WAFCheck: uma ferramenta para auxiliar nos testes de *Web Application Firewalls* (WAFs)

Felipe Homrich Melchior[1], **Maurício Fiorenza**[1], **Diego Kreutz**[1]

[1]Universidade Federal do Pampa (Unipampa)

fehmel@gmail.com,{mauriciofiorenza,diegokreutz}@unipampa.edu.br

Resumo. *Estatísticas indicam que até 90% das aplicações disponibilizadas na Internet possuem algum tipo de vulnerabilidade de software. Estudos recentes apontam também que firewalls de aplicação, mais conhecidos como WAFs, podem contribuir para mitigar a exploração de mais de 70% das vulnerabilidades de sistemas web. Entretanto, determinar a capacidade e a qualidade de um WAF pode ser um desafio. Nós propomos a ferramenta WAFCheck para auxiliar nos testes de latência, carga e acurácia de detecção de WAFs. A WAFCheck permite a inclusão e avaliação de conjuntos diversos de payloads, que são utilizados na exploração de vulnerabilidades de sistemas web. Utilizando centenas de payloads das dez vulnerabilidades mais recorrentes segundo a OWASP, nós avaliamos o desempenho dos WAFs gratuitos ModSecurity, Naxsi, ShadowD e xWAF, demostrando o funcionamento e a aplicabilidade da ferramenta.*

1. Introdução

Estudos apontam que 50% e 90% das aplicações *web* disponíveis na Internet possuem pelo menos uma vulnerabilidade de nível de risco alto e médio, respectivamente [ACUNETIX, 2019]. Segundo relatórios especializados, a vulnerabilidade de 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* das sessões, podendo assim, acessar recursos e dados privados dos usuários e sistemas [CIMPANU, 2019].

Uma das formas de conter muitos dos ataques que exploram vulnerabilidades antigas e recorrentes, como XSS e SQL *Injection*, é através da utilização de ferramentas de segurança como firewalls de aplicação *web* (WAFs). Um WAF é um serviço de segurança implementado entre o cliente (*e.g.*, navegador) e a aplicação (*e.g.*, sistema executando em um servidor *web*). 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 o ModSecurity[1] é capaz de mitigar a exploração de vulnerabilidades recorrentes em mais de 70% [FERRAO et al., 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 de alguns cenários com o *framework* Symfony[2], utilizado para o desen-

[1]https://modsecurity.org

[2]https://symfony.com

volvimento de aplicações *web* PHP, e os WAFs Naxsi[3] e ShadowDaemon[4]. Enquanto o Symfony, através dos seus próprios mecanismos de segurança, mitiga até 60% de vulnerabilidades comuns em aplicações *web*, a adição dos WAFs Naxsi e ShadowDaemon reduz a mitigação para 50%. Foi constatado que estes dois WAFs interferem nos mecanismos de segurança implementados pelo *framework* de desenvolvimento Symfony [FERRAO, 2018, MELCHIOR et al., 2020b].

Estudos no contexto de WAFs tem abordado diferentes desafios técnicos e oportunidades de pesquisa, como algoritmos para detectar ataques que objetivam explorar vulnerabilidades específicas (*e.g.*, CSRF e XSS) [SROKOSZ et al., 2018], mecanismos para aumentar o desempenho de processamento de requisições em cenários com grandes volumes de requisições (*e.g.*, milhares de requisições por segundo) [MOOSA and ALSAFFAR, 2008] e revisão minuciosa do estado da arte, procurando comparar e compreender o funcionamento dos diferentes WAFs existentes [CLINCY and SHAHRIAR, 2018, RAZZAQ et al., 2013, MELCHIOR et al., 2020a]. Na prática, há uma carência de ferramentas e trabalhos que investiguem WAFs de forma empírica, identificando aspectos importantes como o impacto do número de regras na latência das requisições aos sistemas Web e o nível de acurácia de detecção de ataques.

Neste trabalho propomos uma ferramenta, denominada WAFCheck, cujo objetivo é auxiliar o processo de avaliação empírica de WAFs no que diz respeito ao impacto da latência e acurácia na detecção de vulnerabilidades em sistemas *web*. A ferramenta nasceu a partir de necessidades identificadas em pesquisas prévias relacionadas à avaliação de aspectos de desempenho e qualidade de WAFs [MELCHIOR et al., 2020a, MELCHIOR et al., 2019, FERRAO, 2018]. A WAFCheck disponibiliza uma base de *payloads* para executar requisições (sequenciais ou paralelas) aos sistemas *web* através dos WAFs. A base atual de *payloads* da ferramenta contém a catalogação de um conjunto significativo de padrões de exploração para cada uma das dez vulnerabilidades mais recorrentes em aplicações *web* segundo a lista da OWASP de 2020 [OWASP, 2020].

