', '/
pictures/view.php?picid=10', 'pictures/upload.php', '?C=D;O=A', '/users
/view.php?userid=2', 'users', '/pictures/view.php?picid=9', 'confirm.
php', 'guestbook.php', 'comments/preview_comment.php', 'admin/index.php
?page=home', '/comments/preview_comment.php', '/calendar.php?date
=1632164366', 'upload.php', 'comments', '/cart/action.php?action=add&
picid=12', '/tos.php', 'cart/', '/pictures/view.php?picid=14', '/cart/
review.php', 'high_quality.php', 'add_comment.php', '/calendar.php?date
=1632164368', 'users/', 'check_pass.php', '?C=S;O=A', '/pictures/view.
php?picid=11', 'passcheck.php', '/cart/action.php?action=add&picid=8',
'/pictures/view.php?picid=7', 'admin', '/pictures/purchased.php', '?C=M
;O=A', '/pictures/view.php?picid=15', '/users/view.php?userid=9', '/
users/view.php?userid=10', 'review.php', '/pictures/view.php?picid=12'}
```

O Roudan foi implementado para que consiga preencher campos de texto de formulários e *inputs*, clicando no botão da página desenvolvido para enviar os dados ao servidor. Essa funcionalidade permite à ferramenta acessar recursos da aplicação em teste que só aparecem quando é feito o preenchimento e envio de dados, como, por exemplo, botões de pesquisa. A ferramenta simula um clique do *mouse* no botão encontrado, para obter um comportamento semelhante ao de um usuário legítimo.

A Figura 3.1 é um fluxograma que mostra como o Roudan é executado, começando pelo *input* do Comando 3.1, tendo como *output* todas as páginas encontradas, conforme *Output* 3.2. Na fase "Procurar páginas e funcionalidades por meio de *hiperlinks* e redirecionamentos" do fluxograma, o Roudan adiciona em seu conjunto de páginas encontradas apenas aquelas nas quais obteve acesso com resposta HTTP 200.

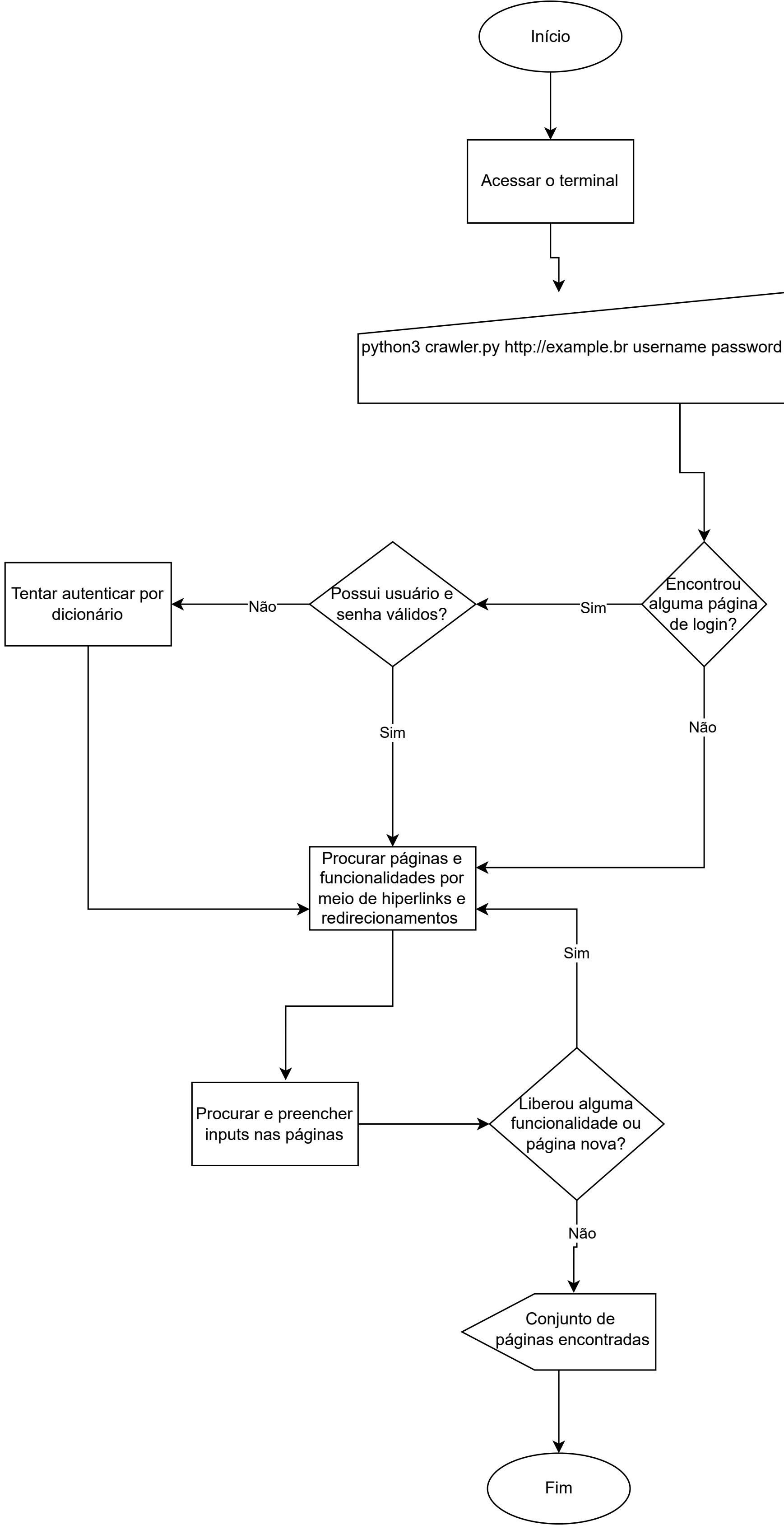

Figura 3.1: *Fluxograma de execução do Roudan.*

Capítulo 4

Estudo de caso

4.1 Método e descrição dos testes

O método adotado durante o desenvolvimento deste estudo foi o da pesquisa experimental exploratória [Waz17], uma vez que nem todas as peculiaridades das ferramentas analisadas eram conhecidas. Foram desenvolvidos experimentos e coletas de informações comuns a todos os *scanners*, envolvendo métricas relevantes para o contexto de testes automatizados.

Para facilitar a replicabilidade dos testes e permitir que outros pesquisadores validem os resultados expostos na Seção 4.5, utilizou-se virtualização. Todo o ambiente foi virtualizado no software VirtualBox [Ora21] versão 6.1.20 r143896 (Qt5.12.8).

Como cada *scanner* testado possui múltiplas dependências, foi necessário adicionar repositórios extras e instalar bibliotecas por vezes desatualizadas ou descontinuadas. Nesse sentido, para facilitar a reprodução do cenário de testes, foi realizada a instalação e configuração de todas as ferramentas e suas dependências e, posteriormente, foi gerada uma imagem da máquina, disponível em: https://bit.ly/2TWWCbN, enquanto os manuais e *scripts* para realização dos testes localizam-se no diretório `/home/pentest/testes` do disco virtual da máquina.

Além disso, para que haja parâmetro de comparação, as máquinas utilizadas durante os testes possuíam as seguintes especificações:

- **Máquina física:** processador Inte Core i7-8750H, com 6 núcleos físicos e 12 *threads*, memória RAM 16 GB DDR4, modelo de notebook Dell G7-7588, sistema operacional Linux Ubuntu 20.04 LTS, software de virtualialização VirtualBox 6.1.10_Ubuntu r138449, com máquinas instanciadas no disco SSD com 560mbs de leitura e 530mbs de escrita;

- **Máquina virtual:** 1 núcleo de processador virtual, 3 GB de memória RAM dedicada, sistema operacional Kali Linux 2020.2 64-Bit e disco VDI de 54 GB.

Destaca-se que a escolha do sistema operacional Kali Linux para a máquina virtual se deu pela sua popularidade dentro da comunidade de testes de segurança automatizados, bem como por nativamente possuir diversas bibliotecas e repositórios exigidos pelos *scanners* testados.

Foi realizado neste estudo uma avaliação do módulo *crawler* de *scanners* (*open source*) de vulnerabilidades de aplicações Web de caixa preta. Não foram encontrados na literatura estudos que façam uma avaliação direcionada ao módulo *crawler* dessas ferramentas. Porém, há estudos que apontam este módulo como o que contém as principais limitações nos *scanners* [DTPP18, FBJ+16,

AAA+17]. Portanto, foi feito neste trabalho um estudo exclusivo e aprofundado do *crawler* para entender as principais limitações e o motivo de apresentarem baixo desempenho.

Todo o ambiente de teste foi montado em uma única máquina virtual. Os sistemas Web testados (WackoPicko e OWASP Juice Shop) foram instalados e configurados em *localhost*, por meio de contêineres do *Docker* [Doc21] disponibilizados pelos seus autores.

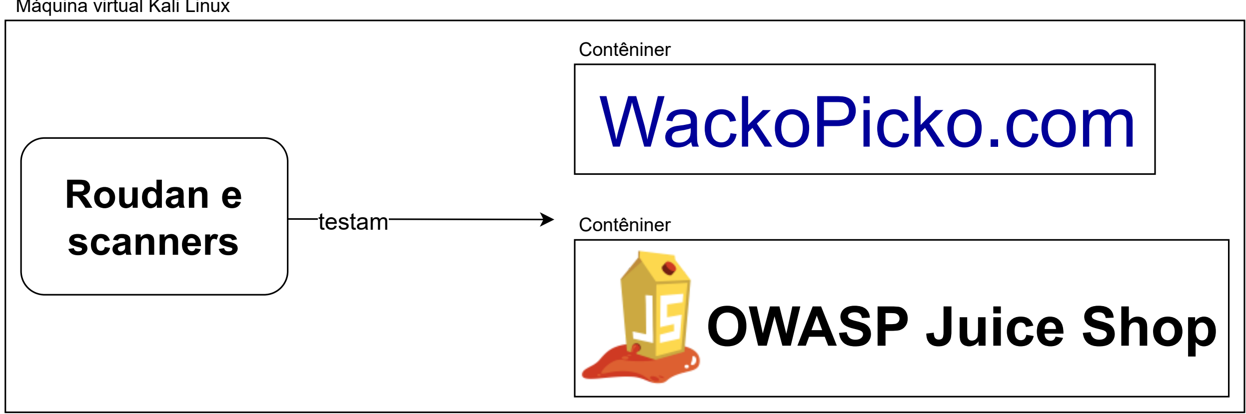

Figura 4.1: *Infraestrutura de testes.*

A Figura 4.1 mostra um desenho do ambiente de teste montado. Os *scanners* testados e o *crawler* desenvolvido estão no mesmo ambiente para que os resultados não beneficiem nenhuma ferramenta testada por conter configurações de infraestrutura diferentes.

Para analisar as requisições geradas pelas ferramentas aos servidores Web, foi utilizado o recurso *proxy* da ferramenta *Burp Suite* [Por21]. Portanto, o fluxo das requisições HTTP começam no *scanner* ou *crawler* testado, mas é interceptado pelo *Burp Suite* que, em seguida, encaminha a requisição ao servidor Web.

4.2 *Scanners* de vulnerabilidades testados

Optou-se por utilizar somente *scanners* de caixa preta com código aberto, boa documentação, e relativa popularidade na comunidade técnico-científica. A categoria de ferramentas selecionadas foi a de *scanners* não desenvolvidos para um único tipo de vulnerabilidade, mas sim uma ferramenta mais abrangente capaz de testar diversas vulnerabilidades e independente da tecnologia do sistema Web a ser testado.

Após uma pesquisa na literatura, foram encontradas onze ferramentas com essas características. São elas: w3af, Grendel-Scan, Paros, OpenVAS, Arachni, Vega, soapUI, Wapiti, Ironwasp, SkipFish e Nikto. Algumas delas não foram incluídas neste trabalho, pois nos trabalhos relacionados elas já foram analisadas e demonstraram muitos falsos negativos e falsos positivos em seus resultados, como por exemplo, a soapUI e a Arachni [MS18].

Dessa forma, os testes deste trabalho restringiram-se ao módulo rastreador das seguintes ferramentas:

- Wapiti – 3.0.3 [Wap21]
- Paros – 3.2.13 [Par18]
- w3af – 2019.1.2 [w3a13]

- Subgraph Vega – 1.0 [Sub14]
- Nikto – 2.1.6 [CIR21]
- SkipFish – 2.10b [Too21]

Apesar dessas ferramentas permitirem configurações personalizadas e avançadas (*e.g.*, passar *cookies* válidos como parâmetro), é importante salientar que foi realizada pouca parametrização dos *scanners* e seus módulos de rasteamento. Os parâmetros padronizados foram: uma combinação de usuário e senha válida; configuração do *proxy* para que as requisições fossem interceptadas pelo *Burp Suite* para as ferramentas não possuidoras de *proxy* próprio; e restrição do escopo de vulnerabilidades buscadas no sistema. Como o objetivo desta pesquisa é avaliar apenas o módulo *crawler*, permitir as ferramentas buscarem todas as vulnerabilidades provocaria uma quantidade muito maior de requisições ao servidor Web, o que poderia comprometer o bom funcionamento da aplicação. As vulnerabilidades XSS e *SQL Injection* ficaram habilitadas para todos os *scanners*, permitindo que a ferramenta explorasse a vulnerabilidade de *SQL Injection* na página de *login* do sistema WackoPicko, o que permite autenticar no sistema sem a necessidade de um usuário e senha válidos. Julgou-se pertinente avaliar apenas a capacidade de cada *scanner* formular sua estratégia de ataque automatizado, ou seja, como cada um acessaria as aplicações Web, autenticaria e exploraria vulnerabilidades sem muitas configurações e interações manuais do testador.

Em relação às configurações das ferramentas, foi definida a URL de ataque `http://127.0.0.1` para o sistema WackoPicko e a URL `http://127.0.0.1:3000/#` para o sistema *Juice Shop* em todos os *scanners*. O caractere # na URL do *Juice Shop* é uma âncora para a página inicial e sem ela os *scanners* não acessam a aplicação corretamente.

As ferramentas Subgraph Vega e Paros apresentam o relatório em suas próprias interfaces gráficas e também possuem *proxies* próprios. A ferramenta Skipfish também possui *proxy* próprio, porém, o relatório foi configurado para gerar um arquivo HTML. Para os demais *scanners*, foi utilizado como *proxy* o Burp Suite Community 2020.4.1 para a realização de todos os testes, e solicitado relatório em HTML. As demais configurações padrões de cada *scanner* foram mantidas.

É importante observar que, como em qualquer experimento dessa natureza, os resultados podem divergir dependendo das especificações do cenário de testes (*i.e.*, versão do software, sistema operacional, *hardware* etc.). Durante os experimentos deste trabalho foram realizadas as repetições necessárias até que o desvio padrão dos resultados dos testes fosse menor que 1%. Quase todas as ferramentas não apresentaram divergências nos resultados em diferentes testes realizados.

4.3 Sistemas Web testados

Os sistemas Web testados foram o WackoPicko [DCV19] e o OWASP *Juice Shop* [OWA21a]. Os sistemas foram desenvolvidos com tecnologias distintas e utilizá-los foi importante para possibilitar a avaliação da capacidade das ferramentas escanearem em diferentes tecnologias. A aplicação WackoPicko foi desenvolvida na linguagem PHP [Gro21], enquanto a OWASP *Juice Shop* foi desenvolvida em Node.js [Fou21], *Express* [Exp21] e *Angular* [Ang21]. Os dois sistemas possuem vulnerabilidades conhecidas, colocadas de forma proposital e documentadas. Essa característica em uma aplicação em teste é importante para medir a quantidade de falsos negativos do *scanner*.

