

UNIVERSIDADE FEDERAL DO PAMPA

Isadora Garcia Ferrão

Análise *black-box* de ferramentas de
segurança na *Web*

Alegrete
2018

Isadora Garcia Ferrão

**Análise *black-box* de ferramentas de segurança na
Web**

Trabalho de Conclusão de Curso apresentado
ao Curso de Graduação em Ciência da Com-
putação da Universidade Federal do Pampa
como requisito parcial para a obtenção do tí-
tulo de Bacharel em Ciência da Computação.

Orientador: Prof. Me. Diego Luis Kreutz

Alegrete
2018

Isadora Garcia Ferrão


Análise *black-box* de ferramentas de segurança na Web

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

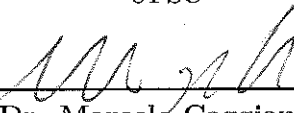
Trabalho de Conclusão de Curso defendido e aprovado em 07 de ...12... de ...18...
Banca examinadora:



Prof. Me. Diego Luis Kreutz
Orientador
UNIPAMPA



Prof. Dr. Douglas Dyllon Jeronimo de Macedo
UFSC



Prof. Dr. Marcelo Caggiani Luizelli
UNIPAMPA



Prof. Dr. Rodrigo Brandão Mansilha
UNIPAMPA



Sr. Thomás Jaskulski Capiotti
XLabs ON Security

Este trabalho é dedicado às mulheres que,
quando pequenas, sonharam em se tornar cientistas.

AGRADECIMENTOS

Um agradecimento muito especial para a minha melhor amiga, minha fonte inesgotável de força, amor e carinho, minha mãe. Mãe, pode ter certeza que sem ti nada teria sido possível. Tu é o meu maior exemplo de persistência, garra e determinação. Agradeço também aos demais familiares e amigos que torceram por mim durante esses anos de graduação.

Agradeço a minha dupla durante o período da faculdade, Sherlon Almeida, pelo companheirismo, coleguismo, amizade, força e por ter me feito acreditar no trabalho em grupo durante a graduação. Foram muitas noites acordados, mas estamos concluindo a faculdade, nós vencemos juntos!

Agradeço ao meu orientador, professor Diego Kreutz, pela amizade, pela paciência, pelos métodos diferenciados aplicados em sala de aula, pelas oportunidades, por ter acreditado que eu era capaz e também pelas orientações.

Por fim, gostaria de agradecer as Gurias na Computação pela força, incentivo, cuidado e por acreditarem em mim e nas outras meninas. Em especial, as professoras Aline Mello e Amanda Melo pelas oportunidades.

RESUMO

Vulnerabilidades em aplicações *Web* continuam sendo um problema comum e recorrente. Estatísticas recentes indicam que mesmo as vulnerabilidades antigas, como *SQL injection*, ainda representam um risco de segurança em grande parte das aplicações *Web*. Como forma de contribuir com o processo de detecção e diagnóstico de falhas de segurança, este trabalho apresenta uma análise de ferramentas de varredura (*scanners*) livres em um cenário controlado contendo as dez vulnerabilidades mais recorrentes em sistemas *Web*. Complementarmente, são avaliados e discutidos também o impacto de *frameworks* de desenvolvimento de *software* e *Web Applications Firewalls* (WAFs) na segurança de sistemas *Web*. Os resultados permitem identificar as principais diferenças entre *scanners* existentes, o impacto de cada *framework* de desenvolvimento e WAF na segurança de aplicações *Web*. Como exemplo, o *framework* de desenvolvimento Laravel mitiga de forma automática 60% das vulnerabilidades avaliadas, enquanto que o *framework* Codeigniter mitiga apenas 30%. Isto significa que a escolha do *framework* de desenvolvimento pode ter um impacto significativo na segurança do sistema.

Palavras-chave: Segurança na *Web*, *Scanners* de vulnerabilidades, *frameworks* de desenvolvimento *Web*, *Web Applications Firewalls* (WAFs).

ABSTRACT

Vulnerabilities in Web applications still are a common and frequent problem. Recent statistics indicate that even old vulnerabilities, such as SQL injection, still represent a security risk in most of Web applications. As a way to contribute to the process of detecting and diagnosing security flaws, this work presents an analysis of scanning tools free scanners in a controlled scenario containing the ten most frequent vulnerabilities in Web systems. Complementarily, the impact of Software Development Frameworks and Web Application Firewalls (WAFs) on the security of Web systems are evaluated and discussed. The results allow the identification of the main differences between existing scanners and the impact of each development framework and WAF on the security of Web applications. As an example, the development framework Laravel automatically mitigates 60 % of the vulnerabilities evaluated, while the Codeigniter framework mitigates only 30 %. This means that the choice of the development framework may have a significant impact on the system's security.

Key-words: *Web Security, Vulnerability Scanners, Web development frameworks, Web Applications Firewalls (WAFs).*

LISTA DE FIGURAS

Figura 1 – Etapas de desenvolvimento do trabalho	29
Figura 2 – Página com a vulnerabilidade de <i>cross-site scripting</i>	34
Figura 3 – Ambiente de testes do trabalho	35

LISTA DE TABELAS

Tabela 1 – Comparativo entre os papers/trabalhos	27
Tabela 2 – Vulnerabilidades que cada <i>scanner</i> se propõe a detectar	32
Tabela 3 – Detecção de vulnerabilidades no cenário controlado	37
Tabela 4 – Principais resultados e métricas no cenário controlado	39
Tabela 5 – Taxa de falsos-positivos no cenário controlado	40
Tabela 6 – Falhas na detecção de vulnerabilidades.	41
Tabela 7 – Vulnerabilidades detectadas nos cenários dos <i>frameworks</i>	42
Tabela 8 – Vulnerabilidades detectadas nos cenários dos <i>frameworks</i> e WAFs . . .	45

LISTA DE TRECHOS DE CÓDIGO

3.1	Exemplo de <i>SQL injection</i>	33
4.1	Regra de injeção de código do <i>ModSecurity</i>	46

LISTA DE SIGLAS

BWA Broken Web Applications

CSS Cascading Style Sheets

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

OWASP Open Web Application Security Project

PHP Hypertext Preprocessor

Unipampa Universidade Federal do Pampa

SUMÁRIO

1	INTRODUÇÃO	21
2	ESTADO DA ARTE	25
2.1	Cenários de testes	25
2.2	Tipos de testes	25
2.3	Trabalhos Relacionados	26
3	DESENVOLVIMENTO	29
3.1	Etapas de desenvolvimento	29
3.2	<i>Scanners</i> de vulnerabilidades <i>Web</i>	31
3.3	Implementação do cenário controlado	31
3.4	Ambiente de desenvolvimento e testes	34
4	RESULTADOS	37
4.1	Cenário Controlado	37
4.2	Cenário Controlado com Frameworks	41
4.3	Web Application Firewalls	44
5	DISCUSSÃO	47
6	CONSIDERAÇÕES FINAIS	49
	REFERÊNCIAS	51
	ANEXO A – CÓDIGOS DO CENÁRIO CONTROLADO . .	57

1 INTRODUÇÃO

A Internet faz parte do cotidiano da maioria das pessoas. Segundo estatísticas, a cada ano milhares de novos sistemas são implementados e hospedados na *Web* (BALDESSAR, 2014). A gama de serviços *online* não para de crescer, indo desde redes sociais, de uso geral, até sistemas mais complexos e críticos, como os sistemas produtivos, bancários e infraestruturas críticas (exemplo: distribuição de energia elétrica). Atualmente existem bancos 100% *online* (exemplos: N26/Alemanha, Revolut/Inglaterra, NuBank/-Brasil), isto é, onde o cliente não precisa, em nenhum momento, apresentar-se fisicamente a uma agência do banco. Aliás, o próprio conceito de “**agência bancária**” deixa de existir neste novo modelo bancário.

Apesar de a maioria desses sistemas *Web* possuírem impacto social ou econômico, um conjunto significativo deles são colocados em produção com vulnerabilidades críticas de segurança (JR et al., 2016). A principal causa desse problema pode ser explicada através do fato de os programadores se preocuparem essencialmente com as funcionalidades do produto final, pois possuem restrições de tempo para entrega do produto e não possuem os conhecimentos necessários sobre segurança da informação (COMPUTER-WORLD, 2017; VIEIRA; ANTUNES; MADEIRA, 2009). Isso leva a um cenário propício à exploração maliciosa de dados e componentes de sistemas (COELHO et al., 2015). Relatórios recentes reportam que o número de ataques e incidentes de segurança, muitos deles envolvendo vulnerabilidades antigas e recorrentes em sistemas *Web*, não para de crescer (Symantec, 2017; HOLÍK; NERADOVA, 2017; SCHWALENBERG, 2017). Além disso, alguns estudos mostram que, em alguns países, a porcentagem de aplicações *Web* vulneráveis pode chegar na casa dos 70% (ALAM et al., 2015), o que pode ser caracterizado como um cenário caótico, sob o ponto de vista da privacidade e da segurança dos dados dos usuários.

Com o rápido desenvolvimento de novas tecnologias, é eminente a necessidade da aplicação de boas práticas de segurança em sistemas *Web*, incluindo a detecção e o diagnóstico das vulnerabilidades presentes nos sites. Todo e qualquer ciclo de desenvolvimento de *software* deveria incluir uma fase de verificação e validação de aspectos de segurança do sistema. Um dos tipos de recursos mais frequentemente utilizados na prática para detectar e diagnosticar diferentes tipos e níveis de falhas de segurança em sistemas *Web* são os *scanners* de vulnerabilidades (FUNK; EPP et al., 2018; BERTOGLIO; ZORZO, 2017; FERRAO; KREUTZ, 2017; ALAM et al., 2015; MAKINO; KLYUEV, 2015; FONSECA et al., 2014; ROCHA; KREUTZ; TURCHETTI, 2012; BAU et al., 2010).

Os testes automatizados realizados por *scanners* de vulnerabilidades tornam possível e viável a verificação e correção periódica, durante todo o ciclo de vida do *software*, de falhas de segurança dos sistemas *Web*. A exemplo, universidades como a Universidade Federal do Pampa (Unipampa) utilizam *scanners* de vulnerabilidade na rede e nas aplicações *Web* para evitar ataques e comprometimento de sistemas através de vulnerabi-

lidades conhecidas e oriundas do processo de desenvolvimento de *software* (DTIC, 2018). Entretanto, como é apresentado e discutido neste trabalho, são necessárias múltiplas ferramentas de varredura e recursos de segurança, algo pouco implementado na prática, para atingir bons níveis de detecção e proteção contra exploração de vulnerabilidades comuns de sistemas *Web*. Uma política mais abrangente de utilização de ferramentas e mecanismos de segurança pode ajudar a reduzir significativamente a probabilidade de ocorrência de incidentes de segurança.

Existem diferentes estudos que analisam e/ou comparam *scanners* de vulnerabilidades voltados para a *Web* (FUNK; EPP et al., 2018; FERRAO; KREUTZ, 2017; HOLÍK; NERADOVA, 2017; PALSETIA et al., 2016; JABALLAH; KHEIR, 2016; GUPTA; GUPTA, 2015; ANTUNES; VIEIRA, 2015; NAGPAL et al., 2015; MAKINO; KLYUEV, 2015; ZHOU; EVANS, 2014; PELLEGRINO; BALZAROTTI, 2014; DOUPÉ et al., 2012; ROCHA; KREUTZ; TURCHETTI, 2012; BAU et al., 2010; DOUPÉ; COVA; VIGNA, 2010; VIEIRA; ANTUNES; MADEIRA, 2009; FONG et al., 2008; FONSECA; VIEIRA; MADEIRA, 2007). Entretanto, a maioria destas pesquisas analisa ferramentas comerciais, ou um subconjunto muito limitado de ferramentas gratuitas (exemplo: apenas uma ou duas ferramentas), ou são consideravelmente desatualizadas, ou investigam ferramentas específicas que detectam um único tipo ou pequenos subgrupos de vulnerabilidades presentes em sistemas *Web*. Além disso, nenhum desses trabalhos investiga o impacto dos principais *frameworks* de desenvolvimento PHP e WAFs livres disponíveis no mercado.

Em uma pesquisa recente, foram estudados e analisados os dez principais *scanners* de vulnerabilidades gratuitos (FERRAO; KREUTZ, 2017). Como ambiente de avaliação, os pesquisadores utilizaram uma máquina virtual da Open Web Application Security Project (OWASP), conhecida como Broken Web Applications (BWA). Esta máquina virtual BWA é constituída por um conjunto de aplicativos *Web* vulneráveis (em diferentes linguagens de programação, como PHP, Perl e Python), especialmente preparados para testes de segurança. Os resultados da pesquisa destacaram a eficácia, em termos de número de vulnerabilidades detectadas, dos *scanners* Zed Attack Proxy e Paros Proxy. Além disso, indicam que são necessários pelo menos quatro *scanners* para cobrir as vulnerabilidades encontradas na BWA. De forma similar, estudos recentes sobre *scanners* oferecidos como serviços online (SaaS) destacam a eficiência de serviços como o Acunetix, que detectou 50% das vulnerabilidades presentes no ambiente de testes (FERRAO et al., 2018). Entretanto, neste caso, mesmo a combinação de todos os *scanners* SaaS avaliados não é o suficiente para detectar todas as vulnerabilidades do sistema *Web*.

