UNIVERSIDADE FEDERAL DO PAMPA

Felipe Homrich Melchior

WAFCheck: uma Ferramenta para Auxiliar na Avaliação de Web Application Firewalls (WAFs)

Alegrete
2021

Felipe Homrich Melchior

WAFCheck: uma Ferramenta para Auxiliar na Avaliação de Web Application Firewalls (WAFs)

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.

Orientador: Prof. Dr. Diego Kreutz

Coorientador: Maurício Fiorenza

Alegrete
2021

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
Universidade Federal do Pampa

Felipe Homrich Melchior

WAFCheck: uma Ferramenta para Auxiliar na Avaliação de Web Application Firewalls (WAFs)

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

Trabalho de conclusão de curso defendido e aprovado em: 07, maio de 2021.

Banca examinadora:

_____

Prof. Dr. Diego Kreutz
Orientador
UNIPAMPA

Maurício Fiorenza

Co-orientador

UNIPAMPA

Prof. Dr. Érico Marcelo Hoff do Amaral

UNIPAMPA

Prof. Dr. Rodrigo Mansilha

UNIPAMPA

Assinado eletronicamente por **RODRIGO BRANDAO MANSILHA**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 12/05/2021, às 11:51, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **MAURICIO MARTINUZZI FIORENZA**, **ANALISTA DE TECNOLOGIA DA INFORMACAO**, em 12/05/2021, às 11:55, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **ERICO MARCELO HOFF DO AMARAL**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 12/05/2021, às 15:29, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **DIEGO LUIS KREUTZ**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 12/05/2021, às 22:58, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **0518963** e o código CRC **6D0E2856**.

Ao meu inexorável refúgio,
Pai, Mãe e Irmão.

RESUMO

Falhas de segurança em sistemas *web* são perigosas e recorrentes. Estatísticas apontam que até 90% das aplicações disponibilizadas na Internet podem possuir algum tipo de vulnerabilidade de software. Garantir a segurança desses sistemas online é crucial e pode contribuir para evitar prejuízos financeiros e vazamento de dados. Esse assunto merece especial atenção com a entrada em vigor de leis como a Lei Geral de Proteção de dados (LGPD). Entretanto, proteger as aplicações online não é simples. Estudos indicam que estratégias como *frameworks* de desenvolvimento podem ajudar a proteger o sistema de até 60% das vulnerabilidades mais frequentes na *web*. Similarmente, estatísticas indicam também que Web Application Firewall (WAF) podem contribuir para mitigar a exploração de mais de 70% das vulnerabilidades de sistemas *web*. Neste trabalho, propomos a ferramenta WAFCheck para auxiliar nos testes de latência, carga e acurácia de detecção de WAFs. A ferramenta permite a inclusão e avaliação de conjuntos diversos de *payloads*, que são utilizados na exploração de vulnerabilidades dos sistemas *web*. Com o auxílio da WAFCheck e de *payloads* das dez vulnerabilidades mais recorrentes segundo a OWASP, realizamos a avaliação dos WAFs gratuitos ModSecurity, Naxsi, ShadowD e xWAF. Os resultados indicam que os WAFs avaliados, em configuração padrão, podem apresentar uma taxa de detecção de cerca de 70% dos ataques. Entretanto, um grande número de regras ativas no WAF pode impactar de sobremaneira (*e.g.*, aumentar em mais de 2685%) a latência das requisições aos sistemas *web*.

Palavras-chave: Segurança da Informação. Web Application Firewalls. Segurança de Sistemas Web.

ABSTRACT

Security breaches in web systems are dangerous and recurring. Statistics indicate that up to 90% of applications on the Internet may have software vulnerabilities. Ensuring the security of these online systems is crucial and can help prevent financial loss and data leakage. This issue deserves special attention with the enforcement of laws such as the General Data Protection Law (LGPD). However, protecting online applications is not an easy task. Studies indicate that software development frameworks can help protect systems against attacks exploring up to 60% of the most frequent vulnerabilities on the web. Similarly, statistics also suggest that a Web Application Firewall (WAF) can contribute to mitigating the exploitation of more than 70% of vulnerabilities in web systems. In this work, we propose the WAFCheck tool to assist in the latency, load, and accuracy detection tests for WAFs. The tool allows the insertion and evaluation of different sets of payloads, which are used to exploit vulnerabilities in web systems. Using WAFCheck and payloads of the ten most recurring vulnerabilities according to OWASP, we evaluated four free WAFs, namely, ModSecurity, Naxsi, ShadowD, and xWAF. Our findings suggest that free WAFs, in standard configuration, achieve a nearly 70% detection rate. However, a large number of active rules in the WAF can greatly impact (e.g., increase by more than 2685%) the latency of requests to web systems.

Key-words: Information Security. Web Application Firewall. Web Systems Security.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

CRS Core Rule Set

CSRF Cross-site Request Forgery

DNS Domain Name Service

HTTP HyperText Transfer Protocol

ICMP Internet Control Message Protocol

LFI Local File Inclusion

LGPD Lei Geral de Proteção de dados

RFI Remote File Inclusion

SaaS Security-as-a-Service

SSRF Server Side Request Forgery

WAF Web Application Firewall

XML Extensible Markup Language

XSS Cross-site scripting

XXE XML External Entity

YML YAML Ain't Markup Language

SUMÁRIO

1 INTRODUÇÃO

Estudos apontam que cerca de 50% das aplicações *web* disponíveis na Internet possuem pelo menos uma vulnerabilidade de alta criticidade, como *SQL Injection*, e quando consideramos o nível de risco médio, cerca de 90% das aplicações são vulneráveis (ACUNETIX, 2019). Segundo relatórios especializados, a vulnerabilidade de Cross-site scripting (XSS) é uma das mais comuns e mais exploradas (representando cerca de 30% dos ataques) em aplicações *web* (HACKERONE, 2019). Além de ser frequente, em alguns casos, a exploração da vulnerabilidade XSS permite que o atacante obtenha *cookies* de sessão de usuários, podendo assim, acessar recursos e dados privados do sistema (CIMPANU, 2019).

Uma das formas de mitigar o impacto de vulnerabilidades antigas e recorrentes, como XSS e *SQL Injection*, é através da utilização de *frameworks* de desenvolvimento de *software* como o Laravel[1], que implementa e oferece recursos de segurança que protegem, de forma automática e transparente, as aplicações *web* PHP contra falhas de código. Um *framework* como o Laravel é capaz de, por si só, mitigar a exploração de até 60% das vulnerabilidades mais recorrentes em aplicações PHP (FERRAO; MACEDO; KREUTZ, 2018).

Outro tipo de ferramenta que tem sido utilizada na proteção de aplicações *web* é WAF. Um WAF é um serviço de segurança implementado entre o cliente (*e.g.* navegador/*browser*) e a aplicação (*e.g.* sistema PHP em um servidor *web* Apache). A função do WAF é interceptar e processar as requisições entre o cliente e a aplicação. A partir de um conjunto de regras, o WAF classifica as requisições em maliciosas (que geralmente são bloqueadas) e não-maliciosas, isto é, que são encaminhadas até a aplicação. Um WAF como ModSecurity[2], associado a um *framework* PHP como o Laravel, é capaz de mitigar a exploração de vulnerabilidades recorrentes em até 70% (FERRAO; MACEDO; KREUTZ, 2018).

Contra intuitivamente, a adição de um WAF pode, também, piorar a segurança de um sistema *web*. Pesquisas recentes apresentam casos onde a adição de um WAF ocasiona uma queda (ao invés de um aumento) no nível de proteção da aplicação *web*. Esse é o caso do cenário utilizando o *framework* Symfony[3], utilizado para o desenvolvimento de aplicações *web* PHP, e os WAFs Naxsi[4] e ShadowDaemon[5]. Enquanto o Symfony mitiga até 60% das vulnerabilidades da aplicação *web*, a adição dos WAFs Naxsi e ShadowDaemon reduz a mitigação para 50% (FERRAO, 2018). Isso significa que houve uma queda de 10% na segurança da aplicação *web* ao adicionar os WAFs.

Os WAFs existentes no mercado podem ser classificados em dois grandes grupos:

[1] https://laravel.com
[2] https://modsecurity.org
[3] https://symfony.com
[4] https://github.com/nbs-system/naxsi
[5] https://shadowd.zecure.org

os que funcionam como ferramentas *standalone*; e os Security-as-a-Service (SaaS). Os WAFs *standalone* são geralmente instalados junto aos servidores *web*, em que as aplicações estão sendo executadas, os WAFs SaaS são oferecidos como serviços de segurança sob demanda por terceiros. ModSecurity e Naxsi são dois exemplos WAFs *standalone* gratuitos, enquanto que o Imperva[6] e o Citrix[7] são comerciais. Os WAFs SaaS são essencialmente *online* e oferecidos por empresas como a Cloudflare[8] e a X Labs ON Security[9], atuando como um *proxy* de redirecionamento, em que a requisição é processada antes de ser encaminhada para o servidor de aplicação propriamente dito.