O aplicativo WackoPicko usa recursos comumente encontrados em aplicações Web que tornam seu rastreamento difícil, como formulários HTML complexos, código JavaScript e Flash, e páginas criadas dinamicamente [DCV10]. Os autores afirmam que as vulnerabilidades introduzidas no código-fonte do aplicativo são representativas das falhas comumente encontradas em aplicativos do mundo real.

Segundo a OWASP (2021), o projeto *Juice Shop* é provavelmente a aplicação Web insegura mais moderna e sofisticada que existe atualmente. Para a OWASP, essa aplicação pode ser usada em treinamentos de segurança, demonstrações de conscientização, competições e como uma cobaia para ferramentas de segurança [OWA21a]. A *Juice Shop* engloba vulnerabilidades de todo o OWASP *Top Ten* [**?**], com muitas outras falhas de segurança encontradas em aplicativos do mundo real.

Durante esta pesquisa, não foi encontrado nenhum trabalho que tenha feito a avaliação de *scanners* de vulnerabilidades de aplicações Web utilizando o sistema OWASP Juice Shop ou outro sistema com as mesmas tecnologias. A maioria dos trabalhos encontrados utilizam apenas sistemas desenvolvidos na linguagem PHP para testar as ferramentas, não avaliando o comportamento dos *scanners* em tecnologias diversas.

4.4 Métricas de desempenho adotadas

As principais métricas apresentadas por trabalhos anteriores são: a quantidade de falsos positivos e a quantidade de falsos negativos nas varreduras realizadas pelos *scanners* [AAA+17, DZ17]. Apesar de alguns trabalhos mensurarem a cobertura dos *crawlers* [AAA+17, DCV10], não foi encontrado nenhum que discutisse isoladamente as métricas de rastreamento obtidas. Isso motivou a criação das seguintes métricas específicas para o módulo rastreador dos *scanners*: renderiza código JavaScript; realiza varredura autenticada; páginas indexadas; páginas vulneráveis acessadas; gera páginas infinitamente; realiza varreduras repetitivas; e tenta logar com dicionário. A seguir, será detalhada cada métrica apresentada nesse parágrafo.

Renderiza código JavaScript indica se a ferramenta renderiza a página Web e todo o código JavaScript antes de realizar varreduras em busca de *links* e redirecionamentos. Para este trabalho, a métrica *renderiza código JavaScript* não é apenas capaz de interpretar esse tipo de código, mas também é capaz de obter todo o código HTML após a renderização completa da página a qual aparece para usuários comuns de uma aplicação Web utilizando um *browser*. Não renderizar código JavaScript é uma grande desvantagem para a ferramenta, pois muitas aplicações modernas não apresentam os *links* diretamente no código HTML, podendo conter códigos referenciados em arquivos externos ao HTML da página acessada.

Realiza varredura autenticada avalia se a ferramenta foi capaz de gerar uma autenticação bem sucedida no sistema e, em seguida, realiza varreduras e novas requisições ao servidor com a sessão autenticada. Quando o *scanner* ou *crawler* consegue realizar varreduras com sessão autenticada, obtém-se acesso a diversas funcionalidades da aplicação e considera-se, portanto, um benefício atender a essa métrica.

Páginas indexadas identifica a quantidade de páginas que a ferramenta indexou da página Web em teste, ou seja, apenas foi descoberto a existência da página Web no servidor.

Páginas HTTP200 indica a quantidade de páginas Web que foram acessadas pela ferramenta e obtiveram uma resposta de acesso bem sucedido do servidor. Ser capaz de indexar uma página não significa que a ferramenta consegue acessá-la, mas apenas sabe da existência do arquivo dentro do servidor Web. Neste experimento, considera-se como páginas HTTP200 quando a ferramenta consegue obter um código de resposta HTTP 200 [IET99] para a requisição feita ao servidor. Quanto menor for a quantidade em *páginas HTTP200*, pior é a eficácia da ferramenta, pois menos funcionalidades serão testadas.

Páginas vulneráveis !HTTP200 é a quantidade de páginas da aplicação Web em teste que contém vulnerabilidades, mas a ferramenta não foi capaz de acessá-las e obter uma resposta HTTP 200. Essa métrica permite mensurar a quantidade de falsos negativos reportados pela ferramenta em decorrência da ineficácia do módulo *crawler*, visto que o *scanner* não será capaz de realizar tentativas de ataque as páginas às quais não se obteve acesso. Quanto maior o valor para essa métrica, pior é a eficácia da ferramenta.

Identifica páginas dinâmicas significa que a ferramenta não sabe identificar páginas geradas dinamicamente e entra em *loop* infinito. As páginas geradas a partir de calendários, por exemplo, podem fazer uma ferramenta mal programada entrar em *loop*. Considera-se eficaz quando o módulo *crawler* consegue fazer tal identificação.

Evita varreduras repetitivas verifica se a ferramenta gera requisições repetidas para uma mesma URL, visto que as aplicações Web podem conter hiperlinks repetidos em várias partes do sistema. Se a ferramenta não verificar se a página já foi testada, pode acarretar em varreduras repetitivas.

Tenta logar com dicionário mede se a ferramenta utilizou algum dicionário de usuários e senhas para tentar autenticar nas páginas de *login*.

Ao analisar todas as métricas descritas anteriormente, é imprescindível a utilização de *proxy* para que seja possível verificar exatamente o tipo de requisição feita pelos *scanners*, quais os dados enviados na requisição e se esta foi bem sucedida. As métricas também permitem analisar a quantidade de falsos negativos nas varreduras por motivo de mal funcionamento do módulo rastreador.

4.5 Resultados e discussão dos experimentos

Conhecer as limitações de um *crawler* é importante para ponderar sobre a eficácia de um *scanner*, pois isso impacta principalmente na métrica de falsos negativos gerados pela ferramenta. Apesar deste trabalho não medir os falsos negativos dos *scanners*, mensurou-se a limitação que o módulo *crawler* provoca, mostrando que diversas páginas com vulnerabilidades não foram acessadas e escaneadas.

Nenhum dos *scanners* estudados renderizam o código JavaScript antes de realizar varreduras. Nos resultados discutidos na seção 4.5.2, são apresentados mais detalhes sobre a limitação das ferramentas quanto a essa métrica, visto que a aplicação *Juice Shop* é inteiramente desenvolvida com código JavaScript. A única ferramenta que renderizou adequadamente foi o *crawler* Roudan.

A Tabela 4.1 apresenta uma visão geral das métricas *renderiza código JavaScript* e *realiza varredura autenticada*. Apenas as ferramentas *Wapiti*, Paros e *Roudan* foram capazes de realizar

Tabela 4.1: *Limitações de renderização de JavaScript e varredura autenticada.*

	Wapiti	Paros	w3af	Vega	Nikto	SkipFish	Roudan
Renderiza código JavaScript	✗	✗	✗	✗	✗	✗	✓
Realiza varredura autenticada	✓	✓	✓	✗	✗	✗	✓

✓- Sim (renderiza código JavaScript ou realiza varredura autenticada);
✗ - Não (não renderiza código JavaScript ou não realiza varredura autenticada).

a autenticação no sistema e varreduras com uma sessão válida. Para o sistema WackoPicko, foi passado a combinação `bob%bob` como usuário e senha e para o sistema *Juice Shop*, foi passado a combinação `teste@teste.com%Teste@26`. Os dados de autenticação do primeiro sistema é uma combinação de usuário e senha já existentes assim que inicializa o contêiner da aplicação, já os dados de autenticação da segunda aplicação precisam ser cadastrados antes de iniciar a varredura. As ferramentas *Wapiti* e Paros foram capazes de realizar varreduras com sessão autenticada apenas no sistema WackoPicko, já no sistema *Juice Shop* apenas o Roudan conseguiu acessar a página de *login* para realizar a tentativa de autenticação. O *Roudan* foi capaz de realizar varreduras autenticadas nas duas aplicações.

Nos experimentos, foram ativados quase todos os módulos de *crawler* da ferramenta w3af, mas ela não conseguiu sequer indexar todas as páginas sem autenticação. Uma forma mais eficiente de se rastrear é utilizar bibliotecas padrões de processamento de páginas HTML para extrair hiperlinks em uma página da Web. O rastreador procura os atributos (como href, src e action) no conteúdo HTML e eventos JavaScript (como window.open, window.location, .load, .location.assign, .href, .action e .src) para identificar as URLs existentes em uma página da Web [DTK+18], depois de renderizar a página. Apesar das ferramentas *Subgraph* Vega e *Wapiti* utilizarem essa técnica de seguir os *links*, o *scanner* Vega não obteve bons resultados para páginas vulneráveis por não conseguir realizar requisições com autenticação. Já o *scanner* Wapiti conseguiu realizar requisições com autenticações bem sucedidas, mas conseguiu atingir apenas 17 páginas do sistema WackoPicko, enquanto o *Roudan* conseguiu atingir 48 páginas, conforme mostra a Tabela 4.2.

Sobre autenticação, é importante que o *crawler* dos *scanners* realize tentativas de *login* com base em um dicionário. A página de autenticação "`/admin/login.php`", possui um usuário `admin` com senha `admin`, porém, apesar de ser uma abordagem comum, nenhuma ferramenta realizou tentativa de autenticação com usuários e senhas comuns (`e.g.`, admin%admin). O sistema WackoPicko possui duas páginas de *login* distintas, uma para usuários comuns "`/users/login.php`" e outra para administradores do sistema "`/admin/login.php`". A combinação de usuário e senha passada como parâmetro a todas as ferramentas testadas é referente à página "`/users/login.php`". A ferramenta desenvolvida neste trabalho foi a única que conseguiu realizar a autenticação baseada em dicionário na página de administrador.

4.5.1 WackoPicko

As ferramentas com os melhores resultados foram as que buscaram por hiperlinks nas páginas acessadas no servidor. As ferramentas Nikto e *SkipFish* utilizam uma técnica baseada em *wordlist* de páginas e diretórios. A *wordlist* possui uma grande lista de páginas e diretórios, em que a ferramenta realiza requisições ao servidor Web e, ao receber um código HTTP 200, conclui-se que