O maior desafio do trabalho (FERRAO; KREUTZ, 2017) foi comparar os *scanners* de vulnerabilidades igualmente, pois utiliza na avaliação um cenário com um número vasto de vulnerabilidades desconhecidas, a máquina virtual BWA. Esta MV consiste em um ambiente composto por dezenas de aplicações vulneráveis, criado especificamente para a formação de especialistas em segurança de sistemas. Na prática, é inviável tentar

enumerar as vulnerabilidades existentes nas centenas de milhares de linhas de código (em diferentes linguagens de programação) das aplicações *Web* da BWA.

O tipo de testes mais aplicado para *scanners* de vulnerabilidades é o *black-box* (HO-LÍK; NERADOVA, 2017; MAKINO; KLYUEV, 2015; ROCHA; KREUTZ; TURCHETTI, 2012; DOUPÉ; COVA; VIGNA, 2010; BAU et al., 2010), também utilizado neste trabalho. Os testes *black-box* são realizados de forma automatizada e sem acesso direto a detalhes de infraestrutura e codificação do sistema alvo.

Adotando uma abordagem diferente e ao mesmo tempo complementar em relação ao trabalho (FERRAO; KREUTZ, 2017), o objetivo principal deste trabalho é avaliar a eficácia de *scanners* de vulnerabilidades em ambientes controlados, o impacto de *frameworks* de desenvolvimento e *Web Applications Firewalls* (WAFs) na segurança de sistemas *Web*.

Os objetivos específicos deste trabalho podem ser sintetizados em:

- (o_1) identificar e selecionar *scanners* de vulnerabilidades gratuitos e livres;
- (o_2) identificar e selecionar as vulnerabilidades mais recorrentes em sistemas *Web*;
- (o_3) implementar um cenário controlado, contendo vulnerabilidades conhecidas;
- (o_4) avaliar *scanners* de vulnerabilidades;
- (o_5) avaliar o impacto de *frameworks* de desenvolvimento na segurança de sistemas *Web*;
- (o_6) avaliar o impacto de *Web Application Firewalls* (WAFs) na proteção de sistemas *Web*.

No que diz respeito à segurança, os *frameworks* utilizam arquivos de configuração com escopo limitado e bem definido, bibliotecas específicas, e mecanismos adicionais como filtros e validação automática de campos de formulários. Apesar destes recursos, cujo objetivo é simplificar o trabalho dos desenvolvedores e promover o desenvolvimento de sistemas mais seguros, há poucas evidências empíricas e analíticas sobre o impacto efetivo dos *frameworks* na segurança dos sistemas *Web*.

Os WAFs são sistemas que trabalham entre o servidor e o cliente filtrando dados de entrada e saída seguindo regras de segurança pré-definidas, cujo objetivo é mitigar ataques (HARVEY, 2018; BECHER, 2007). Apesar desses sistemas serem projetados para compor uma camada extra de segurança às aplicações *Web*, há poucas evidências empíricas sobre os efeitos práticos dos WAFs existentes e livremente disponíveis no mercado. No geral, a literatura existente concentra-se na proposição de WAFs com recursos específicos (FUNK; EPP et al., 2018; APPELT et al., 2018; FRUTH et al., 2017), deixando a desejar no que se refere a uma comparação prática, em cenários reais, entre diferentes *Web Application Firewalls*.

As contribuições deste trabalho podem ser resumidas em:

- (c_1) identificação e seleção dos dez principais *scanners* de vulnerabilidades gratuitos disponíveis no mercado, incluindo Uniscan (ROCHA; KREUTZ; TURCHETTI, 2012), Paros Proxy (SHI; CHEN; YU, 2010), Zed Attack proxy (INFOSEC, 2017), Nessus (BOY; ARANHA, 2016), Andiparos (SHI; CHEN; YU, 2010), Grabber (MACHADO et al., 2016), Wapiti (WILMSHURST; FRYXELL; HUDSONB, 1995), Ratproxy (TEODORO; SERRÃO, 2011), Skipfish (CARVALHO et al., 2013) e Vega (MUNIZ, 2013);
- (c_2) identificação e seleção dos sete principais *frameworks* de desenvolvimento *Web* PHP utilizados no mercado, incluindo Laravel (LARAVEL, 2018), Phalcon (PHALCON, 2018), Codeigniter (CODEIGNITER, 2018), Yii (YII, 2018), Zend (ZEND, 2018), Symfony (Symantec, 2017) e Cakephp (CAKEPHP, 2018);
- (c_3) identificação e seleção dos três principais WAFs tradicionais (isto é, não oferecidos como serviços online), incluindo ModSecurity (MODSECURITY, 2018), Shadow Daemon (DAEMON, 2018) e Naxsi (NAXSI, 2018);
- (c_4) identificação e implementação, em um cenário controlado, dos dez tipos de vulnerabilidades *Web* mais recorrentes em aplicações *Web*, conforme lista tri-anual de 2013 da OWASP (OWASP, 2013);
- (c_5) análise da eficácia de cobertura dos *scanners* em um ambiente controlado;
- (c_6) identificação de falhas de cobertura (vulnerabilidades críticas não detectadas) dos *scanners*;
- (c_7) identificação de inconsistências entre os manuais das ferramentas e os resultados de varredura;
- (c_8) avaliação do impacto dos *frameworks* de desenvolvimento na segurança do cenário controlado;
- (c_9) avaliação dos WAFs como ferramentas de suporte à mitigação de exploração de vulnerabilidades em sistemas *Web*.

O restante deste texto está organizado como segue. O Capítulo 2 discute os trabalhos relacionados. O Capítulo 3 detalha o desenvolvimento do trabalho. No Capítulo 4 são apresentados e discutidos os resultados. No Capítulo 5 são elencadas as discussões deste trabalho. Por fim, são apresentadas as considerações finais no Capítulo 6.

2 ESTADO DA ARTE

Neste capítulo são discutidos os cenários e tipos de testes utilizados na literatura. Além disso, doze trabalhos distintos são comparados com este trabalho utilizando sete características distintas.

2.1 Cenários de testes

Para avaliar os *scanners* de vulnerabilidades, os estudos existentes utilizam essencialmente um de três tipos de ambientes, OWASP BWA (FERRAO; KREUTZ, 2017; MAKINO; KLYUEV, 2015), sites oficiais de instituições governamentais e/ou privadas (BAU et al., 2010; VIEIRA; ANTUNES; MADEIRA, 2009) e cenários controlados, isto é, cenários contendo vulnerabilidades implementadas especificamente para os testes (HOLÍK; NERADOVA, 2017; ROCHA; KREUTZ; TURCHETTI, 2012; DOUPÉ; COVA; VIGNA, 2010; BAU et al., 2010; FONG et al., 2008). A maioria dos trabalhos utiliza um cenário controlado pelo fato de permitir uma análise mais minuciosa e precisa dos resultados de varredura dos *scanners* de vulnerabilidades. Os cenários controlados tornam também viável e efetiva a detecção de falsos-positivos, ou seja, identificar o que é detectado como vulnerabilidade pelo *scanner* mas na verdade não oferece risco.

Este trabalho utiliza ambientes controlados para a avaliação da eficácia dos *scanners* de vulnerabilidades. Seguindo a mesma metodologia de trabalhos relacionados, o processo inicia-se pela escolha das vulnerabilidades a serem implementadas, seguido da definição do ambiente de testes e das análises manuais das vulnerabilidades implementadas e dos relatórios de saída dos *scanners* (FONSECA et al., 2014; FONSECA; VIEIRA; MADEIRA, 2007). Entretanto, vale ressaltar que a maioria dos trabalhos relacionados foca em vulnerabilidades específicas, como falhas de injeção de código e *cross-site scripting*. Este trabalho, além de implementar estas duas vulnerabilidades, implementa e avalia mais oito falhas presentes na lista das dez mais recorrentes em sistemas *Web* (OWASP, 2013).

2.2 Tipos de testes

O tipo de testes mais aplicado para *scanners* de vulnerabilidades é o *black-box* (FERRAO; KREUTZ, 2017; HOLÍK; NERADOVA, 2017; MAKINO; KLYUEV, 2015; ROCHA; KREUTZ; TURCHETTI, 2012; DOUPÉ; COVA; VIGNA, 2010; BAU et al., 2010), também utilizado neste trabalho. Os testes *black-box* são realizados de forma automatizada e sem acesso direto a detalhes de infraestrutura e codificação do sistema alvo.

Existem também trabalhos que investigam soluções híbridas, ou *grey-box*, que buscam tirar proveito dos aspectos das duas abordagens mais usualmente utilizadas, a *black-box* (utilizada neste trabalho) e a *white-box* (JABALLAH; KHEIR, 2016). A abordagem *white-box* é bastante útil para equipes de desenvolvimento que tem acesso a infraestrutura

e ao código fonte. Uma análise *white-box* pode complementar a análise *black-box* através de testes exaustivos e mais formais de todas as funcionalidades do sistema, por exemplo. Entretanto, atacantes remotos dificilmente terão condições de realizar testes diferentes de *black-box*.

2.3 Trabalhos Relacionados

Existem diferentes estudos que analisam e/ou comparam *scanners* de vulnerabilidades voltados para a *Web* (FUNK; EPP et al., 2018; FERRAO; KREUTZ, 2017; HOLÍK; NERADOVA, 2017; PALSETIA et al., 2016; JABALLAH; KHEIR, 2016; GUPTA; GUPTA, 2015; ANTUNES; VIEIRA, 2015; NAGPAL et al., 2015; MAKINO; KLYUEV, 2015; ZHOU; EVANS, 2014; PELLEGRINO; BALZAROTTI, 2014; DOUPÉ et al., 2012; ROCHA; KREUTZ; TURCHETTI, 2012; BAU et al., 2010; DOUPÉ; COVA; VIGNA, 2010; VIEIRA; ANTUNES; MADEIRA, 2009; FONG et al., 2008; FONSECA; VIEIRA; MADEIRA, 2007). Entretanto, a maioria destas pesquisas analisa ferramentas comerciais, ou um subconjunto muito limitado de ferramentas gratuitas (exemplo: apenas uma ou duas ferramentas), ou são consideravelmente desatualizadas, ou investigam ferramentas específicas que detectam um único tipo ou pequenos subgrupos de vulnerabilidades presentes em sistemas *Web*. Além disso, nenhum desses trabalhos investiga o impacto dos principais *frameworks* de desenvolvimento PHP e WAFs livres disponíveis no mercado.

Estudos demonstram que há uma significativa disparidade entre as ferramentas existentes em termos de abrangência e níveis de exploração das vulnerabilidades. Isso indica a necessidade do uso de não apenas uma única ferramenta, mas sim um conjunto de *scanners* para garantir uma boa cobertura na detecção de vulnerabilidades de sistemas *Web* (FERRAO; KREUTZ, 2017; HOLÍK; NERADOVA, 2017; MAKINO; KLYUEV, 2015; ROCHA; KREUTZ; TURCHETTI, 2012; DOUPÉ; COVA; VIGNA, 2010; VIEIRA; ANTUNES; MADEIRA, 2009). Além disso, de forma recorrente, os estudos apontam a necessidade de evolução e avaliação empírica constante desse tipo de ferramenta.

A Tabela 1 apresenta um comparativo entre alguns dos principais trabalhos encontrados na literatura e citados anteriormente. Para facilitar a representação na tabela, os *papers*/trabalhos foram enumerados da seguinte forma: [P1] (DOUPÉ et al., 2012), [P2] (ROCHA; KREUTZ; TURCHETTI, 2012), [P3] (BAU et al., 2010), [P4] (DOUPÉ; COVA; VIGNA, 2010), [P5] (VIEIRA; ANTUNES; MADEIRA, 2009), [P6] (FONG et al., 2008), [P7] (FONSECA; VIEIRA; MADEIRA, 2007), [P8] (ZHOU; EVANS, 2014), [P9] (MAKINO; KLYUEV, 2015), [P10] (BERTOGLIO; ZORZO, 2017), [P11] (HOLÍK; NERADOVA, 2017), [P12] (FUNK; EPP et al., 2018) e P[13] (este trabalho).

A seguir são apresentadas e discutidas as sete características e/ou parâmetros selecionados para comparar os trabalhos relacionados.

Taxa de falsos-positivos (C1): Esta característica indica se o trabalho avalia (ou não)

Tabela 1 – Comparativo entre os papers/trabalhos

	[P1]	[P2]	[P3]	[P4]	[P5]	[P6]	[P7]	[P8]	[P9]	[P10]	[P11]	[P12]	[P13]
C1	X	X	X	X	X	X	X	X	X	X			X
C2		X							X	X	X	X	X
C3	X		X	X	X	X	X	X					
C4							X	X				X	
C5													X
C6													X
C7												X	X

a taxa de falsos-positivos detectados pelos *scanners*. Entende-se como falso-positivo o que é identificado como uma vulnerabilidade pelo *scanner*, mas que na verdade não representa perigo algum, pois não é uma falha do sistema.