Estudos vêm identificando diferentes desafios e oportunidades de pesquisa no contexto de *Web Application Firewalls*, como algoritmos para detectar ataques que objetivam explorar vulnerabilidades específicas (e.g. Cross-site Request Forgery (CSRF)) e XSS) (SROKOSZ; RUSINEK; KSIEZOPOLSKI, 2018; RAO; PRASAD; RAMESH, 2016), mecanismos para aumentar o desempenho de processamento de requisições em cenários com grandes volumes de requisições (*e.g.* milhares de requisições por segundo) (MOOSA; ALSAFFAR, 2008) e revisão minuciosa do estado da arte, procurando comparar e entender o funcionamento dos diferentes WAFs existentes (CLINCY; SHAHRIAR, 2018; RAZZAQ et al., 2013). Na prática, até onde se sabe, há uma carência de ferramentas e trabalhos que investiguem WAFs de forma empírica, identificando aspectos importantes como o impacto do número de regras do WAF na latência das requisições aos sistemas Web e o impacto das opções de configuração da ferramenta na acurácia da detecção de ataques.

Neste trabalho propomos uma ferramenta, denominada WAFCheck, cujo objetivo é auxiliar o processo de avaliação de WAFs no que diz respeito ao impacto da latência e acurácia na detecção de vulnerabilidades em sistemas *web*. A ferramenta utiliza uma base de *payloads* para executar requisições (sequenciais ou paralelas) aos sistemas *web* através dos WAFs. A base atual de *payloads* contém um conjunto de *payloads* para cada uma das dez vulnerabilidades mais recorrentes em aplicações *web* segundo a organização OWASP (OWASP, 2020).

A ferramenta WAFCheck foi utilizada para avaliar o impacto da quantidade de regras ativas na latência das comunicações entre clientes e servidores *web*. Avaliamos a latência gerada pelos WAFs ModSecurity, Naxsi, ShadowD e xWAF com um número variável de regras (da configuração padrão até 100.000). Os resultados indicam que acima de 1.000 regras há um aumento significativo na latência de comunicação para todos os WAFs, isso é, a um mecanismo de segurança desse tipo pode acarretar em uma penalidade nas requisições aos sistemas *web* (*e.g.*, aumentar de 2,53ms para 46,73ms a latência média de cada requisição HTTP utilizando o método GET). Adicionalmente, avaliamos também

[6] https://www.imperva.com/products/web-application-firewall-waf/
[7] https://www.citrix.com.br/products/citrix-web-app-firewall/
[8] https://www.cloudflare.com/pt-br/waf/
[9] https://www.xlabs.com.br/waf/

a acurácia dos mecanismos na detecção dos WAFs.

As principais contribuições do trabalho são: (a) proposta, projeto e implementação da ferramenta WAFCheck, que auxilia usuários na avaliação da latência e acurácia de WAFs; (b) avaliação do impacto da quantidade de regras na latência das requisições entre os clientes e o servidor *web*; e (c) avaliação da acurácia de detecção de *payloads* maliciosos nos WAFs gratuitos ModSecurity, Naxsi, ShadowD e xWAF.

O restante do trabalho está organizado como segue. No Capítulo 2, revisamos o estado da arte no contexto de WAFs. A arquitetura e a implementação da ferramenta WAFCheck são discutidas no Capítulo 3. Por fim, os Capítulos 4 e 5 apresentam os resultados e as considerações finais do trabalho, respectivamente.

2 ESTADO DA ARTE

As soluções de WAFs são classificados em dois tipos. A primeira, dos WAFs *standalone* são geralmente instalados junto ao serviço, e aplicam suas regras nas conexões entre o cliente e a aplicação. O segundo tipo, os WAFs SaaS, utilizam redirecionamento Domain Name Service (DNS)[1] no qual o tráfego é processado nos servidores da empresa proprietária do WAF antes de ser encaminhado ao servidor do sistema *web*.

Os WAFs *standalone* apresentam diferentes desafios, que requerem recursos humanos minimamente qualificados, como: (i) instalação, configuração e manutenção; (ii) gerenciamento de regras; (iii) otimização de regras; e (iv) suporte a grandes volumes de tráfego. Os WAFs SaaS eliminam esses desafios pelo fato de serem oferecidos por terceiros, como um serviço. Porém, utilizar um WAF SaaS implica, também, em um contrato de confiança entre o contratante e a contratada, pois o tráfego do contratante irá passar pelos sistemas da contratada. Em outras palavras, a privacidade dos dados torna-se uma questão delicada. Qualquer pessoa, com acesso a infraestrutura da contratada, é potencialmente capaz de realizar uma interceptação de tráfego, por exemplo.

A Tabela 1 resume as principais contribuições, oportunidades de pesquisa e evidências empíricas encontradas em trabalhos relacionados à WAFs. Com relação à Principal Contribuição, há propostas de novos algoritmos para detecção de ataques a vulnerabilidades específicas (e.g. CSRF e XSS). Por exemplo, ataques que visam explorar vulnerabilidades XSS podem ser bloqueados através da análise e identificação de padrões (e.g. caracteres "<" e ":") frequentemente utilizados em requisições maliciosas aos sistemas *web* (RAO; PRASAD; RAMESH, 2016).

Outro ponto importante são as boas práticas na criação de novas regras para o WAF. O primeiro passo é entender o objetivo da regra que está sendo criada (CLINCY; SHAHRIAR, 2018). Isso é crucial para o bom funcionamento e desempenho de um WAF. Por exemplo, uma regra estática, que bloqueia ataques de SQL *Injection* que utilizam a expressão "' OR 1=1– #", não irá bloquear um ataque que utilize a expressão "' OR 2=2– #".

No contexto de Oportunidades de Pesquisa, há estudos focados em reunir informações, através de revisões da literatura, sobre as propostas de diversos mecanismos de defesa, incluindo WAFs, destacando realizando uma análise crítica sobre as soluções, elencando pontos negativos (*e.g.* ModSecurity não consegue detectar ataques de força bruta em IDs de sessões) e positivos dos mecanismos (desempenho positivo das soluções Barracuda, F5 e SecurSphere) (RAZZAQ et al., 2013). No entanto, não foram realizadas análises visando a taxa de acurácia na detecção de cada uma dessas soluções.

Evidências Empíricas são dados concretos encontrados pelos estudos e são cruciais para validar uma pesquisa. Por exemplo, o ModSecurity (na configuração padrão) em um cenário controlado pode chegar a 70% de taxa de detecção (FERRAO, 2018).

[1] <https://www.pcmag.com/encyclopedia/term/41620/dns>

Tabela 1 – Implementação e Avaliação de WAFs

	Principal Contribuição	**Oportunidades de Pesquisa**	**Evidências Empíricas**
(MOOSA; ALSAFFAR, 2008)	WAF híbrido para aplicações governamentais	Evoluir o WAF com novas heurísticas	Suporta grandes volumes de requisições
(RAZZAQ et al., 2013)	Comparação analítica de quinze WAFs tradicionais	Comparar de forma empírica os WAFs e soluções *SaaS*	-
(PRANDL; LAZARESCU; PHAM, 2015)	*Benchmark* das solução ModSecurity, WebKnight e Guardian	Efetuar o mesmo teste em outros WAFs	ModSecurity apresenta o melhor desempenho
(RIETZ et al., 2016)	Visão de WAF modernos	Avaliar WAFs em cenários controlados	-
(RAO; PRASAD; RAMESH, 2016)	Detecção de ataques *XSS*	Implementar e avaliar os algoritmos propostos	-
(FUNK; EPP; A., 2018)	Melhora na taxa de detecção de ataques	Comparar com outros WAFs, além do ModSecurity	Taxa de detecção 60% maior que a do ModSecurity
(SROKOSZ; RUSINEK; KSIEZOPOLSKI, 2018)	Detecção de ataques *CSRF*	Implementar e avaliar os algoritmos propostos	-
(CLINCY; SHAHRIAR, 2018)	Boas práticas para criação de regras de WAFs	Avaliar configurações padrão de WAFs	-
(SINGH; Samuel; Zavarsky, 2018)	Análise dos níveis de Paranoia do ModSecurity	Avaliar diferentes configurações	Os níveis de Paranoia aumentam a taxa de detecção e os falsos positivos
(FERRAO, 2018)	Avaliação dos WAFs ModSecurity, Naxsi e ShadowDaemon em cenários controlados	Avaliar outros WAFs e investigar WAFs para mitigar as vulnerabilidades mais recorrentes	ModSecurity consegue mitigar 70% das vulnerabilidades do Top Ten da OWASP (OWASP, 2013)

Fonte: Elaborado pelo autor.

Alguns dos principais desafios encontrados na utilização de *Web Application Firewalls* são identificar ataques complexos de detectar, como CSRF, manter um bom desempenho em cenários adversos (*i.e.*, com um grande volume de requisições) e possuir

conhecimentos avançados sobre ferramenta a fim de otimizar o processo de detecção. Por exemplo, é recomendado compreender e utilizar os níveis de Paranoia, mecanismo de ativação de regras adicionais do ModSecurity (SINGH; Samuel; Zavarsky, 2018).