Tabela 4.2: *Visão geral dos resultados obtidos nos testes realizados nessa pesquisa.*

	Wapiti	Paros	w3af	Vega	Nikto	SkipFish	Roudan
Páginas indexadas	17	19	28	28	7	8	48
Páginas HTTP200	17	19	15	8	6	8	25
Páginas vulneráveis !HTTP200	4	6	8	8	12[1]	9[1]	2
Identifica páginas dinâmicas	✓[2]	✓	✓	✓	✓	✓	✓
Evita testes repetitivos	✗	✓	✓	✓	✗	✗	✓
Tenta logar com dicionário	✗	✗	✗	✗	✗	✗	✓

✓- Sim (identifica páginas dinâmicas, evita testes repetitivos ou tenta logar com dicionário);
✗ - Não (não identifica páginas dinâmicas, não evita testes repetitivos ou não tenta logar com dicionário).

a página solicitada existe no servidor. Pode-se observar na Tabela 4.2 que essa técnica é a menos eficiente. Apesar dos *scanners* Nikto e *SkipFish* não terem conseguido rastrear uma quantidade significativa de páginas, eles conseguiram descobrir páginas Web que as outras ferramentas não conseguiram encontrar, conforme pode ser observado na Tabela 4.3. Essas ferramentas foram as únicas a encontrarem a página "`/about.php`", enquanto que a página "`/test.php`" foi acessada apenas pelo Nikto. O *scanner* que menos conseguiu acessar páginas foi o do Nikto, enquanto o *Roudan* foi o *crawler* que acessou o maior número de páginas.

Quanto à métrica *Páginas vulneráveis não acessadas*, a ferramenta com o melhor desempenho foi a Roudan, uma vez que possui o menor número de páginas vulneráveis não acessadas. Essas páginas foram as "`/pictures/highquality.php`" e "`/submitname.php`" que, conforme pode ser observado na Tabela 4.4, não foram acessadas por nenhuma ferramenta. Apesar dos *scanners* w3af e *Subgraph* Vega terem conseguido acessar uma quantidade diferente de páginas, ambos não conseguiram acessar oito páginas vulneráveis. A ferramenta com menor desempenho para essa métrica foi a Nikto, com doze páginas vulneráveis não acessadas. Com base nesses resultados, é possível mensurar a quantidade de falsos negativos gerados em decorrência de limitações do módulo rastreador.

O Roudan e os *scanners* w3af e Nikto foram capazes de indexar uma quantidade de páginas Web superior à quantidade de páginas com resposta HTTP 200 por utilizarem a técnica de fazer uma requisição HTTP nos diretórios descobertos, obtendo o nome de todos os arquivos disponíveis no diretório acessado. Por isso, essas ferramentas conseguiram indexar uma quantidade significativa de páginas Web, mas por não fazerem uma requisição autenticada, ou por não preencher formulários disponíveis adequadamente, não obtiveram uma resposta HTTP 200 em todas as páginas descobertas.

O Paros é um *proxy* e também *scanner* de vulnerabilidade. Para essa ferramenta, é necessária inicialmente a navegação manual utilizando o *browser*, para que seja detectado o sistema a ser testado. Ao fazer a requisição ao sistema, o Paros intercepta a requisição do servidor e guarda a URL. No Paros é necessário realizar a autenticação manualmente usando o *browser*. Após a autenticação, a ferramenta guarda a sessão autenticada e o testador pode iniciar a varredura.

Nenhuma das ferramentas em análise geraram páginas infinitamente, porém, o *Wapiti* gerou 37 páginas baseadas em calendários, enquanto a média de outras ferramentas foi de 8 páginas. Se o

[1] Os scanners Nikto e SkipFish foram as únicas ferramentas com a quantidade de Páginas vulneráveis !HTTP200 maior que a quantidade de Páginas indexadas e Páginas HTTP200. A explicação dessa ocorrência encontra-se no texto.

[2] Mas entrou em *loop* e gerou um grande número de páginas dinamicamente.

Tabela 4.3: *Páginas não vulneráveis com resposta HTTP 200 de cada ferramenta.*

	Wapiti	Paros	w3af	Vega	Nikto	SkipFish	Roudan
/about.php	✗	✗	✗	✗	✓	✓	✗
/admin/index.php?page=create	✗	✗	✗	✗	✗	✗	✓
/admin/index.php?page=home	✗	✗	✗	✗	✗	✗	✓
/calendar.php?date=(date)	✓	✓	✓	✓	✓	✗	✓
/cart/action.php?action=add	✗	✗	✗	✗	✗	✗	✓
/cart/action.php?action=delete	✓	✗	✗	✗	✗	✗	✓
/cart/add_coupon.php	✗	✗	✓	✓	✗	✗	✗
/comments/preview_comment.php	✗	✗	✗	✗	✗	✗	✓
/index.php	✓	✓	✓	✓	✓	✓	✓
/pictures/purchased.php	✗	✓	✗	✗	✗	✗	✓
/pictures/recent.php	✓	✓	✓	✗	✗	✗	✓
/test.php	✗	✗	✗	✗	✓	✗	✗
/tos.php	✓	✓	✓	✗	✗	✓	✓
/users/home.php	✓	✓	✗	✗	✗	✗	✓
/users/logout.php	✓	✓	✗	✗	✗	✗	✓
/users/register.php	✓	✓	✓	✗	✗	✓	✓
/users/view.php?userid=(id)	✗	✓	✗	✗	✗	✗	✓

✓- Página acessada com resposta HTTP 200;
✗ - Página não acessada (resposta HTTP diferente de 200).

Tabela 4.4: *Páginas vulneráveis com resposta HTTP 200 de cada ferramenta.*

	Wapiti	Paros	w3af	Vega	Nikto	SkipFish	Roudan
/admin/index.php?page=login	✓	✓	✓	✓	✓	✓	✓
/cart/review.php	✓	✓	✗	✗	✗	✗	✓
/guestbook.php	✓	✓	✓	✓	✗	✓	✓
/passcheck.php	✓	✓	✗	✓	✗	✗	✓
/piccheck.php	✗	✗	✗	✗	✗	✗	✓
/pictures/highquality.php	✗	✗	✗	✗	✗	✗	✗
/pictures/search.php?query=bla	✓	✓	✓	✓	✗	✓	✓
/pictures/upload.php	✓	✓	✗	✗	✗	✗	✓
/pictures/view.php?picid=(id)	✓	✓	✗	✗	✗	✗	✓
/submitname.php	✗	✗	✗	✗	✗	✗	✗
/users/login.php	✓	✓	✓	✓	✗	✓	✓
/users/sample.php?userid=(id)	✓	✓	✓	✗	✗	✗	✓
/users/similar.php	✗	✓	✗	✗	✗	✗	✓

✓- Página acessada com resposta HTTP 200;
✗ - Página não acessada (resposta HTTP diferente de 200).

crawler de uma ferramenta não prever a geração de páginas dinâmicas, o *scanner* pode entrar em um *loop* infinito.

Na Tabela 4.3, foram colocados apenas os *links* que foram acessados por pelo menos uma das ferramentas testadas com resposta HTTP 200. Os *links* `/about.php` e `/test.php` só podem ser acessados por ferramentas que façam requisições ao servidor usando uma *wordlist* de nomes de arquivos e diretórios, uma técnica não utilizada na maioria dos *scanners* e nem no *crawler* Roudan. Já a Tabela 4.4 possui todas as páginas com vulnerabilidade documentada em [DCV19].

A Tabela 4.5 apresenta o porcentual de falsos negativos apresentados por cada ferramenta em decorrência do mal funcionamento do módulo rastreador. Para realizar esse cálculo, primeiro

Tabela 4.5: *Porcentagem de falsos negativos em decorrência do mal funcionamento do módulo crawler.*

Wapiti	Paros	w3af	Vega	Nikto	SkipFish	Roudan
26.66 %	20 %	53.33 %	53.33 %	80 %	60 %	13.33 %

observou-se a quantidade de vulnerabilidades existentes no WackoPicko [DCV19] e em quais páginas estavam tais vulnerabilidades. Como todas as ferramentas testadas foram capazes de acessar as páginas de *login*, que são as únicas com mais de uma vulnerabilidade, a cada página não acessada (quantitativo mostrado na Tabela 4.2) é uma vulnerabilidade não encontrada em decorrência do não acesso pela ferramenta. A aplicação possui 15 vulnerabilidades documentadas, que corresponde a 100% das vulnerabilidades.

4.5.2 OWASP *Juice Shop*

O desempenho das ferramentas para o sistema *Juice Shop* não será apresentado com todas as tabelas e dados apresentados no sistema WackoPicko, pois os resultados obtidos não foram suficientes para uma análise semelhante. A eficácia das ferramentas foi baixa, dado que não conseguiram acessar nenhuma página do servidor, além da URL de entrada. Primeiro, é necessário entender o funcionamento da aplicação para, em seguida, compreender o resultado das ferramentas. A *Roudan* foi a única ferramenta capaz de autenticar no *Juice Shop* e realizar varreduras com sessão válida.

Para todas as ferramentas testadas, foi passada a URL "`http://127.0.0.1:3000/#`". O caractere "#" é indispensável na URL para acessar a aplicação corretamente, pois é uma âncora para a página inicial da aplicação. Também houve a tentativa de passar a URL sem a âncora, porém, as ferramentas obtiveram os mesmos resultados. Algumas ferramentas removem a âncora antes de realizar a requisição ao servidor Web e não fazem o redirecionamento automático, tal como é feito nos *browsers*, para a página inicial. Porém, a principal limitação, que foi o motivo da maioria das ferramentas não conseguirem rastrear adequadamente o sistema, foi não renderizarem adequadamente o código JavaScript. Ao analisar o código HTML 4.1, é possível entender o problema enfrentado pelas ferramentas.

Listing 4.1: *Código HTML da página inicial da aplição OWASP Juice Shop.*

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>OWASP Juice Shop</title>
6   <meta name="description" content="Probably the most modern and sophisticated
        insecure Web application">
7   <meta name="viewport" content="width=device-width, initial-scale=1">
8   <link id="favicon" rel="icon" type="image/x-icon" href="assets/public/
        favicon_js.ico">
9   <link rel="stylesheet" type="text/css" href="//cdnjs.cloudflare.com/ajax/libs/
        cookieconsent2/3.1.0/cookieconsent.min.css"/>
10  <script src="//cdnjs.cloudflare.com/ajax/libs/cookieconsent2/3.1.0/
        cookieconsent.min.js"></script>
11  <script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.2.4/jquery.min.js">
        </script>
12  <script>
13    window.addEventListener("load", function(){
```

```
14        window.cookieconsent.initialise({
15          "palette": {
16            "popup": { "background": "#546e7a", "text": "#ffffff" },
17            "button": { "background": "#558b2f", "text": "#ffffff" }
18          },
19          "theme": "classic",
20          "position": "bottom-right",
21          "content": { "message": "This Website uses fruit cookies to ensure you
              get the juiciest tracking experience.", "dismiss": "Me want it!", "
              link": "But me wait!", "href": "https://www.youtube.com/watch?
              v=9PnbKL3wuH4" }
22        })});
23    </script>
24  <link rel="stylesheet" href="styles.css"></head>
25  <body class="mat-app-background bluegrey-lightgreen-theme">
26    <app-root></app-root>
27  <script src="runtime-es2018.js" type="module"></script><script src="
      runtime-es5.js" nomodule defer></script><script src="polyfills-es5.js"
      nomodule defer></script><script src="polyfills-es2018.js" type="module">
      </script><script src="vendor-es2018.js" type="module"></script><script src="
      vendor-es5.js" nomodule defer></script><script src="main-es2018.js" type="
      module"></script><script src="main-es5.js" nomodule defer></script></body>
28  </html>
```