Utilizamos a WAFCheck para avaliar o impacto da quantidade de regras ativas na latência das comunicações entre clientes e aplicações *web*. Durante os experimentos, avaliamos a latência gerada pelos WAFs ModSecurity, Naxsi, ShadowD e xWAF com um número variável de regras (da configuração padrão até 100.000), replicando um estudo anterior, no qual os testes de latência haviam sido realizados de forma mais rudimentar, isto é, sem utilizar uma ferramenta como a WAFCheck [MELCHIOR et al., 2019]. Os resultados indicam que, quando o número de regras ativas no WAF ultrapassa a marca de 1.000, há um aumento significativo na latência de comunicação (*e.g.*, de 2,53ms para 46,73ms a latência média de cada requisição HTTP utilizando o método GET) para todos os WAFs analisados. Adicionalmente, avaliamos também a acurácia dos mecanismos de detecção dos WAFs para os *payloads* catalogados das dez vulnerabilidades mais recorrentes em sistemas *web*.

As principais contribuições técnicas do trabalho são: o projeto e a implementação da ferramenta WAFCheck, que auxilia os usuários na avaliação da latência e da acurácia de WAFs; avaliação do impacto da quantidade de regras na latência das requisições entre os clientes e a aplicação *web*; e avaliação da acurácia de detecção de *payloads* maliciosos

[3]https://github.com/nbs-system/naxsi

[4]https://shadowd.zecure.org

nos WAFs gratuitos ModSecurity, Naxsi, ShadowD e xWAF.

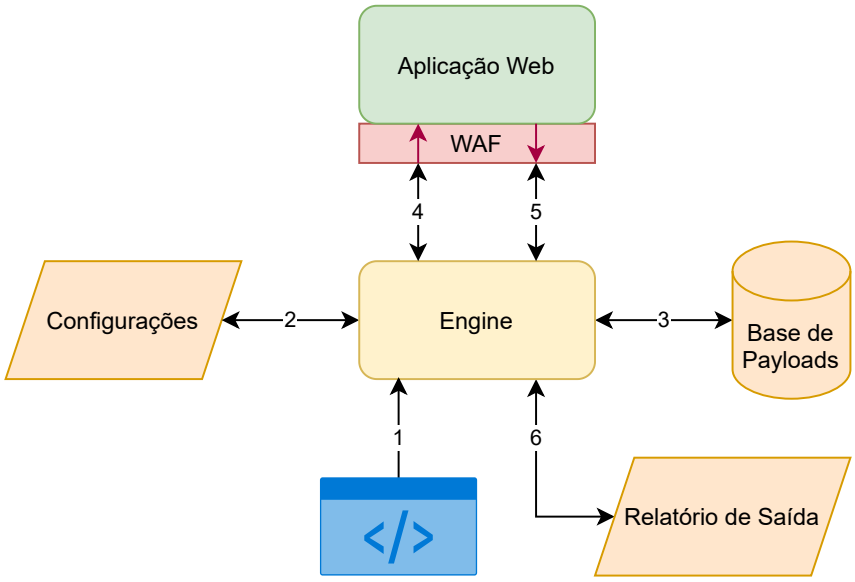
Na Seção 2 apresentamos a arquitetura e detalhes de funcionamento e implementação da ferramenta WAFCheck. As avaliações de latência das requisições e acurácia de detecção dos WAFs, apresentamos na Seção 3. Finalmente, nas Seções 4 e 5 descrevemos a demonstração e apresentamos as considerações finais, respectivamente.

2. Arquitetura e Implementação

A Figura 1 ilustra a arquitetura e o fluxo de operação da WAFCheck. A ferramenta possui três módulos principais: *engine*, base de *payloads* e configurações. O módulo *engine* é o responsável por gerar cargas de requisições HTTP para a aplicação *web*. Além disso, esse módulo produz também o relatório de execução, que é apresentado para o usuário da ferramenta.

A base de *payloads* contém listas de padrões de exploração de vulnerabilidades, catalogados por tipo de vulnerabilidade, que são utilizados nos testes de latência das requisições e na avaliação da acurácia dos mecanismos de detecção dos WAFs. As listas de *payloads* podem ser criadas a partir de categorias conhecidas de vulnerabilidades, como as dez mais recorrentes em aplicações *web* no ano de 2020 segundo a OWASP [OWASP, 2020].

Figura 1. Arquitetura e fluxo de operação da ferramenta



No módulo de configurações são definidas as opções de execução que viabilizam algumas das funcionalidades da WAFCheck. Por exemplo, nas configurações pode ser incluída a definição de *templates* que permitem a identificação das mensagens de bloqueio personalizadas dos WAFs. Em alguns casos, embora o código de retorno da requisição HTTP seja 200 (indicando sucesso), o WAF pode ter bloqueado a requisição. Um exemplo prático é o Naxsi, que permite a configuração de uma página personalizada de bloqueio (*e.g.*, *status* HTTP 200). Quando a requisição for bloqueada, a página personalizada será apresentada para o navegador (usuário final). Através de um *template*, podemos identificar a mensagem personalizada do Naxsi para a WAFCheck.