Detecção de Ataques. Falhas que exploram as solicitações ao sistema, como CSRF e Server Side Request Forgery (SSRF), são difíceis de detectar e impedir a exploração (SROKOSZ; RUSINEK; KSIEZOPOLSKI, 2018). Para detectar ataques CSRF, um WAF pode utilizar técnicas como utilização de autenticação adicional necessária ou até mesmo de algoritmos que analisam similaridades nas requisições de cada usuário de acordo com seu histórico.

Desempenho. Em alguns cenários, o desempenho de WAFs é crítico devido a quantidade de tráfego a ser processada (MOOSA; ALSAFFAR, 2008). Utilizando uma arquitetura híbrida com heurísticas de inteligência artificial e redes neurais é possível melhorar o desempenho desses WAFs nesse tipo de caso.

Estudos com objetivo de realizar testes de acurácia nos mecanismos mostram que a solução gratuita ModSecurity apresenta uma performance melhor em geral quando comparado com as outras soluções, como Naxsi e WebKnight (PRANDL; LAZARESCU; PHAM, 2015; FERRAO; MACEDO; KREUTZ, 2018). O WAF foi colocado à prova ao realizar detecções de diversos ataques, como XSS, *SQL Injection*, Remote File Inclusion (RFI) e Local File Inclusion (LFI), e em ambos os estudos, o resultado obtido pela solução foi o mesmo, alcançando até 70% de acurácia.

Domínio da ferramenta. Explorar as diferentes configurações de um WAF é importante para atingir uma melhor eficiência de detecção. Tomando como exemplo o ModSecurity, a solução possui os chamados níveis de Paranoia, que são tipos de regras ativadas apenas em momentos específicos, como uma da detecção de uma sequência de ataques, por exemplo (SINGH; Samuel; Zavarsky, 2018). No nível mais baixo, é garantido que o número de falsos positivos seja menor, porém a taxa de detecção também será menor. Ao ativar o nível mais alto de Paranoia, todas as regras da configuração são ativadas ao mesmo tempo, detectando um maior número de ataques, porém levando a um aumento na taxa de falsos positivos.

Estudos recentes indicam que a combinação de *frameworks* como Laravel e Symfony e WAFs como ModSecurity possibilitam o alcance 70% de cobertura das vulnerabilidades em cenários controlados (FERRAO, 2018). Entretanto, combinações como *framework* Symfony e WAFs Naxsi e ShadowDaemon acabam gerando resultados contraintuitivos, onde a segurança piorou de 60% (sem WAFs) para 50% (com WAFs).

3 WAFCHECK

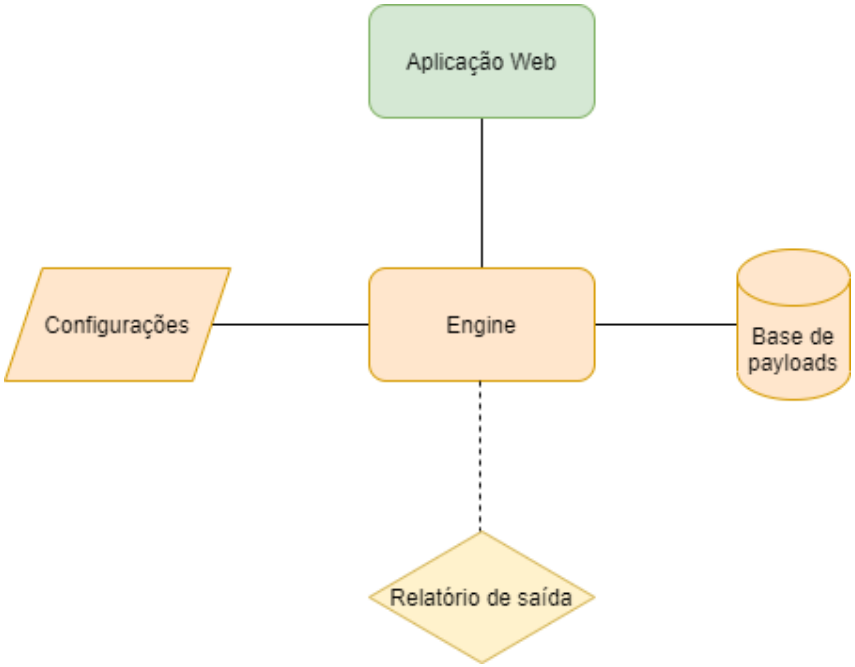
Neste capítulo apresentados a arquitetura da ferramenta WAFCheck na Seção 3.1. Na sequência (Seção 3.2), discutimos detalhes de implementação de uma instância da arquitetura.

3.1 Arquitetura

A WAFCheck é uma ferramenta projetada para automatizar o envio de requisições para uma aplicação *web* protegida por um WAF. Além de automatizar, a ferramenta permite avaliar a latência das requisições e a taxa de acurácia dos mecanismos de defesa utilizados no WAF.

A Figura 1 ilustra a arquitetura proposta para a WAFCheck. Na arquitetura há três módulos principais: *engine*, base de *payloads* e configurações. Na base de *payloads* contém listas de *payloads*, catalogados por tipo de vulnerabilidade, que são utilizados nos testes de latência das requisições e na avaliação da acurácia dos mecanismos de detecção do WAF. As listas de *payloads* podem ser criadas a partir de categorias conhecidas de vulnerabilidades, como as mais recorrentes em aplicações *web*, segundo a OWASP, no ano de 2020 (OWASP, 2020).

Figura 1 – Arquitetura - WAFCheck



Fonte: Elaborado pelo autor.

No módulo de configurações são definidas opções de execução que balizam o funcionamento de algumas funcionalidades da ferramenta. Por exemplo, nas configurações

pode ser incluída a definição de *templates* (ou mensagens personalizadas) de bloqueio do WAF. Em alguns casos, embora o código de retorno da requisição HyperText Transfer Protocol (HTTP) seja 200 (indicando sucesso), o WAF pode ter, de fato, bloqueado a requisição. As mensagens personalizadas contribuem para identificar todas as requisição HTTP bloqueadas pelo WAF. Um exemplo prático é o WAF Naxsi, que permite a configuração de uma página personalizada de bloqueio (*e.g.*, *status* HTTP 200). Quando a requisição do usuário for bloqueada, a página personalizada será apresentada para o usuário, como no exemplo do Algoritmo 2.

Algorithm 1 Exemplo de página bloqueada

```
1: <html>
2:     <body>
3:         <h1>Requisição maliciosa bloqueada</h1>
4:     </body>
5: </html>
```

É possível então, definir que o WAFCheck busque na resposta da requisição algum padrão encontrado nesta página de bloqueio. Neste caso, o conteúdo "<h1>Requisição maliciosa bloqueada</h1>" poderia ser utilizado como "*template*", fazendo com que o programa identifique a requisição bloqueada, mesmo que o código HTTP indique sucesso.

Se uma requisição retornar qualquer código HTTP diferente de 200 ou for encontrado algum dos *templates* de bloqueio no conteúdo da resposta, então o WAFCheck irá considerar que aquela requisição maliciosa foi detectada com sucesso pelo mecanismo. Caso contrário, o programa irá armazenar esse *payload* em uma lista de requisições maliciosas que foram aceitas pelo WAF. Estes resultados são apresentados para o usuário no final da execução do programa.

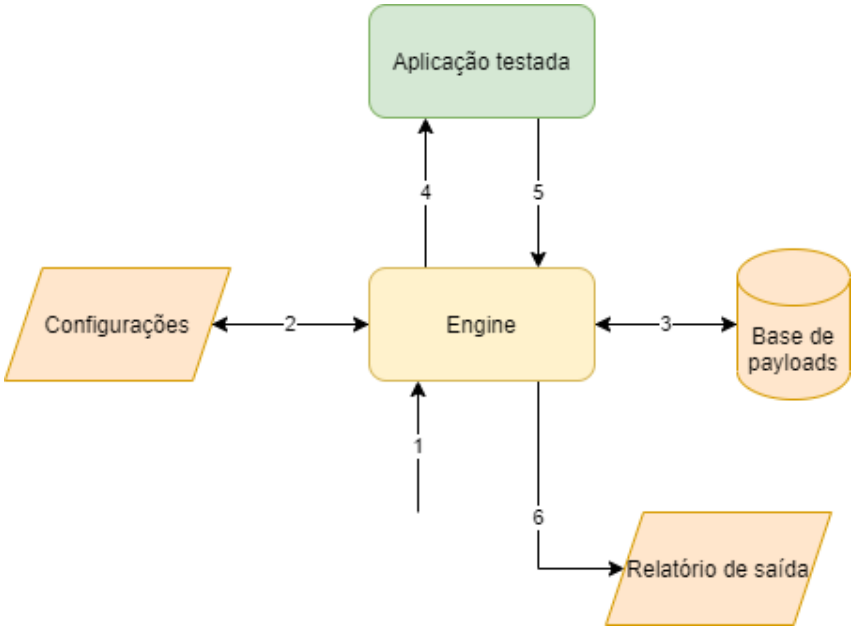
A Figura 2 exemplifica o funcionamento da ferramenta durante um teste válido. A lista seguir enumera cada etapa, representada por números na figura, do fluxo de um teste realizado utilizando a ferramenta WAFCheck:

1. Um teste válido realizado pelo WAFCheck começa com a inserção de parâmetros que serão utilizados durante o processo, isso inclui o *host* a ser testado e a lista de *payloads* usada.