Esse é o código da página inicial ao acessar a aplicação. Na linha 27 do código, nota-se que a *tag* `<script>` faz referência a vários arquivos com extensão `.js`, que são arquivos com códigos JavaScript. Algumas ferramentas são capazes de detectar e até fazem a indexação desses arquivos JavaScript. Porém, o código mostrado acima é simples e não condiz com a aplicação acessada. Antes de buscar hiperlinks na página, a ferramenta precisa acessar todos os códigos JavaScript referenciados na linha 27 e renderizá-los dentro da própria página para que as funcionalidades do sistema apareçam. Esse método de acesso é feito pelos *browsers* e também pelo *Roudan*. A biblioteca *selenium* permite que o *Crawler* acesse a página Web por meio do *browser* Mozilla *Firefox* e faça toda a renderização adequada.

O Roudan conseguiu acessar os *links* `/about`, `/basket`, `/chatbot`, `/complain`, `/contact`, `/deluxe-membership`, `/forgot-password`, `/login`, `/photo-wall`, `/register` e `/search`, totalizando onze links acessados. Já as demais ferramentas conseguiram acessar apenas a página inicial, que foi a URL passada como parâmetro. O Paros detectou diversos arquivos e diretórios no sistema, mas as páginas indexadas foram somente as acessadas manualmente.