Ferramentas gratuitas e de código aberto (C2): O objetivo é identificar os trabalhos que consideram e avaliam ferramentas gratuitas e de código aberto.

Ferramentas comerciais (C3): Este parâmetro tem como finalidade identificar os trabalhos que avaliam ferramentas comerciais.

Ferramentas específicas (C4): O objetivo do *C4* é indicar se o trabalho analisa apenas um conjunto restrito de uma a três vulnerabilidades.

Impacto de *frameworks* na segurança de sistemas *Web* (C5): Na coluna **C5** da tabela são identificados os trabalhos que avaliam o impacto de *frameworks* de desenvolvimento na segurança de sistemas *Web*.

Documentação versus prática (C6): A característica **C6** indica se o trabalho leva em consideração a documentação (teoria) e os resultados empíricos dos *scanners* investigados. Como exemplo, se uma ferramenta promete detectar a vulnerabilidade X, ela deve, de fato, detectar essa vulnerabilidade na prática.

Impacto de *WAFs* na segurança de sistemas *Web* (C7): A característica **C7** é utilizada para identificar os trabalhos que consideram e avaliam o impacto de *WAFs* na segurança de sistemas *Web*.

Com relação à apresentação das taxas de falsos-positivos (C1) pode ser observado que apenas dois trabalhos não contemplam esta característica (P11 e P12). No caso de ferramentas gratuitas e de código aberto (C2), seis trabalhos (P2, P9, P10, P11, P12, P13) contemplam este quesito. Entretanto, vale ressaltar que este trabalho cobre uma quantidade substancialmente maior das principais ferramentas gratuitas quando comparado com os demais trabalhos. Neste trabalho são avaliados 10 *scanners*, enquanto os demais trabalhos avaliam de 1 a 4 ferramentas.

Observando a Tabela 1, pode-se verificar que este trabalho é o único que avalia o impacto de *frameworks* de desenvolvimento de *software* e investiga a documentação das ferramentas com o objetivo de verificar se elas cumprem o que prometem. Por fim, apenas dois trabalhos (P12 e P13) consideram ou avaliam o impacto de WAFs na segurança de sistemas *Web*. O P12 propõe a implementação de um WAF, enquanto este trabalho (P13) avalia diferentes WAFs na segurança de sistemas *Web*.

Para finalizar, levando em consideração os trabalhos analisados, a seguir são listadas algumas das questões que os pesquisadores frequentemente procuram responder. Este trabalho procura responder essas questões em um contexto atualizado e levando em consideração algumas das principais ferramentas livres disponíveis no mercado.

- (p_1) Qual a quantidade de vulnerabilidades encontradas pelos *scanners* de vulnerabilidades quando usados na prática em sistemas *Web*?
- (p_2) Qual a taxa de falsos-positivos?
- (p_3) Quais são os tipos mais comuns de vulnerabilidades em ambientes *Web*?
- (p_4) Quais vulnerabilidades são as mais detectadas pelas ferramentas?

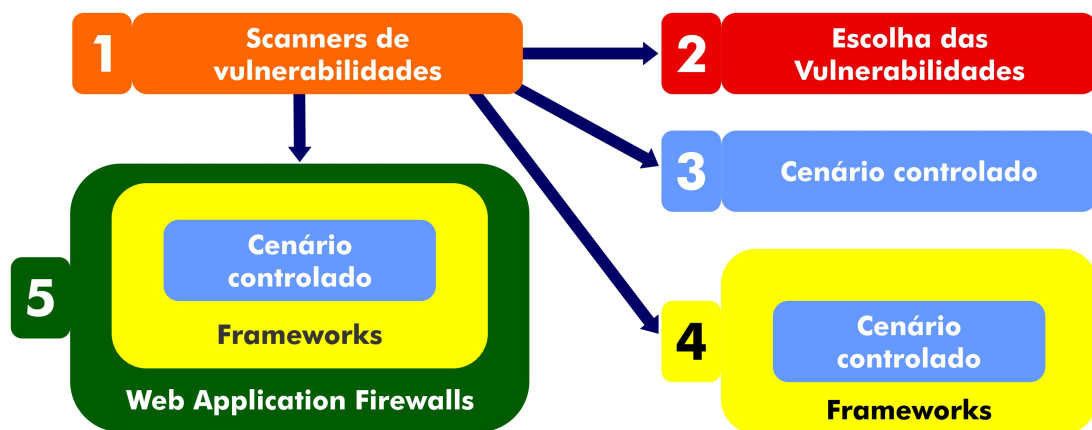
3 DESENVOLVIMENTO

A seguir são discutidas as etapas de desenvolvimento do trabalho. Adicionalmente, são apresentados também alguns detalhes do cenário controlado, incluindo as vulnerabilidades implementadas, ambiente de testes e exemplos práticos de exploração.

3.1 Etapas de desenvolvimento

As cinco etapas de desenvolvimento do trabalho são ilustradas na Figura 1. A **primeira etapa** é constituída pela escolha, instalação e estudo dos *scanners* de vulnerabilidades. De forma similar a estudos recentes (FERRAO; KREUTZ, 2017), foram selecionados dez dos principais *scanners* gratuitos disponíveis na Internet, incluindo Uniscan (ROCHA; KREUTZ; TURCHETTI, 2012), Paros Proxy (SHI; CHEN; YU, 2010), Zed Attack proxy (INFOSEC, 2017), Nessus (BOY; ARANHA, 2016), Andiparos (SHI; CHEN; YU, 2010), Grabber (MACHADO et al., 2016), Wapiti (WILMSHURST; FRYXELL; HUDSONB, 1995), Ratproxy (TEODORO; SERRÃO, 2011), Skipfish (CARVALHO et al., 2013) e Vega (MUNIZ, 2013). Os principais motivos da seleção das ferramentas foram o fato de serem gratuitas e de código aberto, similaridade de objetivos entre elas e sites de classificação de ferramentas de segurança (HACKER, 2017; Terminal Root, 2017), inclusão em distribuições Linux focadas em segurança, como é o caso do Uniscan, que vem no Kali Linux.

Figura 1 – Etapas de desenvolvimento do trabalho



Fonte: Elaborado pela autora.

A seleção das vulnerabilidades a serem implementadas e analisadas no cenário controlado foi realizada na **segunda etapa** da pesquisa. O parâmetro de seleção das vulnerabilidades foi o índice de recorrência ao longo do tempo. Segundo a OWASP, organização internacional que monitora vulnerabilidades em sistemas *Web*, as mais recorrentes, conforme a classificação tri-anual de 2013, são falhas pertencentes às seguintes categorias:

- (c_1) injeção de código;
- (c_2) quebra de autenticação e gerenciamento de sessão;
- (c_3) *cross-site scripting* (XSS);
- (c_4) quebra de controle de acesso;
- (c_5) má configuração de segurança;
- (c_6) exposição de dados sensíveis;
- (c_7) falta de proteção contra ataques;
- (c_8) *cross-site request forgery* (CSRF);
- (c_9) uso de componentes com vulnerabilidades conhecidas; e
- (c_{10}) APIs desprotegidas.

A **terceira etapa** foi constituída pela implementação de um cenário controlado contendo uma vulnerabilidade catalogada em cada uma das dez categorias relacionadas anteriormente. O cenário controlado foi criado a partir da distribuição Kali Linux (versão 4.9.0), servidor MySQL (versão 5.6) e servidor *Web* Apache (versão 2.4.27). O Kali Linux foi escolhido pelo fato de já possuir a maioria das ferramentas instaladas no sistema e ser voltado para testes de segurança. As vulnerabilidades foram implementadas na linguagem de programação PHP (versão 7.0). Enquanto os *scanners* foram instalados no sistema hospedeiro Kali Linux, o cenário controlado foi instalado em uma máquina virtual utilizando o VirtualBox (versão 5.2.18). Os *scanners* de vulnerabilidades foram testados sobre o ambiente controlado. O tipo mais comum de testes para sistemas *Web*, empregado na prática pela maioria dos atacantes, é o *black-box*, também utilizado neste trabalho. Os testes *black-box* são realizados de forma automatizada e sem acesso direto a detalhes de infraestrutura e codificação do sistema alvo.

A **quarta etapa** é dividida em 3 tarefas:

- (t_1) escolha dos *frameworks* de desenvolvimento *Web*;
- (t_2) implementação das dez vulnerabilidades em cada um dos *frameworks*; e
- (t_3) avaliação dos *scanners* em cada *framework*.

Os critérios de seleção foram: (i) *frameworks* para o desenvolvimento de sistemas PHP, gratuitos e de código aberto; (ii) sites de classificação online (MONUS, 2018; ONAIR-CODE, 2017; KINGHOST, 2015; INDIATESTBOOK, 2017); e (iii) o fato de não existirem estudos que comparem estes *frameworks* com relação a aspectos de segurança adicionados automaticamente e transparentemente nas aplicações *Web*. Foram escolhidos *frameworks*

PHP porque, segundo o ranking do site RedMonk (RedMonk, 2014), tanto com relação ao número de projetos no GitHub quanto nas *tags* do StackOverflow, o PHP figura no topo da lista das linguagens mais populares para a *Web*. Universidades como a própria Unipampa utilizam *frameworks* PHP no desenvolvimento dos sistemas institucionais, mais especificamente o Codeigniter.

Com base nos critérios estabelecidos, foram selecionados os *frameworks* Laravel (LARAVEL, 2018), Phalcon (PHALCON, 2018), Codeigniter (CODEIGNITER, 2018), Yii (YII, 2018), Zend (ZEND, 2018), Symfony (SYMFONY, 2018) e Cakephp (CAKEPHP, 2018). O cenário controlado, implementado em cada um dos sete *frameworks*, é composto por um sistema *Web* que contém dois formulários e implementação das dez vulnerabilidades mais recorrentes em sistemas *Web*. Cada um dos dez *scanners* foi avaliado em cada um dos *frameworks*.

Por fim, a **quinta etapa** é composta pela escolha, instalação e testes dos WAFs. Os critérios de seleção foram o fato de serem ferramentas gratuitas e de código aberto, serem de instalação local e sites de classificação (LINKS, 2018; KINGHOST, 2016). Com base nestes parâmetros, foram selecionados os WAFs ModSecurity (MODSECURITY, 2018), Shadow Daemon (DAEMON, 2018) e Naxsi (NAXSI, 2018). Cada um dos WAFs foi testado através dos *scanners* de vulnerabilidades em quatro cenários distintos, nos dois melhores nos dois piores *frameworks* PHP em termos de mitigação de falhas de segurança (conforme apresentado e discutido nos resultados – Capítulo 4).

3.2 *Scanners de vulnerabilidades Web*

A Tabela 2 apresenta o que cada *scanner* se propõe a detectar, segundo suas respectivas documentações técnicas. Como pode ser observado, as ferramentas detectam diferentes tipos de deficiências. Com exceção da Nessus, todas outras ferramentas asseguram detectar *cross-site scripting*. Exceto a Wapiti e a Nessus, todas as outras ferramentas asseguram detectar *SQL injection*. Isto corrobora com o fato de estas duas vulnerabilidades estarem entre as mais detectadas pelos *scanners*. Entretanto, vale ressaltar que algumas das ferramentas não detectam (de forma eficaz) todas as vulnerabilidades listadas na sua documentação, como é o caso da Wapiti, que não detectou injeção de código e nem *cross-site scripting* (ver mais exemplos e detalhes no Capítulo 4).

3.3 Implementação do cenário controlado

Para a implementação das dez vulnerabilidades mais recorrentes em aplicações *Web*, foram criados dois formulários utilizando a linguagem de programação PHP. O primeiro para o acesso ao sistema (login) e segundo para o cadastro de usuários. Estes formulários foram escolhidos por estarem presentes na maioria dos sistemas *Web*. Ambos os formulários possuem os campos login/nome e senha.