2. Além dos parâmetros, definidos durante a execução, a ferramenta possui a possibilidade de determinar opções adicionais, como diretório da base de *payloads* e *templates* de requisições bloqueadas.

3. Com a configuração do diretório de *payloads* realizada com sucesso, esse diretório então é enumerado, buscando todas as possíveis listas a serem utilizadas no teste. Tudo ocorrendo bem, a lista definida na Etapa 1 será carregada em memória.

Figura 2 – Funcionamento - WAFCheck



Fonte: Elaborado pelo autor.

4. A cada *payload* carregado em memória, uma requisição HTTP é feita para a aplicação.

5. Cada requisição traz uma resposta diferente, tendo esses dados, o WAFCheck realiza uma análise para compreender se aquela requisição foi bloqueada pelo WAF ou não. Em caso negativo, a carga usada é armazenada para ser mostrada ao usuário.

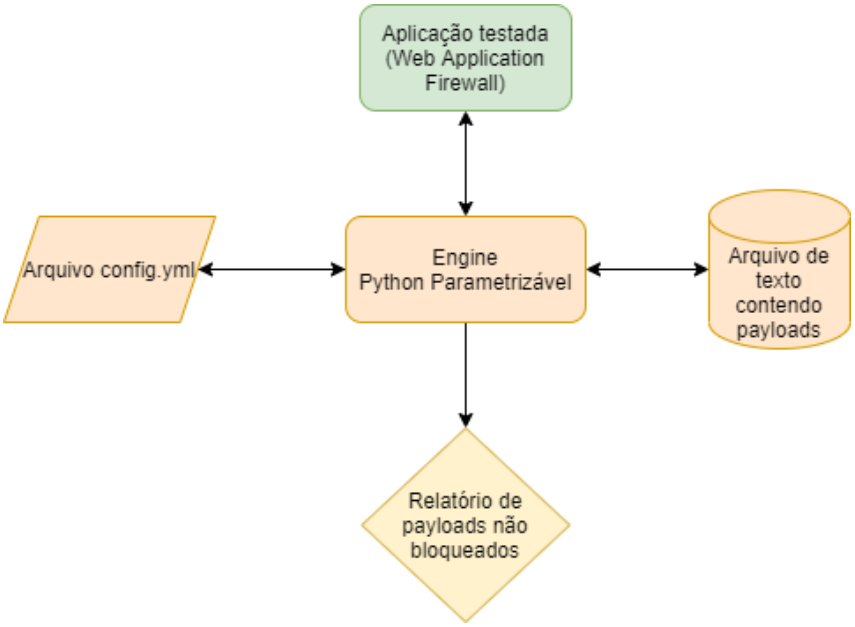
6. Após a análise de todas as requisições, o resultado do teste é exibido na tela do usuário, gerando também um arquivo que contém a lista de *payloads* não detectados pelo mecanismo.

3.2 Implementação

Dada a arquitetura apresentada, a próxima etapa do desenvolvimento da ferramenta foi a sua implementação de fato, utilizando-se a linguagem de programação Python. A escolha da linguagem se deu por conta da grande quantidade de bibliotecas já disponíveis na Internet, além de seguir o movimento dos últimos anos, onde a linguagem Python é amplamente utilizada para a construção de ferramentas focadas em segurança (ECCOUNTIL, 2021).

Como comentado anteriormente, a ferramenta conta três módulos para o seu funcionamento. O módulo chamado de *engine* (i) é o responsável por interagir com a aplicação *web*, através de requisições usando o protocolo HTTP. Além disso, este módulo tam-

Figura 3 – Esquema de implementação - WAFCheck



Fonte: Elaborado pelo autor.

bém gera o relatório que é apresentado para o usuário ao final da execução. Para que o WAFCheck envie seus *payloads* para a aplicação, foi utilizada a biblioteca Requests.

Já para a base de *payloads* (ii) foram utilizados arquivos de texto que agrupam séries de cargas úteis, de diferente tipos de ataques. De forma padrão (*i.e.* execução sem nenhuma modificação), o programa possui dez listas de *payloads*, agrupados com base na OWASP Top Ten de 2020 (OWASP, 2020). Novas listas são facilmente adicionadas ao programa, bastando apenas acrescentar um novo arquivo de texto no diretório configurado para armazenar os *payloads* que serão utilizados.

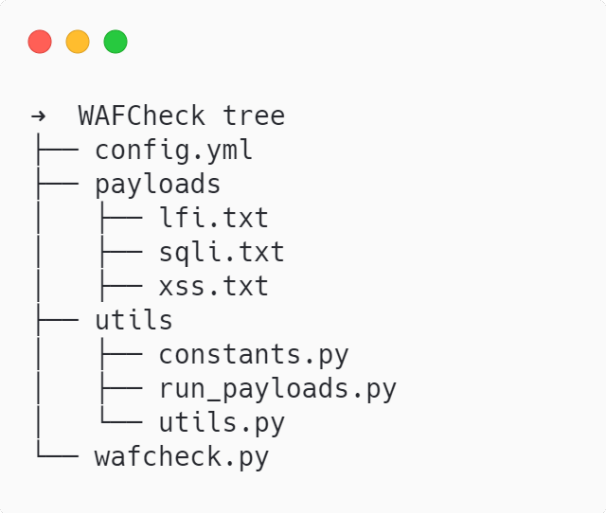
O programa automaticamente adiciona uma lista virtual, chamada de *all*, responsável por carregar e utilizar todas as demais listas durante o teste. Também é possível realizar teste usando uma única lista. Esse comportamento é definido por um parâmetro, recebido pelo programa em sua inicialização.

Na Figura 4 é possível visualizar a estrutura de diretórios e arquivos do programa. Na figura, mostra-se apenas três listas dentro do diretório *payloads*, porém trata-se apenas de uma ilustração, pois, em sua disposição padrão, o programa contém dez listas de cargas para serem utilizadas.

O ponto inicial do código é o arquivo `wafcheck.py` nele é feito o tratamento de parâmetros em cada execução do programa. Além desse arquivo, existem códigos um nível abaixo da pasta "utils", os quais têm o objetivo de armazenar constantes utilizadas em alguns fluxos, interagir de fato com a aplicação realizando requisições HTTP, carregar da base de *payloads*, as cargas que serão utilizadas no teste, e também, gerar o relatório

de saída ao final da execução.

Figura 4 – Estrutura de arquivos - WAFCheck



Fonte: Elaborado pelo autor.

Em configuração padrão, a ferramenta WAFCheck possui listas contendo *payloads* de dez tipos de vulnerabilidades diferentes, escolhidas com base na OWASP Top Ten (OWASP, 2020). Na Tabela 2 é descrito o conteúdo de cada lista em relação com o tipo da vulnerabilidade que o *payload* busca atingir. Somando todas as listas, foram agrupados, ao todo, 1.500 *payloads*.

Tabela 2 – Relação de Vulnerabilidades e listas de payloads

	Lista	**Qtd. Payloads**
Cross-site Scripting	xss.txt	200
SQL Injection	sqli.txt	200
NoSQL Injection	nosqli.txt	50
LDAP Injection	ldapi.txt	20
XML External Entity (XXE)	xxe.txt	100
Insecure Deserialization	insecuredeserialization.txt	10
Local File Inclusion	lfi.txt	900
Misconfiguration	misconfiguration.txt	7
Sensitive Information Exposure	sensitive.txt	7
Vulnerable Components	vulnerablecomponents.txt	6

Fonte: Elaborado pelo autor.

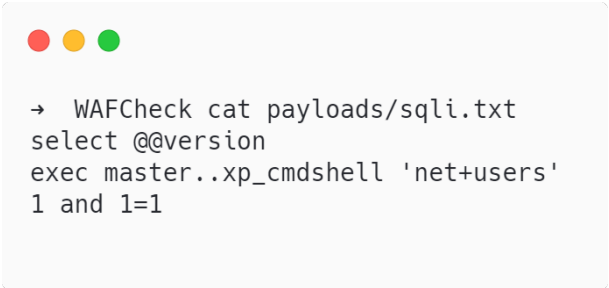
Cada lista da base de *payloads*, trata-se de um arquivo de texto, com uma carga a cada linha, da forma apresentada pela Figura 5. Não existem limites definidos para quantos *payloads* podem ser incluídos por uma lista, na figura, ilustrativamente, são mostrados

apenas três cargas úteis, porém, na versão real do arquivo, existem cerca de duzentos *payloads* nesta lista.

Todas as cargas que foram agrupados para a base de *payloads* padrão do programa foram obtidos de bases públicas, disponibilizadas na Internet, em sites como Github. As duas principais fontes foram os repositórios Payload All The Things [1] e Awesome Payloads [2].

Embora seja a falha mais recorrente, a vulnerabilidade XSS possui menos *payloads* do que a falha LFI na base de *payloads* padrão do WAFCheck. Porém este cenário ocorre apenas pela facilidade de criação de *payloads* para a vulnerabilidade de inclusão de arquivos locais, já que, por muitas vezes, apenas o nome do arquivo à ser incluído é considerado como malicioso (*e.g.*, enviar uma requisição contendo `/etc/passwd`).