4.6 Considerações

Com base nos resultados apresentados nessa seção, deve-se atentar para o módulo rastreador dos *scanners*, pois este tem sido um dos principais limitadores na apresentação de falsos negativos. Para que um *crawler* apresente boa cobertura, é indispensável que seja feita uma renderização correta das páginas Web, acessando assim funcionalidades ocultas do sistema em teste. O Roudan, ferramenta proposta nesta pesquisa, apresentou a melhor eficácia na cobertura dos sistemas testados e pode melhorar a detecção de vulnerabilidades em *scanners* de vulnerabilidade de aplicações Web.

Capítulo 5

Trabalhos relacionados

Os trabalhos apresentados nesse capítulo ajudam a entender o estado da arte dos *scanners* de vulnerabilidade de caixa preta e também as contribuições científicas desta pesquisa. Entende-se como trabalhos relacionados aqueles que apresentam um novo *scanner* de vulnerabilidade Web de caixa preta ou que realizam um comparativo de *scanners* existentes. Não há trabalhos anteriores direcionados ao módulo *crawler* dessas ferramentas, ademais, serão considerados trabalhos relacionados apenas os apresentados posteriormente a 2009, pois sistemas Web e ferramentas tiveram diversas atualizações e mudanças tecnológicas nos últimos anos.