Tabela 2 – Vulnerabilidades que cada *scanner* se propõe a detectar

<i>Scanners</i>	<i>Vulnerabilidades</i>
Uniscan	<i>Backup Files, Blind SQL Injection, Local File Include, Hypertext Preprocessor (PHP) CGI Argument Injection, Remote Command Execution, Remote File Include, SQL Injection, Cross-Site Scripting (XSS) e Web Shell Finder.</i>
Paros Proxy	<i>Password Autocomplete in browser, Lotus Domino default files, Directory browsing, SQL Injection Fingerprinting, Cross site scripting, Cross site scripting without brackets, Obsolete file, Private IP disclosure, Session ID in URL rewrite e Obsolete file extended check.</i>
Zed Attack proxy	<i>Cross Site Scripting (Reflected), Remote OS Command Injection, Injeção SQL, External Redirect, Remote File Inclusion, Path Traversal, Injeção SQL - Hypersonic SQL, Application Error Disclosure, X-Frame-Options Header Not Set, Format String Error, Directory Browsing, Buffer Overflow, X-Content-Type-Options Header Missing, Web Browser XSS Protection Not Enabled, Cookie No HttpOnly Flag, Private IP Disclosure e Password Autocomplete in Browser.</i>
Nessus	<i>Apache Banner Linux Distribution Disclosure, Apache Hypertext Transfer Protocol (HTTP) Server Byte Range DoS, HTTP TRACE / TRACK Methods Allowed, Apache HTTP Server httpOnly Cookie Information Disclosure, Apache Server ETag Header Information Disclosure, HTTP Server Type and Version, HyperText Transfer Protocol (HTTP) Information, Web Site Cross-Domain Policy File Detection, HTTP Methods Allowed (per directory), PHP Version e SSL Certificate Cannot Be Trusted.</i>
Wapiti	<i>File disclosure, Database Injection, XSS, injection, Command Execution detection, CRLF Injection, XXE, SSRF, use of know potentially dangerous files, Weak e Shellshock.</i>
Andiparos	<i>Password Autocomplete in browser, Lotus Domino default files, Directory browsing, SQL Injection Fingerprinting, Cross site scripting, Cross site scripting without brackets, Obsolete file, Private IP disclosure, Session ID in URL rewrite e Obsolete file extended check.</i>
Grabber	<i>Cross-Site Scripting, SQL Injection, File Inclusion, Backup files check, Simple AJAX check, Hybrid analysis/Crystal, JavaScript source code analyzer.</i>
Ratproxy	<i>Problems with adding scripts, SQL injection, content placement issues, defenses XSRF e XSS.</i>
Skipfish	<i>Code Injections, Integer Overflow Vulnerabilities, Sites Accepting HTTP PUT, XSS Vectors, Cross Directory Inclusion, Assorted File POIs, Script Provided by the Attacker and Cascading Style Sheets (CSS) Include Vectors, MIME Types, Missing or Wrong Characters in Renderable, Conflicting Information of MIME / charset in renders and Incorrect caching policies in cookie configuration responses. Directory redirection vectors, Redirecting URLs provided by the attacker, Embedded content, SQL injection, HTTP credentials on URLs, SSL certificates expired or not yet valid, HyperText Markup Language (HTML) forms without XSRF protection, Self-signed SSL certificates, SSL certificate host name differences, and Incorrect cache policies on less sensitive content</i>
Vega	<i>XSS, SQL injection, Teflected cross-site scripting, stored cross-site scripting, blind SQL injection, remote file include e shell injection</i>

Os códigos PHP utilizados pelos formulários implementados permitem a passagem de variáveis como parâmetro de uma página para outra, o que é uma prática comum em sistemas *Web* PHP. A seguir, são apresentados quatro exemplos de como um atacante pode explorar as vulnerabilidades de injeção de código, utilização de componentes com vulnerabilidades conhecidas, *cross-site scripting* e quebra de controle de acesso.

O trecho de código 3.1 representa a consulta SQL utilizada no sistema para verificar o **usuario** e a **senha**. O comando **SELECT** retorna todas as linhas da tabela usuários, sendo que **usuario** é igual a variável **usuario** e **senha** é igual a variável **hashDaSenha**. Como pode ser observado, parece ser um código trivial e correto mas na verdade é um código vulnerável que permite *SQL injection*.

```
1 SELECT * FROM usuarios
2     WHERE usuario = '$usuario'
3     AND senha = '$hashDaSenha'
```

Trecho de código 3.1 – Exemplo de *SQL injection*

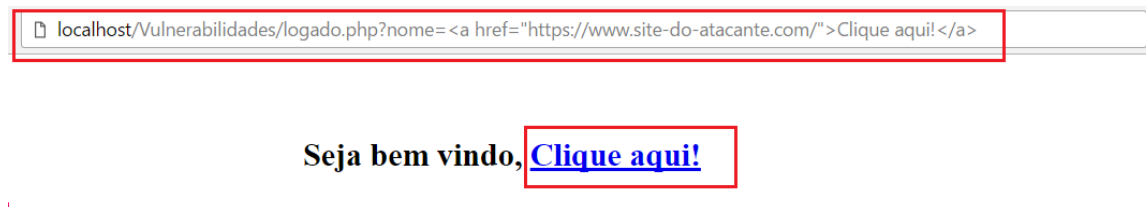
No comando SQL do trecho de código 3.1, as variáveis **usuario** e **senha** representam um problema grave de segurança. Como os dados de entrada destas variáveis não são testados, um atacante pode explorar elas para mudar ou completar o comando SQL que será executado no banco de dados.

Suponha que o atacante entre com o valor “1’ or ‘1’ = ‘1” no campo **usuario** do formulário de *login*. Consequentemente, a variável **usuario** vai impactar na construção do comando SQL, uma vez que o caracter aspas simples (') é especial. Sempre que 1 for igual a 1, a sessão será liberada sem verificar a senha armazenada no banco de dados. Como 1 é sempre igual a 1, a sessão sempre será liberada. Isso ocorre porque os campos **usuario** e **senha** não estão sendo validados. Uma das formas de resolver o problema é através de filtros que limitam os valores de entrada do usuário.

Outra falha encontrada com frequência em sistemas *Web* são as derivadas de componentes com vulnerabilidades conhecidas. Estas vulnerabilidades são descobertas em componentes disponíveis na Internet e utilizadas em diferentes sistemas. Uma vez publicadas as deficiências do componente, qualquer sistema que utilize o componente pode ser explorado por usuários maliciosos. No cenário controlado, foi utilizado o XAMPP 2.4.27, cuja versão possui vulnerabilidades de injeção de código, permitindo ao atacante executar comandos SQL arbitrários. Por exemplo, digamos que um usuário mal intencionado insira uma consulta incluindo a concatenação de uma variável através do método POST, pelo *INPUT* de um dos formulários de *login*. Através do método `$_POST` em PHP, tem-se a indicação de uma requisição pelo método HTTP POST, onde existe a passagem de variáveis no momento da submissão, neste caso, chamada de “valor”. Supondo que o valor do POST seja modificado, o resultado implicaria em uma consulta no banco de dados que daria ao atacando acesso a informações não autorizadas.

Através da vulnerabilidade de *cross-site scripting*, o atacante pode usar uma página PHP para redirecionar usuários para sites que contenham vulnerabilidades, reconfigurar páginas e/ou roubar sessões ativas de usuários. Um exemplo de exploração desta vulnerabilidade é a através da inserção de um comando HTML na atribuição de valor a uma variável passada por parâmetro a uma página PHP. Por exemplo, a URL `<http://localhost/Vulnerabilidades/logado.php?nome=Cli%que%aqui!>` vai criar um *link* para o site do atacante. Quando o atacante enviar essa URL do sistema para a vítima, ela vai visualizar a URL modificada como parte da página do sistema. O resultado de uma página reconfigurada, através da URL modificada, é ilustrada na Figura 2.

Figura 2 – Página com a vulnerabilidade de *cross-site scripting*



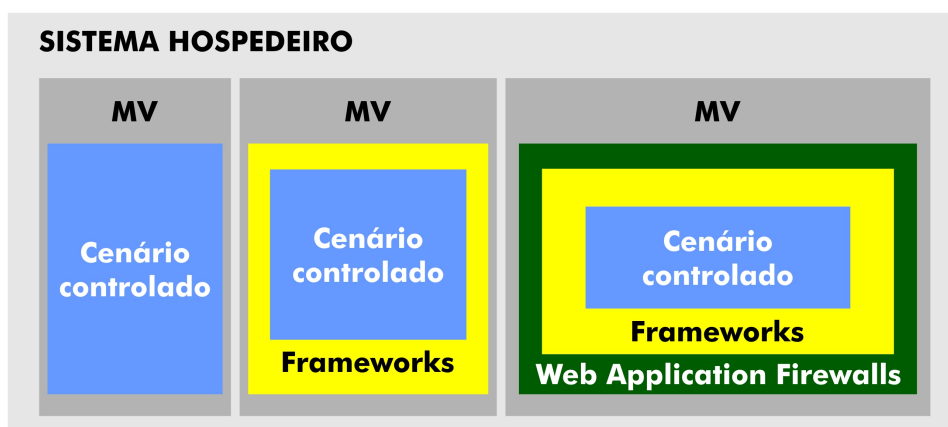
Fonte: Elaborado pela autora.

No caso da quebra de controle de acesso, o sistema não verifica se o usuário está (ou não) autenticado no sistema. Esta vulnerabilidade permite a um atacante utilize força bruta para conseguir acesso ao sistema. Na prática, utilizando a URL `<http://localhost/Vulnerabilidades/logado.php?nome=usuarioValido>`, o atacante poderá quebrar o controle de acesso do sistema. Para isso, basta descobrir um nome de usuário válido.

3.4 Ambiente de desenvolvimento e testes

A Figura 3 ilustra o ambiente de testes. Os testes práticos deste trabalho foram realizadas em uma máquina Dell *Inspiron Special Edition*, Intel core i7, sexta geração, memória RAM de 16 GB e 1 TB de HD + 8 GB de SSD. Para cada cenário controlado foi criada uma máquina virtual VirtualBox (versão 5.2.18) com o sistema operacional Linux Kali 4.9.0, a linguagem de programação PHP 7.0, e servidor de bando de dados MySQL 5.6 e o servidor *Web Apache* 2.4.27. A máquina física contém um sistema hospedeiro que armazena e gerencia as máquinas virtuais. Para cada um dos cenários de testes, foi preparada uma máquina virtual distinta.

Figura 3 – Ambiente de testes do trabalho

MÁQUINA

Fonte: Elaborado pela autora.

4 RESULTADOS

Neste capítulo são apresentados e discutidos os principais resultados obtidos no decorrer desse trabalho. Na seção 4.1 são introduzidos os resultados obtidos com a avaliação dos *scanners* no cenário controlado. Na seção 4.2 são discutidos os resultados das ferramentas de varredura sobre os *frameworks* PHP. Por fim, os WAFs são adicionados e avaliados em conjunto com quatro *frameworks* (seção 4.3).

4.1 Cenário Controlado

A Tabela 3 sumariza os principais resultados obtidos com os testes *black-box* no cenário controlado. Na tabela são apresentadas as categorias das vulnerabilidades e as ferramentas que a detectaram, respectivamente.

Tabela 3 – Detecção de vulnerabilidades no cenário controlado

Vulnerabilidades	Web scanners
Injeção de Código	Uniscan Skipfish
Quebra de autenticação e gerenciamento de sessão	
<i>Cross-site scripting</i>	Andiparos Nessus Grabber Skipfish Vega Zed Attack proxy
Quebra de controle de acesso	
Má configuração de segurança	Andiparos Uniscan Paros Skipfish Vega Zed Attack proxy
Exposição de dados sensíveis	Andiparos Nessus Skipfish Vega Zed Attack proxy
Falta de proteção contra ataques	
<i>Cross-site requesty forgery (CSRF)</i>	
Uso de componentes com vulnerabilidades conhecidas	Nessus
APIs desprotegidas	

As vulnerabilidades mais detectadas (ou detectadas por um maior número de ferramentas) foram *cross-site scripting* e má de configuração de segurança, como pode ser observado na Tabela 3. Estes resultados são condizentes com as conclusões de trabalhos recentes em cenários como o BWA (FERRAO; KREUTZ, 2017). A vulnerabilidade *cross-*

site scripting está entre as mais frequentemente detectadas pelos *scanners* em diferentes cenários.

Ao todo, os *scanners* de vulnerabilidades detectaram apenas 50% das 10 vulnerabilidades implementadas no cenário controlado. Apenas as vulnerabilidades injeção de código, *cross site scripting*, exposição de dados sensíveis, má configuração de segurança e uso de componentes com vulnerabilidades conhecidas foram identificadas pelos *scanners*. Este é um cenário real preocupante uma vez que muitos usuários confiam nessas ferramentas como forma de detectar e diagnosticar problemas em sistemas *Web*.

Nenhum dos *scanners* detectou vulnerabilidades como quebra de controle de acesso e APIs desprotegidas. A falha de APIs desprotegidas é uma das mais recentes a entrar no *ranking* das vulnerabilidades mais recorrentes da OWASP. Esta é uma das principais razões pela qual os *scanners* falham na detecção, ou seja, as ferramentas de varredura ainda não oferecem *plugins* ou recursos para detectar APIs desprotegidas.

As vulnerabilidades em APIs desprotegidas podem facilitar o acesso e a manipulação não autorizada de dados sensíveis, possibilitar a desconfiguração de páginas do site ou até mesmo permitir ao atacante comprometer o sistema. No cenário controlado, as chaves da API Javascript do Google Maps foram incorporadas diretamente no código, ou seja, elas estão expostas. Para corrigir esta falha, a chave de API deve ser armazenada em variáveis de ambiente ou em arquivos fora da árvore de origem do sistema *Web*.