Figura 5 – Lista de payloads (SQL *Injection*) - WAFCheck



Fonte: Elaborado pelo autor.

As demais configurações (iii) foram implementadas através de um arquivo YAML Ain't Markup Language (YML), chamado de *config.yml*, onde é possível definir opções como: diretório de armazenamento dos *payloads* e templates utilizados para determinar se uma requisição foi bloqueada com sucesso. O conteúdo padrão deste arquivo YML pode ser observado na Figura 6.

Por fim, o relatório final apresentado para o usuário no fim da execução de um teste, trata-se um arquivo de texto contendo todos os *payloads* que foram enviados para a aplicação, todavia não foram detectados. Esse tipo de informação pode ser útil auxiliar numa futura melhoria no conjunto de regras configuradas no WAF. A Figura 7 demonstra um exemplo de um conjunto de *payloads* que foram aceitos pelo mecanismo (*i.e.*, não foram detectados), separados pela lista pertencente e pelo método utilizado para a requisição HTTP.

A primeira versão da implementação da ferramenta está disponível como projeto de código aberto no GitHub, através do link <https://github.com/felipemelchior/WAFCheck>.

[1] https://github.com/swisskyrepo/PayloadsAllTheThings
[2] https://github.com/Muhammd/Awesome-Payloads

Figura 6 – Arquivo YML de configurações - WAFCheck

```
➜  WAFCheck cat config.yml
payloadsDir: 'payloads'
blockedRequest:
  - 'Blocked'
  - 'blocked'
  - '<h1>Requisição maliciosa bloqueada</h1>'
```

Fonte: Elaborado pelo autor.

Figura 7 – Relatório de Saída - WAFCheck

```
➜  WAFCheck cat accepted_payloads.txt
## Lista: XSS - 200 Payloads aceitos ##
Payload: <svg onload=alert(1)>
Payload: "><svg onload=alert(1)//
Payload: "onmouseover=alert(1)//
```

Fonte: Elaborado pelo autor.

4 RESULTADOS

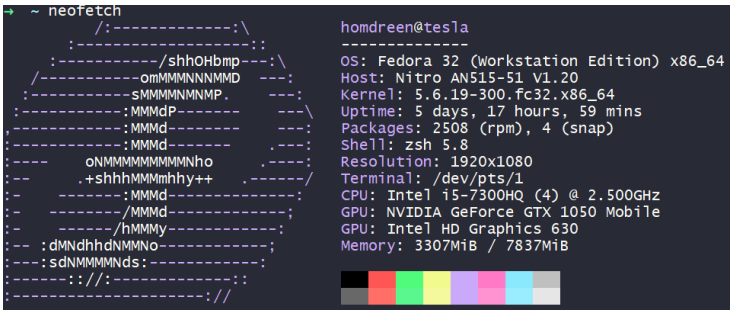
Neste capítulo são apresentadas as etapas de preparação do ambiente e também os estudos realizados à partir do uso da ferramenta implementada WAFCheck, cada análise está descrita em uma subseção. Na primeira análise, foi avaliado o aumento do tempo de latência na comunicação conforme são habilitadas novas regras. Por fim, a execução do teste de acurácia na detecção de ataques é descrita na segunda subseção, demonstrando os resultados encontrados nesta etapa.

4.1 Preparação do ambiente

Os parâmetros para escolha dos WAFs foram os seguintes: WAFs *standalone* e que sejam gratuitos, de forma análoga a trabalhos recentes (FERRAO, 2018). Entretanto, além de avaliar e comparar os WAFs ModSecurity[1], Naxsi [2] e ShadowDaemon[3] na configuração padrão, em um único cenário controlado, neste trabalho avaliamos também a solução de código aberto e mantida pela comunidade xWAF[4].

Preparação e instalação de quatro máquinas virtuais, uma para cada WAF, possuindo 2GB de RAM e com sistema operacional Ubuntu 18.04. A máquina hospedeira possui processador i5 7300-HQ quad-core de 2.5Ghz, 8GB de memória RAM, placa de vídeo GTX 1050, disco rígido Western Digital, modelo WD10SPZX de um terabyte, 5.400 rotações por minuto e cache de 128MB, controladora HM170/QM170 Chipset SATA de 6.0GHz, executando a distribuição Linux Fedora versão 32 (Workstation Edition) e a plataforma de virtualização Virtual Box na versão 6.0.6.

Figura 8 – Especificações da máquina



Fonte: Elaborado pelo autor.

As quatro máquinas virtuais, uma para cada WAF, já configuradas, estão disponíveis através do link <https://bit.ly/2RPzbU0>. Para confirmar que não houve ne-

[1] https://modsecurity.org
[2] https://github.com/nbs-system/naxsi
[3] https://shadowd.zecure.org
[4] https://github.com/Alemalakra/xWAF

nhum problema no processo de *download*, o arquivo íntegro disponibilizado possui o *hash* SHA256 `03f8f5b57c2f9e8dee23fd848fbe851d86ce1decf82963f0cd49698a55f474d6`.

Instalação da aplicação *web*. No cenário controlado, foi utilizado uma aplicação PHP que implementa as dez vulnerabilidades presentes no *Top Ten* da OWASP (OWASP, 2013). Para rodar a aplicação, foi utilizado o PHP versão 7.0.3, o MySQL versão 5.7.25 e o servidor *web* Apache (versão 2.4.18), exceto no caso do WAF Naxsi, que é compatível apenas com o servidor *web* Nginx (foi utilizada a versão 1.13.1). Em resumo, o cenário controlado é uma réplica do cenário utilizado em estudos recentes (FERRAO, 2018). Isto significa que os resultados empíricos deste trabalho poderão ser comparados (em detalhes) com outros estudos.

Instalação dos WAFs. As instalações dos WAFs seguiram a documentação de cada solução. O ModSecurity foi instalado através do próprio gerenciador de pacotes do Ubuntu, enquanto que o Naxsi e o ShadowDaemon foram instalados a partir do código fonte, disponível no site oficial dos respectivos WAFs. Por fim, no caso do xWAF, que é implementado em PHP, foi necessário adicionar apenas ao parâmetro `auto_prepend_file` no servidor Apache, fazendo com que o código fonte do xWAF fosse incluso no início de cada arquivo PHP da aplicação.

No caso do ModSecurity, foi utilizado o pacote Core Rule Set (CRS) para a configuração das regras padrões. As soluções ModSecurity e Naxsi possuem um arquivo de configuração, onde é possível determinar a pasta de localização dos arquivos de regras. O WAF ShadowDaemon necessita que as regras sejam adicionadas manualmente na aplicação Web. Já o mecanismo xWAF possui todas as regras no próprio código fonte, porém, como trata-se de um projeto *open source*, é possível modificá-lo e adicionar novas regras.

Teste de funcionamento dos WAFs. Para verificar o funcionamento de cada WAF foi criada uma regra para bloquear requisições especificas provenientes do comando cURL. Utilizando como exemplo o ModSecurity, a seguir é apresentada a implementação da regra. A regra utiliza o arquivo `curl.txt`, que contém uma lista dos "User-Agents" (um por linha) utilizado pelo comando cURL, como *"curl/7.64.1"*. O WAF bloqueia (`deny` no Algoritmo 2) e registra (`log`) todas as requisições cujo cabeçalho contém um "User-Agent" listado no arquivo "curl.txt". Ao bloquear uma requisição, o WAF registra o motivo do bloqueio (`msg`) e os detalhes da solicitação, como URL da requisição e endereço IP do solicitante.

Algorithm 2 Bloqueia requisições do cURL

```
1: SecRuleEngine On
2: SecRule REQUEST_HEADERS:User-Agent "@pmFromFile curl.txt
   id:12345,deny,log,status:403,msg:'cURL tentando enviar requests'"
```

Se o usuário enviar uma requisição ao servidor (e.g. `"curl 192.168.56.20"`), utilizando uma versão do comando cURL listada no arquivo `curl.txt`, a solicitação será automaticamente bloqueada. Para o usuário, a resposta da solicitação é apenas o código

HTTP 403 (`status`), que significa que a solicitação foi entendida pelo servidor, porém não será atendida (DOCS, 2019).

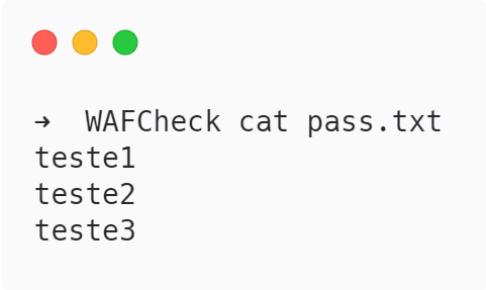
4.2 Análise de Latência

Ao instalar um *Web Application Firewall*, o administrador do sistema deve analisar as necessidades da aplicação e configurá-lo, elaborando regras específicas para ataques específicos (RAO; PRASAD; RAMESH, 2016; CLINCY; SHAHRIAR, 2018) ou aumentando o número de regras ativas. Um número maior de regras ativas pode levar a uma maior capacidade de detecção, por outro lado, não há estudos que apresentam dados empíricos detalhados sobre o impacto do número de regras no desempenho (*e.g.* latência) das requisições entre o cliente e a aplicação *web*.