No trabalho [DCV10], os autores selecionaram onze *scanners* de vulnerabilidades de caixa preta para aplicações Web, contendo ferramentas de código aberto e também ferramentas proprietárias com custo de até milhares de dólares. Os *scanners* foram Acunetix, AppScan, Burp, Grendel-Scan, Hailstorm, Milescan, N-Stalker, NTOSpider, Paros, w3af e Webinspect. O teste foi realizado sobre um sistema real desenvolvido pelos próprios autores, no qual foram colocadas, propositalmente, vulnerabilidades conhecidas para avaliar melhor a capacidade de detecção dos *scanners*. Portanto, a utilização de apenas 1 sistema Web para os testes não avalia o comportamento das ferramentas em diferentes tecnologias, o que pode gerar resultados não precisos. Diferentemente, a presente pesquisa testa as ferramentas utilizando dois sistemas com tecnologias diferentes.

Doupé, Cova e Vigna (2010) identificaram diversos desafios que os *scanners* precisam superar para testar com sucesso aplicações Web modernas, tanto em termos de rastreamento quanto em capacidade de análise de ataque. Segundo os resultados apresentados pelos autores, todas as onze ferramentas deixaram de detectar diversas vulnerabilidades implantadas, mostrando que nenhuma das ferramentas, até mesmo as pagas, foi considerada eficiente e eficaz. Quase todas às ferramentas apresentaram muitos falsos positivos, que se repetiram quando algumas delas foram testadas neste trabalho. Nota-se que, desde 2010, as ferramentas Paros e w3af apresentam quantidade significativa de falsos negativos, então, acredita-se que a utilização do Roudan como *crawler* da ferramenta diminuiria a quantidade de falsos negativos, pois o Rondan consegue atingir uma maior quantidade de páginas vulneráveis.

Joshi e Kumar (2016) fizeram uma avaliação da eficácia das ferramentas Netsparker, Acunetix e Burp Suite na detecção dos principais tipos de vulnerabilidades relatadas pela OWASP. Para isso, foi desenvolvido um sistema Web em PHP, implantando tais vulnerabilidades e, em seguida, a realização de testes de segurança com os *scanners* de vulnerabilidade. Após os testes de segurança, foi feita uma comparação dos resultados de cada ferramenta [JK16]. Os autores se limitaram a analisar quatro das principais vulnerabilidades e apesar dessa restrição de escopo, relaram que todas as ferramentas

deixaram de detectar algumas vulnerabilidades existentes no sistema em teste. Segundo Joshi e Kumar (2016), o rastreamento foi significativamente melhorado, mas ainda existem limitações que afetam a taxa de detecção de vulnerabilidades, ou seja, os autores também identificaram defeitos no módulo *crawler*. Os autores desse trabalho não especificaram adequadamente quais critérios utilizaram na seleção dos quatro melhores *scanners* para os testes, nem foi especificado qual o critério de seleção das duas ferramentas selecionadas para realizar o estudo de caso. Por exemplo, foi escolhida a versão mais antiga do *scanner* Arachni, passível de gerar resultados negativos para a ferramenta, pois poderia haver limitações já corrigidas na nova versão.

Em [DTPP18], foi desenvolvido um protótipo chamado DetLogic, com o objetivo de detectar diferentes tipos de vulnerabilidades lógicas em aplicativos da Web em uma abordagem de caixa preta. O DetLogic age como um *proxy* que intercepta as requisições HTTP feitas ao servidor Web. Os autores classificam as falhas de segurança de aplicativos Web em falha de injeção e falha lógica, ambas podem levar um sistema Web a diversas consequências indesejáveis. Para Deepa *et al.* (2018), por já existir diversos aplicativos que verificam a falha de injeção, os atacantes têm priorizado falhas lógicas. Por isso, a proposta da ferramenta DetLogic é ser específica para falhas lógicas, e foi desenvolvida para detectar três diferentes tipos de vulnerabilidades lógicas: manipulação de parâmetros, controle de acesso e vulnerabilidades de desvio de fluxo de trabalho. A DetLogic tenta extrair o comportamento pretendido do aplicativo, sendo essa a principal dificuldade enfrentada, e, posteriormente, tenta burlar esse comportamento detectado. A ferramenta não é totalmente automatizada, pois os fluxos de trabalho de negócios são deduzidos por meio de uma navegação manual. Já o Roudan apresenta uma proposta totalmente automatizada para realizar o rastreamento das páginas Web, dispensando a navegação manual, com exceção da criação de usuários válidos. O DetLogic não possui suporte a algumas tecnologias, como por exemplo JQuery e VBScript, o que limita a quantidade de sistemas que podem utilizar a ferramenta proposta pelos autores, enquanto o Roudan independe da tecnologia utilizada no desenvolvimento.

No trabalho [LX11], os autores apresentaram a primeira ferramenta de caixa preta para detecção de ataques de violação de estados. O Block é uma ferramenta desenvolvida sobre o *proxy* WebScarab e tem como proposta ser um intermediador das requisições HTTP entre a aplicação Web e o usuário, o que é um pouco diferente da maioria dos *scanners*, porém, também é capaz de detectar vulnerabilidades Web. A ferramenta analisa o comportamento legítimo de requisições e respostas entre usuários e aplicativos Web, percebendo as invariantes do sistema Web. A violação dessas invariantes são identificadas como possíveis ataques de violação de estados. O Block foi uma grande inovação, pois independe do código fonte da aplicação a ser testada e foi a primeira abordagem de caixa preta nesse modelo integrado ao proxy. Para a realização de testes, os autores utilizaram sistemas reais de código aberto com falhas de segurança conhecidas. O comportamento desses sistemas foram analisados para identificar as sequências de solicitação e resposta e seus valores de variáveis de sessão associados durante sua execução sem ataques. A partir da observação dessas interações entre os clientes e o aplicativo, é obtido o conjunto de invariantes e seus valores de variáveis de sessão associados. O Block identifica possíveis violações de estado quando qualquer solicitação ou resposta da Web viole as invariantes associadas, como por exemplo, o tempo de execução das solicitações e respostas geradas. Apesar da contribuição, os autores analisaram sistemas Web com um número restrito de tecnologias, como, por exemplo, apenas sistemas desenvolvidos na linguagem PHP. A ferramenta não é capaz de manipular os ataques que violam os estados persistentes presentes nas

tabelas do banco de dados, além disso, ela não garante a exatidão das invariantes inferidas. Ademais, a utilização da ferramenta aumenta consideravelmente o tempo de resposta da aplicação Web.

Alsaleh *et al.* (2017) selecionaram quatro *scanners* de vulnerabilidade Web de código aberto para avaliar seu desempenho em termos de velocidade de execução, cobertura do rastreador, precisão de detecção e recursos extras suportados. Os *scanners* selecionados foram duas versões do *scanner* Arachni, o Wapiti e o Skipfish [AAA+17]. Esse artigo é o que mais se aproxima da proposta desta pesquisa, pois estudou apenas *scanners* de vulnerabilidade Web de código aberto, embora os autores não tenham especificado adequadamente como eles selecionaram estas quatro ferramentas. Por exemplo, a ferramenta W3af foi o *scanner* de vulnerabilidade Web de código aberto com os melhores resultados em [DCV10] e não houve justificativa da não utilização dessa ferramenta no estudo, enquanto que esta pesquisa inclui a ferramenta w3af nos experimentos realizados. Ainda, os resultados dos autores apontam que o desempenho do módulo rastreador é insatisfatório, visto que o *crawler* dos *scanners* não detectou algumas funcionalidades e páginas Web, impossibilitando que fosse realizada a varredura em busca de vulnerabilidades.

Salas e Martins *et al.* (2015) utilizaram o *scanner* de vulnerabilidades soapUI para simular ataques e automatizar o envio de requisições aos serviços Web. Os autores selecionaram 69 sistemas Web para simular ataques XSS, Fuzzing, vulnerabilidades XML mal formatado, SQL Injection e XPath Injection para serem testados pelo *scanner*. Segundo a pesquisa, o *scanner* soapUI reportou que 92.75% dos sistemas testados possuíam vulnerabilidades, já os pesquisadores apontam que 97,1% dos sistemas Web pesquisados possuem pelo menos uma vulnerabilidade, portanto, os dados reportados pela soapUI não são confiáveis pois a ferramenta apresentou diversos falsos positivos e falsos negativos quando testada [SM15].