A vulnerabilidade de quebra de controle de acesso concede ao atacante acesso a dados restritos através da verificação incorreta da autenticação do usuário (está ou não logado) no sistema. Esta vulnerabilidade permite ao atacante utilizar força bruta para explicar e conseguir acesso ao sistema. Para quebrar o controle do acesso, é o suficiente descobrir um nome de um usuário válido no sistema *Web*. A prevenção contra esta falha requer uma verificação dos acessos ao sistema. Para cada painel do sistema, deve-se incluir uma verificação com *tokens*, para garantir que o usuário esteja autorizado a ter acesso ao recurso solicitado. A cada solicitação deve-se comparar o *token* de solicitação com o salvo na sessão do usuário. Caso o *token* não corresponder, a solicitação é classificada com inválida.

O *scanner* que se destacou em relação as demais foi o Skipfish, detectando 80% das categorias de vulnerabilidades identificadas pelas ferramentas. Em segundo lugar, o Nessus detectou 60% das vulnerabilidades. Estranhamente, em uma pesquisa recente, utilizando o ambiente não controlado da BWA, os *scanners* que se destacaram foram o Zed Attack Proxy e Paros Proxy (FERRAO; KREUTZ, 2017). Isto nos leva a concluir que o cenário atual é mais complicado e crítico do que o imaginado, ou seja, não há uma fórmula simples para escolher ferramentas de varredura que serão utilizados para detectar as vulnerabilidades de uma aplicação *Web*. De fato, observando trabalhos recentes e o estado da arte, como pergunta de pesquisa, podem ser listadas perguntas como: *Qual(is) scanner(s) de vulnerabilidades é(são) mais eficaz(es) para um determinado contexto?* Esta

questão, entre outras, é discutida no Capítulo 5.

O Nessus foi o único *scanner* a detectar a vulnerabilidade de componentes com vulnerabilidades conhecidas. O Nessus possui mais de 2.600 *plugins* para detectar vulnerabilidades conhecidas em aplicativos da *Web*. Esses *plugins* são escritos para enumerar e detectar vulnerabilidades que foram publicamente relatadas em algum produto da *Web*, seja ele de código aberto ou comercial. No cenário controlado, foi utilizado o XAMPP 2.4.27, cuja versão possui vulnerabilidades de injeção de código, permitindo aos atacantes executarem comandos SQL arbitrários. Neste caso em particular, diferentemente dos outros *scanner*, o Nessus possui um *plugin* que permite a detecção desta vulnerabilidade.

A Tabela 4 apresenta alguns resultados e métricas relativos aos processos de detecção de cada ferramenta no cenário controlado, como a porcentagem em relação a todas as vulnerabilidades detectadas e o tempo de execução. Como pode ser observado na tabela, o tempo de execução não tem a ver necessariamente com a cobertura. O Uniscan, apesar de possuir um alto tempo de execução, não detectou um número significativo de vulnerabilidades. Este elevado tempo de execução ocorre devido ao fato de esta ferramenta possuir um número maior (e mais diversificado) de combinações de testes para determinadas vulnerabilidades, como *SQL injection*. A ferramenta de melhor desempenho foi a Skipfish, detectando 80% das vulnerabilidades detectadas pelos *scanners*. Já o *scanner* que obteve a melhor relação entre o tempo de execução e a porcentagem de detecção foi o Vega, detectando 60% das vulnerabilidades em apenas 1 minuto. Observe que o Vega fica próximo do Skipfish em termos de porcentagem de detecção, entretanto, consumindo apenas 1,11% do tempo computacional.

Tabela 4 – Principais resultados e métricas no cenário controlado

Ferramenta	Porc. de detecção	Tempo de execução
Uniscan	40%	12 hrs e 14 min
Paros	20%	20 minutos
Zed Attack	60%	5 minutos
Nessus	60%	5 minutos
Ratproxy	0 %	14 minutos
Grabber	20%	4 minutos
Wapiti	0 %	2 minutos
Andiparos	60%	18 minutos
Skipfish	80%	1 hora 30 minutos
Vega	60 %	1 minuto

A taxa de falsos-positivos do conjunto de *scanners* testados ficou em 20%. A identificação dos falsos-positivos foi realizada através da verificação dos relatórios de saída de cada ferramenta e consequente análise manual da vulnerabilidade relatada pela ferramenta. Os relatórios dos *scanners* contém os nomes das vulnerabilidades, o nível de gravidade, localização da vulnerabilidade no sistema e técnicas para corrigi-las. Conforme é apresentado na Tabela 5, o Skipfish detectou 85,71% dos falsos-positivos encontrados.

A ferramenta identificou funcionalidades e arquivos supostamente vulneráveis que não oferecem risco algum, como arquivos de estilização do sistema (CSS). Em segundo lugar, a ferramenta Zed Attack Proxy detectou 28,57% dos falsos-positivos.

Tabela 5 – Taxa de falsos-positivos no cenário controlado

Ferramenta	Porc. de falsos-positivos
Uniscan	0 %
Paros	14,29 %
Zed Attack	28,57 %
Nessus	0 %
Ratproxy	0 %
Grabber	0 %
Wapiti	0 %
Andiparos	14,29 %
Skipfish	85,71 %
Vega	14,29 %

Apesar do número de falsos-positivos, vale ressaltar que um falso positivo consiste na identificação de uma vulnerabilidade, que, na verdade, não representa nenhum perigo. Em outras palavras, no pior caso, os falsos-positivos implicam em uma segunda verificação de propriedades, funcionalidades e códigos dos sistemas. Por um lado, isto ajuda a ratificar a inexistência de falhas de segurança nos sistemas. Por outro lado, uma segunda verificação manual aumenta o custo do ciclo de desenvolvimento e evolução do *software*. Portanto, fica evidente que ainda há espaço para investigação e desenvolvimento no sentido de eliminar a geração de falsos-positivos.

Considerando os resultados apresentados, os *scanners* de vulnerabilidades, em especial os gratuitos, precisam claramente evoluir, isto é, melhorar a eficácia em termos de detecção de vulnerabilidades em sistemas *Web*. Isto fica ainda mais evidente na Tabela 6, onde são relacionadas as vulnerabilidades que os *scanners* prometem detectar, mas, de fato, não detectam ou detectam de forma muito limitada (exemplo: em casos muito específicos de ocorrência da vulnerabilidade). Isto significa que não é possível confiar-se na documentação técnica dos *scanners*. Para realizar esta análise, foram analisados os manuais e a documentação técnica das ferramentas. Algumas ferramentas não detectaram algumas das vulnerabilidades que deveriam detectar segundo a sua própria documentação. Vale ressaltar que foram utilizadas as formas mais básicas de implementação das vulnerabilidades *Web* no cenário controlado.

Um dos casos investigados, em um nível técnico mais granular, foi o *scanner* Grabber, cuja documentação assegura detectar *SQL injection*. O Grabber possui interface por linha de comando e foi desenvolvido na linguagem de programação Python. Para a vulnerabilidade de *SQL injection*, a ferramenta possui uma lista com alguns casos de testes de injeção de código. Esta lista é denominada `sqlAttacks`. Porém, o caso implementado no ambiente controlado, que é um dos ataques SQL mais simples e comuns (exemplo:

Tabela 6 – Falhas na detecção de vulnerabilidades.

Vulnerabilidades	<i>scanners</i> que falharam em detectar
<i>Injeção de Código</i>	<i>Andiparos</i> <i>Ratproxy</i> <i>Wapiti</i> <i>Grabber</i> <i>Paros proxy</i> <i>Vega</i>
<i>Cross-site scripting</i>	<i>Ratproxy</i> <i>Uniscan</i> <i>Wapiti</i> <i>Paros proxy</i>
Quebra de controle de acesso	<i>Skipfish</i>
Má configuração de segurança	<i>Nessus</i> <i>Ratproxy</i> <i>Wapiti</i> <i>Grabber</i>

“’OR 1=1; #”), não consta na lista de testes da ferramenta. Além disso, a quantidade de testes de injeção SQL do Grabber é muito limitada, restringindo-se a apenas sete testes distintos.

Esses resultados apontam para um cenário bastante preocupante no ecossistema da *Web*. Os usuários precisam preocupar-se em utilizar múltiplos *scanners* de vulnerabilidades e não podem confiar na própria documentação técnica das ferramentas.

4.2 Cenário Controlado com Frameworks

A Tabela 7 sumariza os principais resultados obtidos com os testes *black-box* nos cenários controlados utilizando os sete *frameworks* PHP. Na tabela aparecem os nomes dos *frameworks*, as vulnerabilidades detectadas, os *scanners* e a porcentagem de vulnerabilidades detectadas, respectivamente. Vale ressaltar que as mesmas dez vulnerabilidades foram implementadas em todos os sete *frameworks*.

Os *scanners* encontraram 70% das vulnerabilidades implementadas nos *frameworks* Phalcon e Codeigniter, garantindo as duas últimas posições em termos de mecanismos e recursos padrão para o desenvolvimento de aplicações *Web* seguras. O Phalcon foi o único *framework* suscetível às vulnerabilidades de quebra de autenticação e gerenciamento de sessão e quebra de controle de acesso, ambas identificadas pelo *scanner* Zed Attack Proxy. Assim como a maioria dos demais *frameworks*, o Phalcon mitiga ataques de injeção de código de forma automática. Entretanto, nestes dois *frameworks* a maioria dos riscos de segurança precisam ser tratados manualmente pelos desenvolvedores e/ou administradores dos sistemas.

Os *scanners* detectaram a vulnerabilidade de *cross site scripting* no Codeigniter,

Tabela 7 – Vulnerabilidades detectadas nos cenários dos *frameworks*

Framework	Vulnerabilidades detectadas	Web scanners	% V.
Laravel	<i>Cross site scripting</i> Má configuração de segurança Exposição de dados sensíveis Vulnerabilidades conhecidas	[Nessus,Grabber,Vega,Zed Attack proxy] [Zed Attack proxy] [Paros,Vega,Zed Attack proxy] [Nessus]	40%
Symfony	Injeção de código Má configuração de segurança Exposição de dados sensíveis <i>Cross-site requesty forgery</i>	[Andiparos,Nessus] [Andiparos,Paros,Skipfish] [Skipfish] [Nessus]	40%
Yii	Injeção de código <i>Cross site scripting</i> Má configuração de segurança <i>Cross-site requesty forgery</i> Vulnerabilidades conhecidas	[Andiparos,Nessus,Paros] [Vega,Zed Attack proxy] [Andiparos,Paros,Vega] [Nessus] [Nessus]	50%
Zend	Injeção de código <i>Cross site scripting</i> Exposição de dados sensíveis <i>Cross-site requesty forgery</i> Vulnerabilidades conhecidas	[Andiparos,Nessus] [Andiparos,Nessus,Grabber,Vega,Zed at- tack proxy] [Andiparos,Paros] [Nessus] [Nessus]	50%
Cakephp	Injeção de código <i>Cross site scripting</i> Má configuração de segurança Exposição de dados sensíveis <i>Cross-site requesty forgery</i> Vulnerabilidades conhecidas	[Nessus] [Vega,Zed attack proxy] [Paros] [Vega] [Nessus] [Nessus]	60%
Codeigniter	Injeção de código <i>Cross site scripting</i> Má configuração de segurança Exposição de dados sensíveis Falta de proteção contra ataques <i>Cross-site requesty forgery</i> Vulnerabilidades conhecidas	[Andiparos,Nessus,Paros] [Nessus,Paros,Vega,Zed attack proxy] [Andiparos,Grabber,Paros,Skipfish] [Uniscan,Skipfish,Vega] [Skipfish] [Nessus] [Nessus]	70%
Phalcon	Quebra de controle de sessão <i>Cross site scripting</i> Quebra de controle de acesso Má configuração de segurança Exposição de dados sensíveis <i>Cross-site requesty forgery</i> Vulnerabilidades conhecidas	[Zed attack proxy] [Vega,Zed attack proxy] [Zed attack proxy] [Paros] [Vega] [Nessus] [Nessus]	70%

bem como em outros *frameworks*. No caso do Codeigniter, realizando uma análise técnica manual, o *framework* possui um método chamado `xss clean` dentro da classe `Security`. O `xss clean` é utilizado para tratamento dos dados contra ataques XSS. Porém, a aplicação do método vem desativada por padrão no Codeigniter. É de conhecimento comum que

a maioria dos programadores utilizam as ferramentas e *frameworks* de desenvolvimento de *software* em modo padrão. Portanto, há duas alternativas. A primeira é conscientizar os desenvolvedores e a segunda é forçar a habilitação do `xss clean` no modo padrão do *frameworks*.

O *framework* Symfony possui suporte a recursos de segurança, como o seu próprio *firewall*. Diferentemente do Codeigniter, o Symfony mitigou a vulnerabilidade de *cross site scripting* porque ele possui um mecanismo denominado Twig pré-instalado. Por padrão, o Twig realiza uma “limpeza” dos dados de entrada fornecidos pelo usuário antes de renderizar a saída HTML.