Com o objetivo de identificar o impacto do número de regras, em diferentes WAFs, na latência das requisições, foi utilizado o programa WAFCheck com duas listas modificadas, simulando dois tipos de usuários, um não malicioso e outro malicioso. Enquanto que as requisições do primeiro não são bloqueadas (**Pass**), as do segundo são bloqueadas (**Match**) pelo WAF.

Aproveitando que a ferramenta aceita a inserção de novas listas de *payloads*, foram criadas duas listas para esse teste (chamadas de *pass.txt* e *match.txt*). Simulando um usuário comum, a lista *pass.txt* armazena apenas requisições que não são bloqueadas pelo WAF (*e.g.* requisição não maliciosa, enviando apenas "teste"), já a lista *match.txt* possui um *payload* de XSS, forçando o bloqueio pelo mecanismo. Em ambos os casos, cada lista possui 1.000 variações do mesmo *payload*, que foi o número de requisições escolhida para ser realizado esta análise. Nas Figuras 9 e 10 é mostrado o início do conteúdo de cada uma dessas listas.

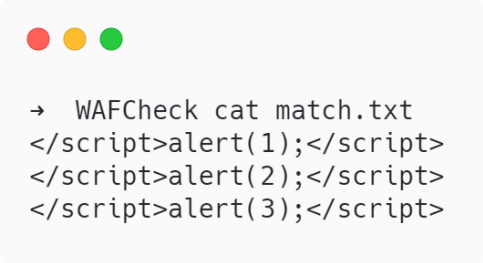
Figura 9 – Lista pass.txt - WAFCheck



Fonte: Elaborado pelo autor.

Com a informação gerada pelo WAFCheck ao final da execução, foi possível agrupar os dados obtidos em uma tabela. Na Tabela 3, a latência de uma requisição (em milissegundos) representa a média de um mil requisições.

Figura 10 – Lista match.txt - WAFCheck



Fonte: Elaborado pelo autor.

Tabela 3 – Tempo de acesso (ms) de acordo com a quantidade de regras (média de 1.000 repetições)

	ModSecurity		**Naxsi**		**ShadowD.**		**xWAF**	
	Pass	**Match**	**Pass**	**Match**	**Pass**	**Match**	**Pass**	**Match**
Padrão	2,53	2,51	1,32	1,01	2,93	2,67	1,26	1,26
+ 500	2,59	2,52	1,57	1,14	2,96	2,71	1,34	1,31
+ 1.000	2,88	2,46	1,73	1,26	3,01	2,86	1,59	1,48
+ 10.000	7,31	3,18	4,03	3,20	6,76	4,12	1,83	1,70
+ 50.000	24,93	6,17	14,29	12,98	16,21	14,08	3,17	2,97
+ 100.000	46,73	11,86	28,71	28,13	29,46	22,79	5,21	4,87

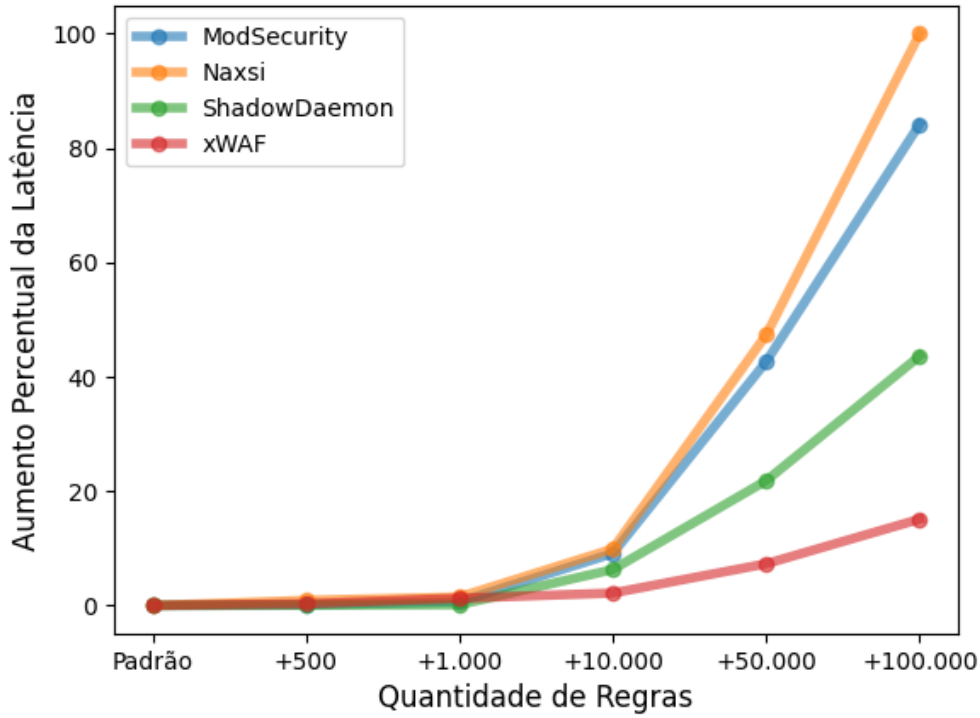
Fonte: Elaborado pelo autor.

Como pode ser observado na Tabela 3, existe uma diferença na latência entre os WAFs. Considerando que todos os WAFs foram executados na configuração padrão (linha "padrão" na tabela), a diferença pode ser explicada parcialmente pela quantidade inicial de regras em cada WAF. O conjunto de regras iniciais do Naxsi possui 60 regras, ModSecurity e ShadowDaemon contam com 170 e 26 regras, respectivamente, já o xWAF possui 49 regras por padrão. Isto, por si só, praticamente já explica a diferença de tempos iniciais entre os WAFs. Outro motivo provável são os próprios mecanismos que o WAF possui.

Por via de regra, o usuário não malicioso acaba sendo mais prejudicado com o aumento do número de regras ativas. Isto ocorre devido ao fato de uma requisição normal (**Pass**) ter de ser analisada e processada por todas as regras ativas. Já uma solicitação maliciosa (**Match**) é bloqueada na primeira regra que identificar o ataque.

Considerando as latências iniciais até 1.000, quantidade similar configurado em ambiente de produção, as soluções Naxsi e xWAF apresentaram os maiores aumentos percentuais para usuários normais (PASS), de 31% e 26%, respectivamente. Analisando até 10.000 regras adicionais, a solução Naxsi continua com maior aumento percentual de 205%, seguido do WAF ModSecurity, que aumentou percentualmente em 188%.

Figura 11 – Aumento percentual do usuário normal (PASS)



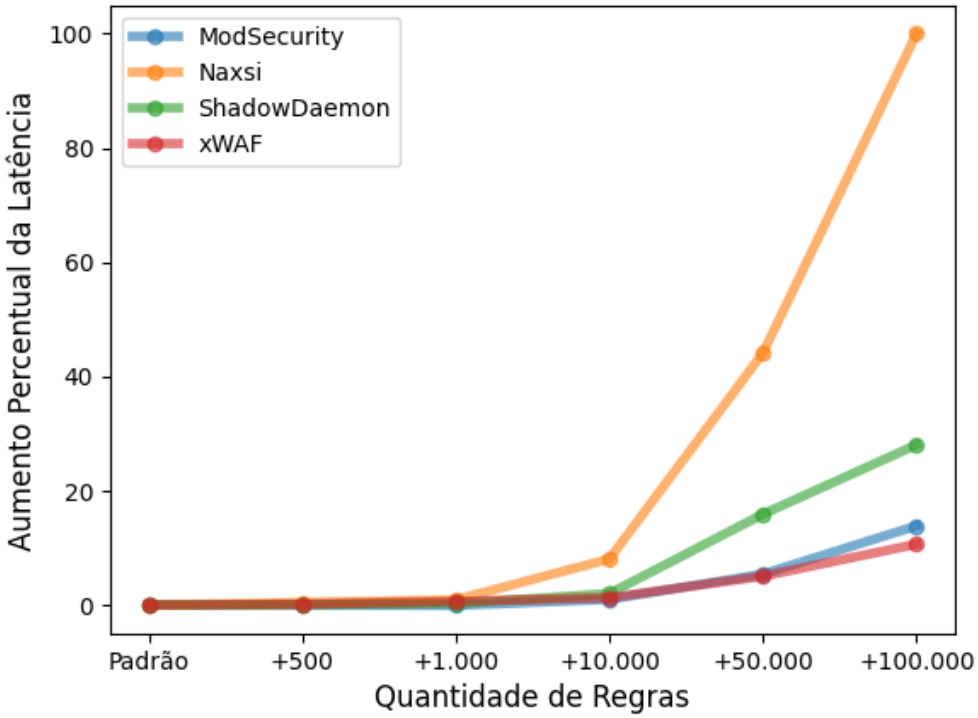
Fonte: Elaborado pelo autor.