No trabalho [JK16], foram selecionados os *scanners* de vulnerabilidade Netsparker, Acunetix e Burp Suite para analisar a eficácia na detecção dos principais tipos de vulnerabilidades relatadas pela OWASP. Foi desenvolvido apenas um sistema Web em PHP, possuindo uma grande limitação em casos de testes, o que pode gerar resultados não precisos. Já nos experimentos realizados nesta pesquisa, foram utilizados 2 sistemas Web, sendo um desenvolvido em PHP e outro em JavaScript, com quantidade de casos de teste superior.

Liu *et al.* (2015) utilizam um método de detecção dinâmico baseado na simulação do comportamento dos navegadores. A ferramenta desenvolvida objetiva encontrar pontos de injeção XSS ocultos, expandindo a cobertura de detecção e é orientada a interpretar o código JavaScript para encontrar os pontos de injeção escondidos nas páginas através de um navegador que atua como um *crawler* [LZWF15]. Segundo o estudo [RTFB20], a desvantagem dessa proposta é usar uma estrutura para lançar ataques em páginas com vulnerabilidades pré-estabelecidas.

Integrar o Roudan aos *scanners* testados nesta pesquisa ou aos *scanners* elencados nesta seção pode ser uma tarefa difícil, pois a maioria deles utilizam bibliotecas desatualizadas que podem conflitar com as bibliotecas utilizadas pelo Roudan. Por esse motivo, o Roudan não foi integrado a outros *scanners* nesta pesquisa.

5.1 Considerações

Todos os trabalhos estudados nesta pesquisa apresentam diversas limitações das ferramentas usadas ou propostas, sendo os falsos positivos e falsos negativos as principais. Com a utilização do

Roudan como *crawler* dos *scanners*, a quantidade de falsos negativos pode ser reduzida consideravelmente, visto que o Roudan apresentou um desempenho superior a todas as outras ferramentas na cobertura do *crawler* dos sistemas Web testados.

Capítulo 6

Conclusões

A primeira contribuição deste trabalho foi realizar experimentos com *scanners* de vulnerabilidades de aplicações Web de código aberto para identificar as principais limitações do módulo *crawler*. Este estudo aponta que a não renderização de códigos JavaScript de forma correta compromete significativamente a rastreabilidade dos *scanners*. É necessário que um *crawler* renderize toda a página antes de realizar uma busca por *links*, redirecionamentos e *inputs* do sistema.

Os *scanners* de vulnerabilidades Web são comercializados muitas vezes como aplicativos *point-and-click*, porém, com base nos resultados apresentados em [DCV10], e também neste trabalho, nota-se que os *scanners* de caixa preta estão longe dessa realidade. Doupé *et al.* (2010) relataram que, mesmo com configurações manuais, as ferramentas testadas não foram capazes de encontrar diversas vulnerabilidades. Para um *scanner* de vulnerabilidade ser eficaz, ele precisa apresentar resultados confiáveis, caso contrário, demandaria muito tempo do testador procurar por falsos negativos e falsos positivos.

O Roudan, outra contribuição desse trabalho, é um *crawler* com cobertura superior às demais ferramentas testadas, portanto, é provável que, ao adotar o Roudan como *crawler* desses *scanners*, a quantidade de falsos negativos diminuiria significativamente, pois a ferramenta seria capaz de testar mais funcionalidades do sistema em teste. Além disso, não conseguir fazer autenticação no sistema sob ataque implica as ferramentas não alcançarem algumas páginas vulneráveis, o que gera resultados insatisfatórios. Conclui-se que técnicas mais sofisticadas de ataques de autenticação (*e.g.*, ataques baseados em dicionários) poderiam melhorar significativamente tal aspecto. O Roudan foi capaz de autenticar na página de administrador no sistema WackoPicko utilizando essa técnica.

Em trabalhos futuros, os experimentos podem ser expandidos a *scanners* proprietários para avaliar se as limitações apresentadas neste trabalho se aplicam somente às ferramentas *open source*. Pode-se também aprimorar o Roudan para aumentar a cobertura e a taxa de acesso às páginas Web com respostas HTTP 200. Para isso, a ferramenta precisa identificar quando há *captcha* e interagir com o testador para que o *captcha* seja resolvido.

Referências Bibliográficas

[AAA+17] Mansour Alsaleh, Noura Alomar, Monirah Alshreef, Abdulrahman Alarifi e AbdulMalik Al-Salman. Performance-based comparative assessment of open source web vulnerability scanners. *Security and Communication Networks*, 2017:1–14, 05 2017. 1, 12, 22, 24, 33

[AAZ+17] Abdulrahman Alzahrani, Ali Alqazzaz, Ye Zhu, Huirong Fu e Nabil Almashfi. Web Application Security Tools Analysis. Em *Proceedings - 3rd IEEE International Conference on Big Data Security on Cloud, BigDataSecurity 2017, 3rd IEEE International Conference on High Performance and Smart Computing, HPSC 2017 and 2nd IEEE International Conference on Intelligent Data and Securit*, 2017. 10

[Ang21] Angular. The modern web developer's platform. https://angular.io/, 2021. Último acesso em 1 de Agosto de 2021. 23

[BBGM10] Jason Bau, Elie Bursztein, Divij Gupta e John Mitchell. State of the art: Automated black-box web application vulnerability testing. Em *2010 IEEE symposium on security and privacy*, páginas 332–345. IEEE, 2010. 12

[Bea21] BeautifulSoup. Beautiful soup. https://www.crummy.com/software/BeautifulSoup/, 2021. Last accessed 28 July 2021. 15

[CIR21] CIRT.net. Nikto2. https://cirt.net/Nikto2, 2021. Último acesso em 31 de Julho de 2021. 23

[CVE22] CVE. Vulnerabilities by type. https://www.cvedetails.com/vulnerabilities-by-types.php, 2022. Last accessed 16 August 2022. 10, 11

[DCKV12] Adam Doupé, Ludovico Cavedon, Christopher Kruegel e Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. Em *21st USENIX Security Symposium (USENIX Security 12)*, páginas 523–538, Bellevue, WA, Agosto 2012. USENIX Association. 12

[DCV10] Adam Doupé, Marco Cova e Giovanni Vigna. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. Em *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010. 1, 11, 15, 24, 31, 33, 35

[DCV19] Adam Doupé, Marco Cova e Giovanni Vigna. Wackopicko. https://github.com/adamdoupe/WackoPicko, 2019. Último acesso em 2 de Agosto de 2021. 23, 28, 29

[dJJM15] Airton A de Jesus Junior[1] e Edward David Moreno[1]. Segurança em infraestrutura para internet das coisas. 2015. 12

[Doc21] Docker. Docker. https://www.docker.com/, 2021. Último acesso em 2 de Agosto de 2021. 22

[DTK+18] G. Deepa, P. Santhi Thilagam, Furqan Ahmed Khan, Amit Praseed, Alwyn R. Pais e Nushafreen Palsetia. Black-box detection of XQuery injection and parameter tampering vulnerabilities in web applications. *International Journal of Information Security*, 17, 2018. 1, 26

[DTPP18] G. Deepa, P. Santhi Thilagam, Amit Praseed e Alwyn R. Pais. DetLogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications*, 109:89–109, 2018. 1, 6, 12, 21, 32

[DZ17] Daniel Dalalana Bertoglio e Avelino Francisco Zorzo. Overview and open issues on penetration test. *Journal of the Brazilian Computer Society*, 2017. 1, 5, 6, 7, 12, 24

[ERRW18] Damiano Esposito, Marc Rennhard, Lukas Ruf e Arno Wagner. Exploiting the potential of web application vulnerability scanning. Em *ICIMP 2018 the Thirteenth International Conference on Internet Monitoring and Protection, Barcelona, Spain, 22-26 July 2018*, páginas 22–29. IARIA, 2018. 16

[ET21] Danilo Pereira Escudero e Routo Terada. Crawler desenvolvido. https://github.com/escuderoDP/crawler, 2021. Último acesso em 31 de Julho de 2021. 17