O Codeigniter foi o único *framework* suscetível a falta de proteção contra ataques. Os mecanismos de proteção contra ataques, como monitoramento e controle do número de sessões abertas simultaneamente por um mesmo usuário, tem por objetivo detectar, responder e bloquear ataques, tornando o trabalho do atacante mais difícil. Por exemplo, é recomendado que aplicativos *Web* não permitam ao usuário manter múltiplas sessões ativas ao mesmo tempo. Na prática, isto significa que um usuário não deveria ter permissão para abrir múltiplas vezes o mesmo aplicativo *Web*, como em navegadores ou computadores distintos. Na melhor das hipóteses, o usuário deveria ser alertado sobre as tentativas de abrir novas sessões no mesmo sistema. Com isso, ele teria condições de decidir se a nova sessão é sua ou resultado de um possível ataque.

Por padrão, o Codeigniter permite que os usuários tenham múltiplas sessões simultaneamente ativas. Este foi o motivo pelo qual os *scanners* detectaram a falta de proteção contra ataques. Para resolver este problema, pode-se acoplar ao código a classe `session` do Codeigniter. Esta classe permite verificar se o usuário está com a sessão ativa, além de rastrear as atividades enquanto ele navega no site. Vale ressaltar, novamente, que esta é uma ação manual que exige ciência dos desenvolvedores da aplicação *Web*.

Os *frameworks* que obtiveram o melhor resultado em termos de mecanismos de segurança para aplicações *Web* foram o Laravel e o Symfony. Nos dois casos, os *scanners* detectaram apenas 40% das vulnerabilidades implementadas, conforme é apresentado na Tabela 7. Isto significa que, em sua configuração padrão, esses dois *frameworks* são significativamente superiores em termos de segurança. Por exemplo, as ferramentas de varredura detectaram 70% das falhas no ambiente controlado do Phalcon e do Codeigniter, o que representa praticamente o dobro de vulnerabilidades.

Um dos objetivos do Laravel é justamente oferecer simplicidade à implementação de recursos de segurança como autenticação. Em termos de segurança, os principais recursos já vêm pré-configurados e habilitados por padrão. Como exemplo, tanto o registro quanto a autenticação de usuários já estão habilitados e disponíveis por padrão para os programadores. Outro exemplo são as senhas, que são cifradas de forma automática pelo *framework*. Vale também ressaltar que o Laravel utiliza os conceitos de “provedores” e “guardas” para facilitar o processo de autenticação. O objetivo dos “guardas” é autenticar

os usuários para cada solicitação que eles fazem, enquanto os “provedores” facilitam a recuperação dos usuários do banco de dados.

O Laravel foi o único *framework* que mitigou a vulnerabilidade *cross site request forgery*. O *framework* utiliza *tokens* CSRF para garantir que atacantes não possam gerar solicitações falsas. A cada solicitação, o Laravel cria e integra um *token*, através de uma chamada de AJAX. Quando a solicitação é invocada, o *framework* compara o *token* de solicitação com o salvo na sessão do usuário. Caso o *token* não corresponder, a solicitação é classificada com inválida e nenhuma outra ação é executada.

Ademais, o Laravel também protege os sistemas *Web* contra injeção de código. Para isto, o *framework* utiliza o *Eloquent ORM* com a ligação de parâmetro PDO que esquia de qualquer entrada utilizando o comando **WHERE** do SQL. Esta ligação garante que os atacantes não possam passar dados de consulta com a intenção de modificar a consulta. Portanto, utilizando o Laravel não há necessidade de limpar cadeias de caracteres.

Os resultados apresentados mostram que, apesar de um bom *framework* poder ajudar a aumentar a segurança de aplicações *Web* de forma automática e transparente, ainda há espaço para pesquisa e desenvolvimento. Novos mecanismos de segurança podem ser adicionados aos *frameworks* para impedir que a maioria das vulnerabilidades mais recorrentes em sistemas *Web*, como as dez implementadas, possa ser facilmente explorada pelos atacantes.

4.3 Web Application Firewalls

A Tabela 8 sumariza os resultados obtidos com os testes dos WAFs adicionados aos cenários controlados dos *frameworks*. Na tabela aparecem os nomes dos WAFs, os nomes dos *frameworks*, as vulnerabilidades detectadas, os *scanners* que as detectaram e a porcentagem de vulnerabilidades detectadas, respectivamente. De forma similar às análises anteriores, os dez *scanners* de vulnerabilidades foram testados em cada um cenários de *frameworks* com WAFs.

Os testes foram realizados em quatro cenários distintos. Primeiro, nos dois *frameworks* que apresentaram o melhor desempenho na avaliação realizada na seção 4.2, o Laravel e o Symfony. Segundo, nos dois *frameworks* mais suscetíveis a falhas, o Codeigniter e o Phalcon. Para cada um dos quatro cenários foram ativados cada um dos três WAFs, resultando em um total de doze cenários de testes, como pode ser observado na Tabela 8.

O WAF que apresentou o melhor desempenho foi o ModSecurity, chegando a evitar a detecção de 70% das vulnerabilidades nos *frameworks* Laravel e Symfony. Se compararmos com os resultados da seção 4.2, isto representa uma melhora de 10%, uma vez que apenas 30% das vulnerabilidades foram detectadas com o ModSecurity ativado, enquanto que 40% foram detectadas com apenas o *framework*. O ModSecurity gera logs do tráfego HTTP. A monitoração e detecção de ataques ocorre em tempo real, através de políticas

Tabela 8 – Vulnerabilidades detectadas nos cenários dos *frameworks* e WAFs

WAFs	Frameworks	Vulnerabilidades detectadas	Web scanners	% V.
ModSecurity	Laravel	Má configuração de segurança Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Zed attack proxy] [Nessus]	30%
	Symfony	Exposição de dados sensíveis Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Skipfish] [Nessus]	30%
	Codeigniter	Má configuração de segurança Exposição de dados sensíveis Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Andiparos, Grabber, Paros, Skipfish] [Uniscan, Skipfish, Vega] [Nessus]	40%
	Phalcon	<i>Cross-site scripting</i> , Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i> Má configuração de segurança Exposição de dados sensíveis	[Nessus] [Paros] [Vega]	50%
Shadow D.	Laravel	Má configuração de segurança <i>Cross-site scripting</i> Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Zed attack proxy] [Nessus, Grabber, Vega, Zed attack proxy] [Nessus]	40%
	Symfony	<i>Cross-site scripting</i> , Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i> Má configuração de segurança Exposição de dados sensíveis	[Vega, Nessus] [Andiparos, Paros, Skipfish] [Skipfish]	50%
	Codeigniter	Má configuração de segurança <i>Cross-site scripting</i> Exposição de dados sensíveis Quebra de sessão Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Andiparos, Nessus, Paros] [Paros, Vega] [Uniscan, Skipfish, Vega] [Skipfish], Quebra de controle de acesso [Nessus]	70%
	Phalcon	Má configuração de segurança Exposição de dados sensíveis Quebra de sessão, Quebra de controle de acesso Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Paros] [Vega] [Skipfish] [Nessus]	60%
Naxsi	Laravel	<i>Cross-site scripting</i> Quebra de controle de acesso Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Nessus, Grabber, Vega] [Skipfish] [Nessus]	40%
	Symfony	<i>Cross-site scripting</i> Má configuração de segurança Exposição de dados sensíveis Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Vega, Zed attack proxy] [Andiparos, Paros, Skipfish] [Skipfish] [Nessus]	50%
	Codeigniter	Má configuração de segurança Exposição de dados sensíveis Quebra de sessão, Quebra de controle de acesso Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Andiparos, Grabber, Paros] [Uniscan, Skipfish, Vega] [Skipfish] [Nessus]	60%
	Phalcon	<i>Cross-site scripting</i> Má configuração de segurança Exposição de dados sensíveis Quebra de sessão Quebra de controle de acesso Vulnerabilidades conhecidas, <i>Cross-site requesty forgery</i>	[Vega, Zed attack proxy] [Paros] [Vega] [Skipfish] [Skipfish] [Nessus]	70%

padrão como bloquear a requisição quando encontrada uma ameaça ou redirecionar o usuário para uma página específica. O ModSecurity oferece mecanismos de regras flexíveis, permitindo a modificação ou criação de regras através de expressões regulares em Perl.

O Trecho de código 4.1 ilustra uma expressão regular utilizada pelo ModSecurity para detectar injeção de código. A expressão regular irá procurar por parâmetros (e.g `echo|exec|printenv|include`) nos cabeçalhos das requisições *Web*.

```
1 SecRule ARGS|REQUEST_HEADERS "@rx (?i)<!--\W*?\#\W*?(?:echo|exec|printenv|
  include)" \
2 "phase:2,t:none,pass,msg:'SSL injection'"
```

Trecho de código 4.1 – Regra de injeção de código do ModSecurity

É interessante observar nos resultados que o impacto mais significativo, com o ModSecurity, foi nos cenários dos *frameworks* Codeigniter e Phalcon. Apenas com os *frameworks*, os *scanners* conseguiam detectar até 70% das vulnerabilidades desses dois cenários. Entretanto, adicionando o ModSecurity, a taxa de detecção ficou em 40% para o Codeigniter e 50% para o Phalcon, ou seja, uma melhora significativa. Os 10% de diferença entre os dois *frameworks* foi na detecção da vulnerabilidade *cross-site scripting*. Esta vulnerabilidade foi mitigada no *framework* Codeigniter porque o ModSecurity possui um arquivo, denominado `modsecurity_crs_41_xssattacks`, com regras que detectam ataques do tipo XSS. Já no caso do Phalcon, as regras não cobriram a assinatura de XSS do *framework*, como exemplo de entrada, `"<script>alert("Clique aqui!")</script>"`.

Os WAFs Shadow Daemon e Naxsi apresentaram uma melhora de 10% para os cenários dos *frameworks* Phalcon e Codeigniter, respectivamente. Exceto o Symfony, nos demais casos não houve alteração. Em outras palavras, isto significa que estes WAFs pouco contribuíram para mitigar as vulnerabilidades dos ambientes *Web* avaliados. No caso do Shadow Daemon, isto era esperado uma vez que este WAF promete mitigar essencialmente vulnerabilidades de injeção de código (como SQL, XML e comandos) e inclusão remota de arquivos. Já o Naxsi afirma mitigar apenas as vulnerabilidades de *SQL injection* e XSS.

Todos os WAFs, em todos os doze cenários, mitigaram a vulnerabilidade de injeção de código. No caso do ModSecurity, ele possui o **slr rules**, que é um diretório de arquivos de regras que oferecem proteção contra ataques de *SQL injection*. Os arquivos contêm regras específicas para diferentes assinaturas desta vulnerabilidade. Finalmente, vale ressaltar ainda que nenhum dos WAFs possui regras específicas para evitar as vulnerabilidades de utilização de componentes com vulnerabilidades conhecidas e *cross site requesty forgery*. Como resultado, estas duas vulnerabilidades não foram mitigadas.

5 DISCUSSÃO

A seguir são listadas e discutidas algumas questões mais específicas relacionadas ao trabalho desenvolvido.

Se um scanner S1 detecta a vulnerabilidade V1 (e.g. XSS) no ambiente controlado A1, isto significa que ele irá detectar a mesma vulnerabilidade, implementada da mesma forma, em um outro ambiente A2 qualquer?

A resposta curta é *não necessariamente*. A detecção vai depender de diferentes fatores, como os mecanismos de segurança empregados no ambiente A2. Por exemplo, suponhamos que no cenário A2 o sistema seja implementado com a linguagem PHP, mas sem a utilização de recursos extras como APIs ou *frameworks*. Já no ambiente A2, o sistema *Web*, contendo também a vulnerabilidade V1, foi codificado utilizando um *framework* F1 (e.g. Codeigniter). Neste caso, se o método `xss_clean` (método utilizado para tratamento dos dados contra ataques XSS) do *framework* F1 estiver habilitado, o *scanner* S1 dificilmente conseguirá detectar a vulnerabilidade V1 no cenário A2.

Se um scanner S2 detecta a vulnerabilidade V2 (e.g. SQL Injection) no sistema Web W1, então ele irá detectar esta mesma vulnerabilidade presente em um outro sistema Web W2 qualquer?

Novamente, a resposta curta é *não necessariamente*. Uma vulnerabilidade V2, como *SQL Injection*, possui várias assinaturas (ou casos de teste). Portanto, o *scanner* S2 vai detectar V2 em um outro sistema *Web* W2 somente se ele estiver programado para reconhecer todas as assinaturas de V2, ou, neste caso, as assinaturas de V2 presentes em W1 e W2. Na prática, algumas ferramentas, como o *scanner* Grabber, detectam apenas um sub-conjunto das assinaturas de *SQL injection*. Como consequência, ele pode detectar a vulnerabilidade em W1, mas não em W2, por exemplo.

A implementação de um sistema Web W1, contendo uma ou mais vulnerabilidades conhecidas, utilizando um framework de desenvolvimento Web, pode alterar os resultados de detecção das ferramentas de varredura quando comparado com o mesmo sistema W1 implementado sem a utilização de um framework?