Com o aumento progressivo no número de regras, o ModSecurity passou de 2,53ms (170 regras) para cerca de 45ms de latência (100.000 regras), o que representa um aumento percentual aproximado de 1718%. Entretanto, percentualmente, a solução Naxsi teve um desempenho pior, variando de 1.32ms (70 regras) para 28,71ms (100.000 regras), o que representa um aumento percentual de latência na ordem de 2095%, como pode ser observado na Figura 11, onde o valor 100 do eixo y representa esse valor.

Já nos testes com requisições bloqueadas pelo *Web Application Firewall*, o mecanismo Naxsi levou cerca de 28ms no cenário com cem mil regras adicionais. Tendo, assim, o maior aumento percentual dentre as outras solução, aumentando 2685% em relação ao seu primeiro tempo registrado neste teste, de apenas 1,01ms. Esse valor, quando comparado com a solução xWAF, representa um desempenho 2400% pior, como mostrado na Figura 12.

Tanto no caso do usuário não malicioso, quanto no caso do agente malicioso, os menores tempos de latência (5ms e 4ms, respectivamente) e menores percentuais de aumento (313% e 286%), de acordo com o número de regras, foram do xWAF. A principal diferença deste WAF para os demais, é que o mesmo tem seu conjunto de regras já carregados em memória, diminuindo então qualquer operação de entrada e saída que seria

Figura 12 – Aumento percentual do usuário malicioso (MATCH)



Fonte: Elaborado pelo autor.

necessária em outras soluções.

Como pode ser observado nos resultados da Tabela 3, a partir de cinquenta mil regras adicionais, a latência média das requisições fica próxima de 24ms utilizando o ModSecurity. A título de comparação, considerando uma em rede cabeada de fibra óptica, a latência de acesso Internet Control Message Protocol (ICMP) ao servidor de resolução de nomes (DNS) do Google (endereço IP 8.8.8.8) também é cerca de 24ms. O aumento desproporcional das regras configuradas, pode ocasionar que a aplicação protegida fique mais lenta, neste caso, mesmo em rede local, foi possível notar um tempo de acesso à aplicação igual à de um serviço que está exposto externamente (*i.e.* na Internet).

Para sanar dúvidas quanto à gargalos de recursos computacionais das máquinas virtuais, foi investigado também o consumo de processamento e memória RAM dos WAFs. A Tabela 4 resume os resultados para a configuração padrão, 50.000 e 100.000 regras. As colunas P e M indicam a utilização de CPU e memória RAM, respectivamente. Estes dados foram obtidos através do programa `HTOP`[5] que foi utilizado para monitorar o uso de memória RAM (em *Megabytes*) e a porcentagem de consumo de CPU.

Novamente, como esperado e de acordo com os resultados apresentados nesta seção,

[5] <https://hisham.hm/htop/>

Tabela 4 – Utilização de recursos computacionais

	ModSecurity		**Naxsi**		**ShadowD.**		**xWAF**	
	P	**M**	**P**	**M**	**P**	**M**	**P**	**M**
Padrão	50%	200	46%	240	41%	253	37%	259
50.000	80%	411	87%	280	84%	303	70%	270
100.000	94%	750	91%	330	89%	315	84%	297

os WAFs ModSecurity e Naxsi apresentaram as maiores percentagens de consumo médio de CPU, ambas acima de 90%. Este aumento significativo (de 50% para 95% no caso do ModSecurity, por exemplo) na utilização do processador pode ajudar a explicar o aumento no tempo de resposta dessas duas soluções. Além disso, como pode ser observado na Tabela 4, mais uma vez de acordo com os resultados apresentados na Tabela 3, a ferramenta xWAF foi a que resultou no menor consumo de CPU para 100.000 regras.

Com relação ao consumo de memória RAM, todas as soluções, com excessão do ModSecurity, mantiveram um consumo inferir a 350MB. Além disso, a variação no consumo de memória, entre a configuração padrão e 100.000 regras, foi pequena. O único caso destoante foi o ModSecurity, que passou de 200MB para cerca de 750MB, representando um aumento de 275% no uso de memória disponível no servidor. Para 100.000 regras, o xWAF foi o que menos consumiu memória RAM. Além disso, o xWAF foi o WAF que percentualmente menos aumentou o consumo de memória, passando de 259MB (com 200 regras) a 297 MB (com 100.000 regras), o que representa um aumento de apenas 15%.

4.3 Análise de Acurácia

Além da verificação do aumento de tempo de resposta através do aumento de regras configuradas, um teste de acurácia foi realizado, visando avaliar a eficácia de detecção dos mecanismos com a configuração padrão (*i.e.* conjunto de regras padrão). Levando como base o conjunto de problemas mais comuns em aplicações *web*, de acordo com a OWASP Top Ten (OWASP, 2020), foi agrupado um conjunto de *payloads* (*i.e* cargas úteis) e enviados para cada aplicação que o WAF estava protegendo. Estes *payloads* foram obtidos através de bases públicas, encontradas no site Github.

Ao todo, foram agrupados 1.500 *payloads* de diferentes tipos de vulnerabilidades, de acordo com o estudo dos problemas recorrentes em aplicações *web*, disponibilizado pela OWASP (OWASP, 2020). No ano de 2020, novamente a categoria de injeções segue liderando a lista de problemas em sistemas *web* e para esse tipo de vulnerabilidade, temos os *payloads* de *Cross-site scripting* (XSS), SQL/NoSQL Injection e injeções de códigos LDAP. Seguindo o *ranking*, são relatadas as categorias de exposição de dados sensíveis e de manipulação de entidades externas em arquivos Extensible Markup Language (XML),

também conhecido como XML External Entity (XXE). Em posições mais baixas, temos categorias que envolvem falhas de configurações, deserialização de objetos feito de modo inseguro e uso de componentes vulneráveis, ou seja, que existem problemas ou falhas já catalogadas naquela versão.

A ferramenta WAFCheck foi utilizada configurada para ser utilizado todos os *payloads* de sua base (*i.e* utilizando a lista virtual *all*, comentada anteriormente). Diferentemente do outro teste, aqui utilizamos apenas *payloads* maliciosos, tendo o intuito de que um bloqueio pelo WAF aconteça, podendo assim enumerar quantos destas cargas foram bloqueadas ou foram aceitas pelo mecanismo.

A Tabela 5 os dados dos testes realizados com os 1500 *payloads*, contendo a quantidade de *payloads* detectados pelos WAFs em cada categoria de falha. É possível notar que os mecanismos possuem uma facilidade maior em detectar ataques que visam injetar códigos ou comandos na aplicação, visto que tratam-se de ataques mais ativos e que possuem um certo padrão de exploração. Enquanto falhas de má configuração, exposição de dados sensíveis e uso de componentes vulneráveis são menos detectadas. Este cenário pode ser causado devido à menor quantidade de *payloads* disponíveis na internet e também ao fato de que estes problemas variam bastante em como são encontrados em uma aplicação, ou seja, não possuem uma método facilitador para a detecção destas falhas, pois dependem de critérios de arquitetura da aplicação, como linguagem de programação usada, servidor *web* utilizado, *frameworks* de desenvolvimento, entre outros fatores.

Tabela 5 – Acurácia na detecção de ataques

	ModSecurity	**Naxsi**	**ShadowD.**	**xWAF**
XSS	188/200	200/200	200/200	188/200
SQLI	103/200	156/200	117/200	77/200
NOSQLI	14/50	48/50	8/50	26/50
LDAPI	3/20	15/20	2/20	3/20
XXE	98/100	100/100	56/100	9/100
I.D	5/10	10/10	0/10	3/10
LFI	666/900	349/900	128/900	0/900
Misconfiguration	3/7	0/7	0/7	0/7
Sensitive information exposure	2/7	1/7	0/7	0/7
Vulnerable components	0/6	0/6	0/6	0/6
Total detectado	1082/1500	879/1500	511/1500	306/1500
Total em %	72.13	58.60	34.06	20.40

Fonte: Elaborado pelo autor.

O mecanismo que obteve a melhor acurácia na detecção de falhas, ainda em sua configuração padrão, foi o ModSecurity, onde conseguiu detectar com sucesso mais de 70% dos *payloads* utilizados no teste, totalizando 1082 detecções. Já o pior desempenho ficou por conta do *Web Application Firewall* xWAF, que detectou pouco mais de 300 ataques na aplicação, totalizando apenas 20% de todos os *payloads*. Vale lembrar que

este mecanismo se propõe apenas a detectar injeções de SQL e *Cross-site Scripting*, o que já explica o seu desempenho geral.

Conhecer as funcionalidades disponíveis pelo mecanismo pode afetar na eficácia na detecção de ataques, como é o caso do ModSecurity, que disponibiliza uma nova série de regras de forma configurável, são os chamados níveis de Paranoia. Ativando essa configuração, no seu nível mais alto (*i.e.* nível 4), a taxa de detecção do WAF passa de 70% para alcançar a detecção do conjunto de todos os *payloads*, ou seja, 100%. Esse resultado não garante que todo e qualquer tipo ataque ou *payload* será detectado pelo WAF, visto que o aumento dos níveis de Paranoia também implica em um aumento no índice de falsos positivos (SINGH; Samuel; Zavarsky, 2018). Neste teste, o aumento dos níveis de Paranoia fez com que o mecanismo fosse capaz de detectar e classificar como maliciosos todos os *payloads* utilizados.