[Exp21] Express. Express framework web rápido, flexível e minimalista para node.js. https://expressjs.com/, 2021. Último acesso em 1 de Agosto de 2021. 23

[FBJ+16] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu e Alexander Pretschner. Security Testing: A Survey. *Advances in Computers*, 101(January 2018):1–51, 2016. xiii, 1, 5, 6, 7, 8, 12, 13, 21

[Fou21] OpenJS Foundation. Node.js. https://nodejs.org/, 2021. Último acesso em 1 de Agosto de 2021. 23

[Gro21] PHP Group. Php: Hypertext preprocessor. https://www.php.net/, 2021. Último acesso em 4 de Agosto de 2021. 23

[HCO11] William GJ Halfond, Shauvik Roy Choudhary e Alessandro Orso. Improving penetration testing through static and dynamic analysis. *Software Testing, Verification and Reliability*, 21(3):195–214, 2011. 11

[HL06] Michael Howard e Steve Lipner. *The Security Development Lifecycle*, volume 34. 06 2006. 9, 10

[IET99] IETF. Hypertext transfer protocol – http/1.1. https://www.rfc-editor.org/rfc/rfc2616.txt, 1999. Último acesso em 2 de Agosto de 2021. 25

[IET00] IETF. Rfc 2818. https://www.rfc-editor.org/rfc/rfc2818.html, 2000. Último acesso em 6 de Agosto de 2021. 5

[IPS] As tendências da rede do sistema de prevenção de intrusões. Em *2ª Conferência Internacional sobre Tecnologia Educacional e Computação de 2010*, volume 4. 16

[Jav21] JavaScript. Javascript. https://www.javascript.com/, 2021. Last accessed 28 July 2021. 15

[JK16] Chanchala Joshi e Umesh Kumar. Security Testing and Assessment of Vulnerability Scanners in Quest of Current Information Security Landscape. *International Journal of Computer Applications*, 2016. 1, 31, 33

[LX11] Xiaowei Li e Yuan Xue. BLOCK: A Black-bOx approach for detection of state violation attacks towards web applications. Em *ACM International Conference Proceeding Series*, 2011. 5, 11, 15, 32

[LZWF15] Yuan Liu, Wenbing Zhao, Dan Wang e Lihua Fu. A xss vulnerability detection approach based on simulating browser behavior. Em *2015 2nd International Conference on Information Science and Security (ICISS)*, páginas 1–4, 2015. 33

[MK15] Yuma Makino e Vitaly Klyuev. Evaluation of web vulnerability scanners. Em *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, páginas 399–402. IEEE, 2015. 11

[Moz21a] Mozilla. Firefox para computador. https://www.mozilla.org/pt-BR/firefox/new/, 2021. Last accessed 28 July 2021. 15

[Moz21b] Mozilla. geckodriver. https://github.com/mozilla/geckodriver/, 2021. Último acesso em 2 de Agosto de 2021. 17

[MS18] B. Mburano e W. Si. Evaluation of web vulnerability scanners based on owasp benchmark. Em *2018 26th International Conference on Systems Engineering (ICSEng)*, páginas 1–6, 2018. 22

[NCS17] Bharti Nagpal, Naresh Chauhan e Nanhay Singh. Secsix: security engine for csrf, sql injection and xss attacks. *International Journal of System Assurance Engineering and Management*, 8(2):631–644, 2017. 10

[Ora21] Oracle. Virtualbox. https://www.virtualbox.org/, 2021. Último acesso em 31 de Julho de 2021. 21

[OWA19a] OWASP. Owasp risk rating methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology, 2019. Last accessed 17 September 2019. 10

[OWA19b] OWASP. Owasp samm. https://owaspsamm.org/v2.0b/introduction/, 2019. Last accessed 16 October 2019. 9, 10

[OWA20] OWASP. Owasp testing guide v4.2. https://owasp.org/www-project-web-security-testing-guide/v42/, 2020. Last accessed 24 July 2021. xiii, 9

[OWA21a] OWASP. Owasp juice shop. https://owasp.org/www-project-juice-shop/, 2021. Último acesso em 1 de Agosto de 2021. 23, 24

[OWA21b] OWASP. Owasp top 10 - 2021. https://owasp.org/www-project-top-ten/, 2021. Acessado em 27 de Dezembro de 2021. 10

[Par18] Paros. Paros. http://www.parosproxy.org, 2018. Último acesso em 31 de Julho de 2021. 22

[PLP15] Stefan Prandl, Mihai Lazarescu e Duc-Son Pham. A study of web application firewall solutions. Em Sushil Jajoda e Chandan Mazumdar, editors, *Information Systems Security*, páginas 501–510. Springer International Publishing, 2015. 16

[Por21] PortSwigger. Burp suite. https://portswigger.net/burp, 2021. Último acesso em 2 de Agosto de 2021. 22

[PSM15] Marcelo Invert Palma Salas e Eliane Martins. A black-box approach to detect vulnerabilities in web services using penetration testing. *IEEE Latin America Transactions*, 13(3):707–712, 2015. 11

[PTE14] PTES. Penetration testing execution standard. http://www.pentest-standard.org/, 2014. Last accessed 24 July 2021. 9

[Pyt21] Python. Python. https://www.python.org/, 2021. Last accessed 28 July 2021. 15

[PZK16] Muhammad Parvez, Pavol Zavarsky e Nidal Khoury. Analysis of effectiveness of black-box web application scanners in detection of stored SQL injection and stored XSS vulnerabilities. Em *2015 10th International Conference for Internet Technology and Secured Transactions, ICITST 2015*, 2016. 1

[Rap21a] Rapid7. Metasploit-framework. https://github.com/rapid7/metasploit-framework/tree/master/data/wordlists, 2021. Last accessed 28 July 2021. 16

[Rap21b] Rapid7. Metasploit-framework. https://www.metasploit.com/, 2021. Last accessed 28 July 2021. 16

[RLHJ+99] Dave Raggett, Arnaud Le Hors, Ian Jacobs et al. Html 4.01 specification. *W3C recommendation*, 24, 1999. 15

[RTFB20] Germán E. Rodríguez, Jenny G. Torres, Pamela Flores e Diego E. Benavides. Cross-site scripting (xss) attacks and mitigation: A survey. *Computer Networks*, 166:106960, 2020. 1, 33

[sec] Secubat: A web vulnerability scanner. Association for Computing Machinery. 11

[Sel21] Selenium. Selenium. https://www.selenium.dev/, 2021. Last accessed 28 July 2021. 15

[Sha20] Mandar Prashant Shah. *Comparative Analysis of the Automated Penetration Testing Tools*. Tese de Doutorado, Dublin, National College of Ireland, 2020. 12

[SM15] Marcelo Invert Palma Salas e Eliane Martins. A Black-Box Approach to Detect Vulnerabilities in Web Services Using Penetration Testing. *IEEE Latin America Transactions*, 13(3), 2015. 33

[SS08] Murugiah P Souppaya e Karen A Scarfone. Technical guide to information security testing and assessment. Relatório técnico, 2008. xiii, 7, 8

[Sub14] Subgraph. Vega vulnerability scanner. https://subgraph.com/vega, 2014. Último acesso em 31 de Julho de 2021. 23

[Sut10] Larry Suto. Analyzing the accuracy and time costs of web application security scanners. *San Francisco, February*, 2010. 1

[Too21] Kali Tools. Skipfish package description. https://tools.kali.org/web-applications/skipfish, 2021. Último acesso em 31 de Julho de 2021. 23

[w3a13] w3af. w3af. http://w3af.org/, 2013. Último acesso em 31 de Julho de 2021. 22

[Wap21] Wapiti. Wapiti. https://wapiti.sourceforge.io, 2021. Último acesso em 31 de Julho de 2021. 22

[Waz17] Raul Sidnei Wazlawick. *Metodologia de pesquisa para ciência da computação*. Elsevier, 3 edição, 2017. 21

[xss] Uma abordagem de detecção de vulnerabilidade xss baseada na simulação do comportamento do navegador. Em *2015 2ª Conferência Internacional sobre Ciência da Informação e Segurança (ICISS)*. 12