A resposta curta é *provavelmente sim*. A maioria dos *frameworks* de desenvolvimento de sistemas *Web* possui mecanismos de segurança nativos e aplicados de forma automática e transparente. Neste trabalho, por exemplo, os *frameworks* representaram uma camada extra de segurança, impedindo a exploração de até 60% das vulnerabilidades do sistema original, implementado em PHP puro.

Qual(is) scanner(s) de vulnerabilidades é(são) mais eficaz(es) para um determinado contexto?

Observando os resultados presentes na literatura e neste trabalho, pode-se concluir que não há uma resposta única ou simples para esta pergunta. A única coisa que se pode afirmar é que, na maioria dos cenários, faz-se necessária a utilização de múltiplos *scanners* de vulnerabilidades. É recomendado ao usuário final a utilização da maior gama possível de ferramentas de varredura para detectar e diagnosticar falhas de segurança em seus sistemas *Web*. Adicionalmente, é recomendado, também, o uso de ferramentas complementares, que adiciona mais um nível de segurança ao ambiente corporativo, como os WAFs.

Os WAFs representam uma camada extra de segurança?

De acordo com os resultados deste trabalho, a resposta é *sim*. Um bom WAF pode mitigar 70% (ou mais) dos ataques a vulnerabilidades conhecidas. O melhor *framework* em termos de mecanismos de segurança, mitigou 60% das vulnerabilidades do cenário controlado utilizado neste trabalho. É importante ressaltar que, diferentemente do *framework*, o WAF é um sistema robusto cujo principal objetivo é mitigar e/ou barrar ataques. Já um *framework* oferece apenas mecanismos de segurança para evitar a exploração de uma vulnerabilidade como injeção de código.

6 CONSIDERAÇÕES FINAIS

Neste trabalho foi analisada a eficácia de diferentes ferramentas de segurança para a *Web*. A avaliação da eficácia de diferentes *scanners* de vulnerabilidades, em um ambiente controlado, reforça a necessidade de se utilizar múltiplas ferramentas para encontrar defeitos em sistemas *Web*. Além disso, os *scanners* detectaram apenas 50% das 10 vulnerabilidades mais recorrentes em sistemas *Web*. De uma maneira geral, estes resultados indicam que os *scanners* investigados, ainda carecem de pesquisa e desenvolvimento.

Outro resultado que vale a pena destacar é o fato de alguns *scanners* não detectarem algumas das vulnerabilidades que deveriam detectar segundo seus respectivos manuais e documentação técnica. A exemplo, *SQL injection*, que é uma das vulnerabilidades mais frequentes e recorrentes em sistemas *Web*, foi a vulnerabilidade onde os *scanners* mais falharam em detectar.

Como foi possível observar nos resultados apresentados, os *scanners* auxiliam na detecção de vulnerabilidades. Além de detectar de forma automatizada um conjunto grande de vulnerabilidades, geram relatórios que indicam formas de corrigir os problemas nos sistemas. A maior lição é que, quando se trata de segurança, não existe uma solução única capaz de cobrir todos os problemas. No caso dos *scanners* de vulnerabilidades, ficou claro que múltiplas ferramentas são necessárias. Além disso, é importante uma atualização constante, o que inclui avaliar novas ferramentas e reavaliar as existentes.

Com relação ao impacto dos *frameworks* de desenvolvimento de *software* na segurança de sistemas *Web*, é possível concluirmos duas coisas. Primeiro, os *frameworks* contribuem de forma significativa no que tange a impedir a exploração de vulnerabilidades conhecidas. Segundo, os resultados indicam também que há uma diferença significativa entre os *frameworks* no que diz respeito aos recursos e mecanismos de segurança. Enquanto que os *frameworks* Laravel e Symfony protegeram o sistema *Web* contra 60% das vulnerabilidades implementadas no cenário controlado, *frameworks* como o Phalcon mitigaram apenas 30% das vulnerabilidades. Isto significa que a simples escolha do *framework* de desenvolvimento de *software* pode ter um impacto significativo na segurança do sistema *Web*.

Nas análises do impacto de WAFs na segurança de sistemas *Web* quem obteve melhor desempenho foi o WAF ModSecurity, evitando que 70% das vulnerabilidades fossem detectadas nos *frameworks* Laravel e Symfony. Os WAFs Shadow Daemon e Naxsi ficaram entre 60% e 50% de porcentagem de detecção das vulnerabilidades nestes *frameworks*. Na Seção 4.2 os *frameworks* Codeigniter e Phalcon apresentaram-se mais suscetíveis nas detecções dos *scanners*. Nos testes com WAFs estes *frameworks* apresentaram-se novamente os mais suscetíveis, uma vez no melhor caso, que é do Codeigniter com o ModSecurity, foi mitigado apenas 60% das vulnerabilidades.

Finalmente, vale ressaltar que tanto os *frameworks* quanto os WAFs adicionam recursos de segurança de forma automática e transparente que ajuda a mitigar varredura

de vulnerabilidades e ataques. Entretanto, apesar disso, os resultados deixam evidente que há espaço para a pesquisa e o desenvolvimento de novos mecanismos de segurança que sejam capazes de mitigar todas as dez vulnerabilidades mais recorrentes em sistemas *Web*, por exemplo.

Trabalhos futuros. Dando continuidade à pesquisa, como trabalhos futuros, podem ser citadas atividades como:

- (a_1) avaliar *scanners* de vulnerabilidades comerciais nos cenários de *frameworks* e WAFs;
- (a_2) avaliar *scanners* oferecidos online como serviço (SaaS) nos cenários de *frameworks* e WAFs;
- (a_3) avaliar outros WAFs (incluindo os oferecidos como serviços) na segurança dos sistemas *Web*;
- (a_4) identificar mecanismos de segurança que podem ser adicionados aos *frameworks* para mitigar todas as vulnerabilidades mais recorrentes em sistemas *Web*;
- (a_5) investigar técnicas e ferramentas de varredura de vulnerabilidades que consigam burlar mecanismos de segurança dos *frameworks* e mecanismos de defesa dos WAFs;
- (a_6) realizar um mapeamento atualizado de vulnerabilidades em sites do governo e da indústria no Brasil;
- (a_7) investigar o desenvolvimento de WAFs capazes de mitigar 100% das vulnerabilidades mais recorrentes em aplicações *Web*;
- (a_8) utilizar os *scanners* de vulnerabilidades para identificar problemas em potencial em sites de prefeituras, escolas e outros órgãos públicos da região de abrangência da UNIPAMPA.

REFERÊNCIAS

- ALAM, D. et al. A case study of sql injection vulnerabilities assessment of .bd domain web applications. In: **Fourth International CyberSec**. [S.l.: s.n.], 2015. p. 73–77. Citado na página 21.
- ANTUNES, N.; VIEIRA, M. Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples. **IEEE Transactions on Services Computing**, v. 8, n. 2, p. 269–283, March 2015. ISSN 1939-1374. Citado 2 vezes nas páginas 22 e 26.
- APPELT, D. et al. A machine-learning-driven evolutionary approach for testing web application firewalls. **IEEE Transactions on Reliability**, IEEE, v. 67, n. 3, p. 733–757, 2018. Citado na página 23.
- BALDESSAR, M. J. Noticiário internacional: um mapa de contradições e influências ideológicas e econômicas¹. **Geografias da Comunicação: espaço de observação**, p. 129, 2014. Citado na página 21.
- BAU, J. et al. State of the art: Automated black-box web application vulnerability testing. In: IEEE. **Security and Privacy (SP), 2010 IEEE Symposium on**. [S.l.], 2010. p. 332–345. Citado 5 vezes nas páginas 21, 22, 23, 25 e 26.
- BECHER, M. **Web application firewalls**. [S.l.]: VDM Verlag, 2007. Citado na página 23.
- BERTOGLIO, D. D.; ZORZO, A. F. Overview and open issues on penetration test. **Journal of the Brazilian Computer Society**, v. 23, n. 1, p. 2, Feb 2017. ISSN 1678-4804. Disponível em: <<https://doi.org/10.1186/s13173-017-0051-1>>. Citado 2 vezes nas páginas 21 e 26.
- BOY, M. F.; ARANHA, D. F. Análise preliminar de segurança do sistema expresso. br. 2016. Citado 2 vezes nas páginas 24 e 29.
- CAKEPHP. **CakePHP Build fast, grow solid**. 2018. Acesso em: 2018-09-11. Disponível em: <<https://cakephp.org/>>. Citado 2 vezes nas páginas 24 e 31.
- CARVALHO, F. R. de et al. Vulnerabilidades em aplicações web. **RE3C-Revista Eletrônica Científica de Ciência da Computação**, v. 8, n. 1, 2013. Citado 2 vezes nas páginas 24 e 29.
- CODEIGNITER. **CodeIgniter Rocks**. 2018. Acesso em: 2018-09-18. Disponível em: <<https://codeigniter.com/>>. Citado 2 vezes nas páginas 24 e 31.
- COELHO, M. S. et al. Identificando vulnerabilidades de segurança em uma aplicação web. In: **SBSeg/WTICG**. [S.l.: s.n.], 2015. Citado na página 21.
- COMPUTER-WORLD. **Educação formal não é suficiente para preparar desenvolvedores**. 2017. Acesso em: 2018-04-02. Disponível em: <<http://computerworld.com.br/educacao-formal-nao-e-suficiente-para-preparar-desenvolvedores-diz-pesquisa>>. Citado na página 21.
- DAEMON, S. **Shadow Daemon Open-Source Web Application Firewall**. 2018. Acesso em: 2018-11-08. Disponível em: <<https://shadowd.zecure.org/overview/introduction/>>. Citado 2 vezes nas páginas 24 e 31.

DOUPÉ, A. et al. Enemy of the state: A state-aware black-box web vulnerability scanner. In: **USENIX Security Symposium**. [S.l.: s.n.], 2012. v. 14. Citado 2 vezes nas páginas 22 e 26.

DOUPÉ, A.; COVA, M.; VIGNA, G. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In: SPRINGER. **International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment**. [S.l.], 2010. p. 111–131. Citado 4 vezes nas páginas 22, 23, 25 e 26.

DTIC. **Coordenação de Infraestrutura e Suporte**. 2018. <<https://goo.gl/yDL57K>>. Citado na página 22.

FERRAO, I. et al. Avaliação de scanners de vulnerabilidades de aplicações web. In: **16a Escola Regional de Redes de Computadores (ERRC) e 3a Workshop Regional de Segurança da Informação e de Sistemas Computacionais (WRSeg)**. [S.l.: s.n.], 2018. <http://arxiv.kreutz.xyz/errc2018_scanners_trad_saas.pdf>. Citado na página 22.

FERRAO, I. G.; KREUTZ, D. Segurança na web: análise black-box de scanners de vulnerabilidades. In: **1a Escola Regional de Engenharia de Software (ERES)**. [S.l.: s.n.], 2017. p. 135–142. <http://arxiv.kreutz.xyz/eres2017_seguranca_web.pdf>. Citado 8 vezes nas páginas 21, 22, 23, 25, 26, 29, 37 e 38.

FONG, E. et al. Building a test suite for web application scanners. In: IEEE. **Hawaii International Conference on System Sciences, Proceedings of the 41st Annual**. [S.l.], 2008. p. 478–478. Citado 3 vezes nas páginas 22, 25 e 26.

FONSECA, J. et al. Analysis of field data on web security vulnerabilities. **IEEE Transactions on DSC**, v. 11, n. 2, p. 89–100, March 2014. ISSN 1545-5971. Citado 2 vezes nas páginas 21 e 25.

FONSECA, J.; VIEIRA, M.; MADEIRA, H. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In: IEEE. **Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on**. [S.l.], 2007. p. 365–372. Citado 3 vezes nas páginas 22, 25 e 26.

FRUTH, R. C. et al. **Passive web application firewall**. [S.l.]: Google Patents, 2017. US Patent 9,853,940. Citado na página 23.

FUNK, R.; EPP, N. et al. Anomaly-based web application firewall using http-specific features and one-class svm. **Revista Eletrônica Argentina-Brasil de Tecnologias da Informação e da Comunicação**, v. 2, n. 1, 2018. Citado 4 vezes nas páginas 21, 22, 23 e 26.

GUPTA, S.; GUPTA, B. B. Php-sensor: A prototype method to discover workflow violation and xss vulnerabilities in php web applications. In: **ACM International Conference on Computing Frontiers**. [S.l.]: ACM, 2015. (CF '15), p. 59:1–59:8. ISBN 978-1-4503-3358-0. Citado 2 vezes nas páginas 22 e 26.

HACKER, M. **Ferramentas para scan de vulnerabilidades web**. 2017. <<https://goo.gl/VNVcnP>>. Disponível em: <<http://mundodoshacker.blogspot.com.br/2013/04/ferramentas-para-scan-de.html>>. Citado na página 29.