É possível também, criar uma regra específica que busque de um arquivo, o *payload* exato para ser barrado pelo WAF, sendo assim, todos os mecanismos podem apresentar uma acurácia perfeita em testes com escopo fechado. Usando os dois mecanismos que obtiveram os piores desempenhos, ShadowDaemon e xWAF, podemos replicar esse comportamento adicionando regras específicas para que seja buscado o *payload* exato na requisição, fazendo com que ambos os mecanismos também detectem com sucesso os 1.500 ataques realizados nesse teste.

Em um ambiente de produção, é muito difícil atingir uma taxa de detecção de 100% pelo fato de existirem muitas variações de payloads. Há técnicas de evasão que tornam possível um ataque bem sucedido, mesmo com um WAF configurado de forma agressiva. Por exemplo, mesmo assumindo o nível máximo de Paranoia do ModSecurity, há *payloads* que não são detectados, como `file=/e'tc/pass'wd` ou `file=/???/??ss??`. Em ambos os exemplos, esses *payloads* exploram uma possível falha de LFI, retornando o conteúdo do arquivo `/etc/passwd`, sem qualquer bloqueio pelo WAF (THEMIDDLE, 2017).

5 CONSIDERAÇÕES FINAIS

Neste trabalho foi proposta e implementada a ferramenta WAFCheck, que tem como objetivo automatizar o processo de análises de soluções de WAF. Para isto, a ferramenta permite a carga de *payloads* de ataques conhecidos, e realiza a checagem fazendo requisições HTTP para a aplicação protegida pelo WAF. Como forma de demonstrar sua aplicabilidade, o WAFCheck foi utilizada para avaliar o impacto do número de regras na de latência das conexões, bem como a taxa de acurácia da detecção de ataques em diferentes soluções de WAFs. A ferramenta desenvolvida está disponível como projeto *open source* no GitHub, através do endereço <https://github.com/felipemelchior/WAFCheck>.

No primeiro teste, foi constatado o real impacto do aumento da quantidade de regras de um *Web Application Firewall* na latência do servidor de uma aplicação *web*. O WAF com o melhor desempenho em termos de tempo de resposta baixo mesmo com a adição de regras novas, foi o xWAF, mantido e criado pela comunidade. Já os piores desempenhos foram obtidos pelos mecanismos Naxsi e ModSecurity, alcançando aumentos de tempos percentuais muito altos. Outro ponto constatado foi o fato que um usuário normal, que efetua apenas requisições seguras ao servidor, é mais prejudicado com o aumento das regras, devido ao fato de suas requisições terem que passar por todo o conjunto de regras ativas.

Foi realizado, também, um estudo visando analisar a acurácia na detecção de ataques, no qual foi possível constatar que as melhores taxas de detecção foram obtidas com as soluções ModSecurity e Naxsi já os mecanismos com as piores acurácias foram Shadow Daemon e xWAF. Além do mais, foi possível obter uma taxa de detecção completa, a partir da configuração dos níveis de Paranoia do mecanismo ModSecurity, porém, garantir a detecção de todos os *payloads* existentes, em ambiente de produção real, não é possível, visto que atacantes utilizam técnicas de evasão dos mecanismos de defesa.

Como trabalhos futuros, estão mapeados: (a) a execução de análises semelhantes as deste trabalho para soluções de *Web Application Firewall* executados de forma *Security-as-a-Service* (SaaS), comparando também os resultados obtidos pelas soluções *Standalone*; (b) adicionar novas funcionalidades ao WAFCheck, como a análise de largura de banda em cenários em que o mecanismo deve responder múltiplas requisições em curto espaço de tempo.

REFERÊNCIAS

ACUNETIX. **Web Application Vulnerability Report**. 2019. Último acesso: 2019-05-17. <https://bit.ly/2wh62TH>. Citado na página 11.

CIMPANU, C. **Security bug would have allowed hackers access to Google's internal network**. 2019. Último acesso: 2019-06-15. <https://zd.net/2F8Mju8>. Disponível em: <https://www.zdnet.com/article/security-bug-would-have-allowed-hackers-access-to-googles-internal-network/>. Citado na página 11.

CLINCY, V.; SHAHRIAR, H. Web application firewall: Network security models and configuration. In: **2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)**. [S.l.: s.n.], 2018. v. 01, p. 835–836. ISSN 0730-3157. Citado 4 vezes nas páginas 12, 15, 16 e 29.

DOCS, M. W. **Códigos de status de respostas HTTP**. 2019. Último acesso: 2019-06-11. <https://mzl.la/2Q7HbcN>. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>. Citado na página 29.

EC-COUNTIL. **WHY PYTHON IS IMPORTANT FOR SECURITY PROFESSIONALS**. 2021. <https://bit.ly/3eafWM7>. Disponível em: <https://blog.eccouncil.org/why-python-is-important-for-security-professionals/>. Citado na página 21.

FERRAO, I. G. **Análise black-box de ferramentas de segurança na Web**. 65 p. Monografia (Dissertação) — Universidade Federal do Pampa, campus Alegrete, Rio Grande Do Sul, 2018. Citado 6 vezes nas páginas 11, 15, 16, 17, 27 e 28.

FERRAO, I. G.; MACEDO, D. D. J. de; KREUTZ, D. Investigação o do impacto de frameworks de desenvolvimentode software na segurança de sistemas 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. Último acesso: 2019-06-17. <https://bit.ly/2WQAfZd>. Citado 2 vezes nas páginas 11 e 17.

FUNK, R.; EPP, N.; A., C. C. 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. ISSN 2446-7634. Disponível em: <https://revistas.setrem.com.br/index.php/reabtic/article/view/266>. Citado na página 16.

HACKERONE. **The HackerOne Top 10 Most Impactful and Rewarded Vulnerability Types**. 2019. Último acesso: 2019-06-15. <https://bit.ly/2RiWbGu>. Disponível em: <https://www.hackerone.com/resources/top-10-vulnerabilities>. Citado na página 11.

MOOSA, A.; ALSAFFAR, E. M. Proposing a hybrid-intelligent framework to secure e-government web applications. In: **Proceedings of the 2Nd International Conference on Theory and Practice of Electronic Governance**. New York, NY, USA: ACM, 2008. (ICEGOV '08), p. 52–59. ISBN 978-1-60558-386-0. Disponível em: <http://doi.acm.org/10.1145/1509096.1509109>. Citado 3 vezes nas páginas 12, 16 e 17.

OWASP. **Top 10 2013**. 2013. <https://bit.ly/1a3Ytvg>. Disponível em: <https://www.owasp.org/index.php/Top_10_2013-Top_10>. Citado 2 vezes nas páginas 16 e 28.

OWASP. **Top 10 2020**. 2020. <https://bit.ly/395k2Uh>. Disponível em: <https://owasp.org/www-project-top-ten/>. Citado 5 vezes nas páginas 12, 19, 22, 23 e 33.

PRANDL, S.; LAZARESCU, M.; PHAM, D.-S. A study of web application firewall solutions. In: JAJODA, S.; MAZUMDAR, C. (Ed.). **Information Systems Security**. Cham: Springer International Publishing, 2015. p. 501–510. ISBN 978-3-319-26961-0. Citado 2 vezes nas páginas 16 e 17.

RAO, G. R. K.; PRASAD, R. S.; RAMESH, M. Neutralizing cross-site scripting attacks using open source technologies. In: **Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies**. New York, NY, USA: ACM, 2016. (ICTCS '16), p. 24:1–24:6. ISBN 978-1-4503-3962-9. Disponível em: <http://doi.acm.org/10.1145/2905055.2905230>. Citado 4 vezes nas páginas 12, 15, 16 e 29.

RAZZAQ, A. et al. Critical analysis on web application firewall solutions. In: **2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS)**. [S.l.: s.n.], 2013. p. 1–6. Citado 3 vezes nas páginas 12, 15 e 16.

RIETZ, R. et al. Firewalls for the web 2.0. In: **2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. [S.l.: s.n.], 2016. p. 242–253. Citado na página 16.

SINGH, J. J.; Samuel, H.; Zavarsky, P. Impact of paranoia levels on the effectiveness of the modsecurity web application firewall. In: **2018 1st International Conference on Data Intelligence and Security (ICDIS)**. [S.l.: s.n.], 2018. p. 141–144. Citado 3 vezes nas páginas 16, 17 e 35.

SROKOSZ, M.; RUSINEK, D.; KSIEZOPOLSKI, B. A new waf-based architecture for protecting web applications against csrf attacks in malicious environment. In: **2018 Federated Conference on Computer Science and Information Systems (FedCSIS)**. [S.l.: s.n.], 2018. p. 391–395. Citado 3 vezes nas páginas 12, 16 e 17.

THEMIDDLE. **Web Application Firewall (WAF) Evasion Techniques**. 2017. <https://bit.ly/3n6geHZ>. Disponível em: <https://medium.com/secjuice/waf-evasion-techniques-718026d693d8>. Citado na página 35.

ÍNDICE