- HARVEY, S. **Web application firewall**. [S.l.]: Google Patents, 2018. US Patent App. 15/897,670. Citado na página 23.
- HOLÍK, F.; NERADOVA, S. Vulnerabilities of modern web applications. In: IEEE. **40th MIPRO**. [S.l.], 2017. p. 1256–1261. Citado 5 vezes nas páginas 21, 22, 23, 25 e 26.
- INDIATESTBOOK. **9 Best PHP Frameworks**. 2017. Acesso em: 2018-05-06. Disponível em: <<https://www.indiatestbook.com/9-best-php-frameworks-modern-web-developers-2017/>>. Citado na página 30.
- INFOSEC. **14 Best Open Source Web Application Vulnerability Scanners**. 2017. Acesso em: 2018-05-02. Disponível em: <<http://resources.infosecinstitute.com/14-popular-web-application-vulnerability-scanners/>>. Citado 2 vezes nas páginas 24 e 29.
- JABALLAH, W. B.; KHEIR, N. A grey-box approach for detecting malicious user interactions in web applications. In: **Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats**. New York, NY, USA: ACM, 2016. (MIST '16), p. 1–12. ISBN 978-1-4503-4571-2. Disponível em: <<http://doi.acm.org/10.1145/2995959.2995966>>. Citado 3 vezes nas páginas 22, 25 e 26.
- JR, A. V. et al. Prevenção de ataques: Xss residente e sql injection em banco de dados postgresql em ambiente web. **Caderno de Estudos Tecnológicos**, v. 3, n. 1, p. 38–50, 2016. Citado na página 21.
- KINGHOST. **Frameworks PHP**. 2015. Acesso em: 2018-05-06. Disponível em: <<https://king.host/blog/2015/09/6-frameworks-php/>>. Citado na página 30.
- KINGHOST. **5 Open Source Web Application Firewall for Better Security**. 2016. Acesso em: 2018-09-06. Disponível em: <<https://geekflare.com/open-source-web-application-firewall/>>. Citado na página 31.
- LARAVEL. **Laravel**. 2018. Acesso em: 2018-09-22. Disponível em: <<https://laravel.com/>>. Citado 2 vezes nas páginas 24 e 31.
- LINKS, L. **Best Free Web Application Firewalls – Add an External Security Layer**. 2018. Acesso em: 2018-09-06. Disponível em: <<https://www.linuxlinks.com/best-free-web-application-firewalls-add-an-external-security-layer/>>. Citado na página 31.
- MACHADO, C. C. et al. Um web crawler para projeções e análise de vulnerabilidades de segurança e consistência estrutural de páginas web. **Revista de Empreendedorismo, Inovação e Tecnologia**, v. 2, n. 2, p. 3–12, 2016. Citado 2 vezes nas páginas 24 e 29.
- MAKINO, Y.; KLYUEV, V. Evaluation of web vulnerability scanners. In: IEEE. **Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 2015 IEEE 8th International Conference on**. [S.l.], 2015. v. 1, p. 399–402. Citado 5 vezes nas páginas 21, 22, 23, 25 e 26.
- MODSECURITY. **ModSecurity: Open Source Web Application Firewall**. 2018. Acesso em: 2018-11-02. Disponível em: <<https://www.modsecurity.org/>>. Citado 2 vezes nas páginas 24 e 31.

MONUS, A. **PHP Frameworks For Developers – Best of**. 2018. Acesso em: 2018-05-06. Disponível em: <<https://www.hongkiat.com/blog/best-php-frameworks/>>. Citado na página 30.

MUNIZ, J. **Web Penetration Testing with Kali Linux**. [S.l.]: Packt Publishing Ltd, 2013. Citado 2 vezes nas páginas 24 e 29.

NAGPAL, B. et al. Tool based implementation of sql injection for penetration testing. In: **International Conference on Computing, Communication Automation**. [S.l.: s.n.], 2015. p. 746–749. Citado 2 vezes nas páginas 22 e 26.

NAXSI. **NAXSI, a web application firewall for Nginx**. 2018. Acesso em: 2018-09-14. Disponível em: <<https://www.nbs-system.com/en/blog/naxsi-web-application-firewall-for-nginx/>>. Citado 2 vezes nas páginas 24 e 31.

ONAIRCODE. **Free Best PHP Frameworks for 2018**. 2017. Acesso em: 2018-05-06. Disponível em: <<https://onaircode.com/free-best-php-frameworks/>>. Citado na página 30.

OWASP. **Top Ten 2013 Project**. 2013. <https://www.owasp.org/index.php/Top_10-2017_Top_10>. Disponível em: <https://www.owasp.org/index.php/Top_10-2017_Top_10>. Citado 2 vezes nas páginas 24 e 25.

PALSETIA, N. et al. Securing native xml database-driven web applications from xquery injection vulnerabilities. **Journal of Systems and Software**, v. 122, p. 93 – 109, 2016. ISSN 0164-1212. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0164121216301571>>. Citado 2 vezes nas páginas 22 e 26.

PELLEGRINO, G.; BALZAROTTI, D. Toward black-box detection of logic flaws in web applications. In: **USENIX NDSS**. [S.l.: s.n.], 2014. Citado 2 vezes nas páginas 22 e 26.

PHALCON. **Phalcon**. 2018. Acesso em: 2018-09-21. Disponível em: <<https://phalconphp.com/pt/>>. Citado 2 vezes nas páginas 24 e 31.

RedMonk. **The RedMonk Linguagem de Programação**. 2014. Acesso em: 2018-11-12. Citado na página 31.

ROCHA, D.; KREUTZ, D.; TURCHETTI, R. A free and extensible tool to detect vulnerabilities in web systems. In: **7th Iberian CISTI**. [S.l.: s.n.], 2012. Citado 7 vezes nas páginas 21, 22, 23, 24, 25, 26 e 29.

SCHWALENBERG, D. **Web Applications Under Attack: Tenable.io and the 2017 Verizon DBIR**. 2017. <https://www.tenable.com/blog/web-applications-under-attack-tenable-io-and-the-2017-verizon-dbir>. Citado na página 21.

SHI, H.-z.; CHEN, B.; YU, L. Analysis of web security comprehensive evaluation tools. In: IEEE. **Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second International Conference on**. [S.l.], 2010. v. 1, p. 285–289. Citado 2 vezes nas páginas 24 e 29.

Symantec. **Internet Security Threat Report**. 2017. <<https://goo.gl/iuhLPX>>. Disponível em: <<https://www.symantec.com/security-center/threat-report>>. Citado 2 vezes nas páginas 21 e 24.

- SYMFONY. **Symfony, High Performance**. 2018. Acesso em: 2018-09-02. Disponível em: <<https://symfony.com/>>. Citado na página 31.
- TEODORO, N.; SERRÃO, C. Automating web applications security assessments through scanners. **Web Application Security**, p. 48, 2011. Citado 2 vezes nas páginas 24 e 29.
- Terminal Root. **100 Melhores Ferramentas Open Source de Segurança**. 2017. <<https://goo.gl/9ksoXR>>. Disponível em: <<http://terminalroot.com.br/2017/02/100-melhores-ferramentas-de-seguranca-de-codigo-aberto.html>>. Citado na página 29.
- VIEIRA, M.; ANTUNES, N.; MADEIRA, H. Using web security scanners to detect vulnerabilities in web services. In: IEEE. **Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on**. [S.l.], 2009. p. 566–571. Citado 4 vezes nas páginas 21, 22, 25 e 26.
- WILMSHURST, J. F.; FRYXELL, J. M.; HUDSONB, R. J. Forage quality and patch choice by wapiti (*cervus elaphus*). **Behavioral Ecology**, Oxford University Press, v. 6, n. 2, p. 209–217, 1995. Citado 2 vezes nas páginas 24 e 29.
- YII. **Yii PHP Framework**. 2018. Acesso em: 2018-09-16. Disponível em: <<https://www.yiiframework.com/>>. Citado 2 vezes nas páginas 24 e 31.
- ZEND. **Zend Framework**. 2018. Acesso em: 2018-09-13. Disponível em: <<https://framework.zend.com/>>. Citado 2 vezes nas páginas 24 e 31.
- ZHOU, Y.; EVANS, D. SSOScan: Automated testing of web applications for single sign-on vulnerabilities. In: **23rd USENIX Security Symposium**. [s.n.], 2014. ISBN 978-1-931971-15-7. Disponível em: <<https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zhou>>. Citado 2 vezes nas páginas 22 e 26.

ANEXO A – CÓDIGOS DO CENÁRIO CONTROLADO

INDEX.php

```

<!DOCTYPE html>
<html>
<head>
<title>Sistema de testes </title>
</head>

<body>
<div class="form-log-in-with-email">
<form class="form-login" method="POST" action="cadastra.php">
<div class="form-white-background">
<div class="form-title-row">
<h1>Formulário de cadastro</h1>
</div>
<div class="form-row">
<label>
<span>Nome</span>
<input type="text" name="nome" placeholder="Isadora Ferrão">
</label>
</div>

<div class="form-row">
<label>
<span>Password</span>
<input type="password" name="senha" required>
</label>
</div>

<div class="form-row">
<button type="submit">Cadastrar</button>
</div>
</div>
</form>
</div>

<form class="form-login" method="post" action="verifica_usuario.php ">
<div class="form-white-background" style="float:left; margin-left:200px">

```

```

<div class="form-title-row">
<h1>Formulário de login </h1>
</div>
<div class="form-row">
<label>
<span>nome</span>
<input type="text" name="nome" required>
</label>
</div>
<div class="form-row">
<label>
<span>Password</span>
<input type="password" name="senha" required>
</label>
</div>
<div class="form-row">
<button type="submit">Entrar</button>
</div>
</div>
</form>

<iframe src="https://www.google.com/maps/embed?pb=!1m14!1m12!1m3!1d3462.
587503728983!2d-55.77078578489995!3d-29.789566926428762!2m3!1f0!2f0!3f0!3m2!
1i1024!2i768!4f13.1!5e0!3m2!1spt-BR!2sbr!4v1508864877378" width="400"
height="300" frameborder="0" style="border:0" allowfullscreen></iframe>
</body>
</html>

```

CADASTRA.php

```

<?php

include "conexao.php" ;

$nome = $_POST['nome'];
$senha = $_POST['senha'];
$strx = $senha;
$senhaencriptada = base64_encode($strx);

```

//SQL que faz a inserção no banco dos dados recebidos!

```

    $sql = mysqli_query($conexao, "INSERT INTO usuarios (nome, senha)VALUES
    ('$nome','$senhaencryptada')") or die ('Erro ao inserir os dados'.
    mysqli_error($conexao));

    echo "Cadastro efetuado com sucesso!";
    header("Location:logado.php?nome=$nome");

?>
<html>
<head>
<title>Redirecionando</title>
<script type="text/javascript"> alert("Cadastrado com sucesso!");
window.location="applications.php"; </script>
</head>
<body>
</body>
</html>

```

CONEXAO.php

```

<?php
define('BD_USER', 'root');
define('BD_PASS', '');
define('BD_NAME', 'vulnerabilidades');
$conexao = mysqli_connect('localhost', 'root', '');
$link = mysqli_select_db($conexao, BD_NAME);
?>

```

VERIFICA.php

```

<?php
require('conexao.php');

```

```
if(!empty($_POST)){
$nome = $_POST['nome'];
$senha = $_POST['senha'];
$error = '';

$senhaencriptada = base64_encode($senha);

$result = mysqli_query ($conexao, "SELECT * FROM usuarios WHERE nome = '$nome'
AND senha = '$senhaencriptada' ");
#echo $result;
#$result=mysqli->query($sql);
$rows = $result->num_rows;

if($rows > 0){
$row = $result->fetch_assoc();
$_SESSION['nome'] = $row['nome'];
$_SESSION['senha'] = $row['senha'];
$_SESSION['nivel_usuario'] = $row['nivel_usuario'];

if($_SESSION['nivel_usuario'] == 0){
header("Location:logado.php?nome=$nome");

}
if($_SESSION['nivel_usuario'] == 1){
header("Location:inicio.php?nome=$nome");
}

}
else{
echo 'erro';
}
}

?>
```

LOGADO.php

<?php

```
session_start();
?>
<?php

if(!isset($_GET["nome"])){
echo "<center><h1>Página não encontrada!</h1></center>";
exit();
}
$nome=$_GET["nome"];
?>

<!DOCTYPE html>
<html>
<head>
<title>Sistema de testes </title>
</head>
<body>
<h1>Logado</h1>
<h2>Seja bem vindo, <?php echo $nome;?></h2>
</body>
</html>
```

INDEXLOGADO.php

```
<?php
session_start();
?>
<?php
if(!isset($_GET["nome"])){
echo "<center><h1>Página não encontradaaaa!</h1></center>";
exit();
}
$nome=$_GET["nome"];
?>

<!DOCTYPE html>
<html>
```

```
<head>
<title>Sistema de testes </title>
</head>

<body>

<h1>Logado com nível de acesso 1</h1>
<h2>Seja bem vindo, <?php echo $nome;?></h2>

<form method="GET" action="busca.php">
<label for="consulta">Buscar:</label>
<input type="submit" value="OK"/>
</form>
</body>
</html>
```