Na Figura 1 é ilustrado, através das setas numeradas, o fluxo de operação da ferramenta. O fluxo é representado pelos seguintes passos: (1) informação dos parâmetros de execução da ferramenta, incluindo parâmetros mínimos (obrigatórios) como endereço da aplicação e a lista de *payloads* a ser utilizada nos testes; (2) leitura das configurações pré-definidas da ferramenta, como o caminho do diretório da base de *payloads* e os *templates* que ajudam a identificar as requisições bloqueadas em WAFs que apresentam mensagens personalizadas; (3) carga das listas de *payloads* que serão utilizados nos testes; (4) envio de uma requisição HTTP para cada *payload* carregado; (5) recebimento e processamento (*e.g.*, identificar se a requisição foi bloqueada) da resposta da aplicação e/ou WAF; (6) geração do relatório final, identificando os *payloads* que não foram detectados pelo WAF.

Para a implementação, foi utilizada a linguagem Python e módulos como o Requests, PyYAML e Colorama. O código fonte está disponível online em `https://github.com/felipemelchior/WAFCheck`. Para a base de *payloads*, foram utilizados arquivos de texto que agrupam séries de cargas úteis, de diferente tipos de ataques. A versão atual da ferramenta disponibiliza 1.500 *payloads*, agrupados em dez listas, sendo 200 XSS, 200 SQLi, 50 NoSQL *Injection*, 20 LDAP *Injection*, 100 XML *External Entity* (XXE), 10 *Insecure Deserialization*, 900 *Local File Inclusion* (LFI), 7 *Misconfiguration*, 7 *Sensitive Information Exposure* e 6 *Vulnerable Components*. Essas cargas úteis foram obtidas de bases públicas, incluindo os repositórios Payload All The Things[5] e Awesome Payloads[6].

Para adicionar uma nova lista de *payloads*, basta o usuário criar um novo arquivo contendo uma carga útil por linha. Ao colocar o arquivo no diretório `payloads`, a nova lista estará automaticamente disponível para utilização na ferramenta. Por exemplo, o arquivo `lfi.txt`, representando as cargas úteis para exploração da vulnerabilidade LFI (*Local File Inclusion*), uma vez adicionado ao diretório, estará disponível para utilização através do parâmetro `-l lfi` da WAFCheck, similar ao que ocorre com bibliotecas em ferramentas como o compilador `gcc`.

Tomando como exemplo a execução da ferramenta ilustrada a seguir, temos: (a) as listas de *payloads* XSS e LFI, (b) o site alvo `192.168.1.1` e (c) o relatório da execução no arquivo `resultado.log`. Ao final da execução, o usuário poderá verificar todos os *payloads* que não foram detectados pelo WAF no arquivo de saída. Em posse dessa informação, o usuário poderá, posteriormente, ajustar as configurações dos WAFs (*e.g.*, especificar novos padrões para detecção de *payloads* não bloqueados).

```
./wafcheck.py -l xss,lfi -u 192.168.1.1 -o resultado.log
```

3. Avaliação

Para os testes de latência e acurácia, foram configuradas quatro VMs Ubuntu Server 18.04 idênticas, uma para cada WAF testado, e executadas na plataforma de virtualização Virtual Box na versão 6.0.6. A ferramenta WAFCheck foi executada na máquina hospedeira Linux Fedora versão 32 (Workstation Edition), com processador i5 7300-HQ quad-core de 2,5GHz, 8GB de memória RAM e disco rígido Western Digital, modelo WD10SPZX de um terabyte, 5.400 rotações por minuto, cache de 128MB, e controladora HM170/QM170 Chipset SATA de 6,0GHz.

[5]https://github.com/swisskyrepo/PayloadsAllTheThings
[6]https://github.com/Muhammd/Awesome-Payloads

As quatro máquinas virtuais, contendo a aplicação *web* e os WAFs instalados e configurados, estão disponíveis através do link `https://bit.ly/2RPzbU0`. O arquivo compactado, contendo as VMs, possui o *hash* criptográfico SHA256 *03f8f5b57c2f9e8dee23fd848fbe851d86ce1decf82963f0cd49698a55f474d6*, que pode ser utilizado para verificar a integridade após o *download*.

Nas VMs, utilizamos o mesmo cenário controlado proposto em [FERRAO et al., 2018]. O cenário consiste de uma aplicação PHP que implementa as dez vulnerabilidades mais recorrentes de aplicações *web*, implementada com 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).

3.1. Latência das Requisições

Ao instalar um WAF, o administrador do sistema deve analisar as necessidades da aplicação e configurá-lo apropriadamente, elaborando regras específicas para ataques específicos ou aumentando o número de regras ativas [RAO et al., 2016, CLINCY and SHAHRIAR, 2018]. Um número maior de regras ativas pode levar a uma maior capacidade de detecção e, ao mesmo tempo, impactar negativamente a latência das requisições entre o cliente e a aplicação *web*.

Com o objetivo de identificar o impacto do número de regras na latência das requisições, utilizamos a ferramenta WAFCheck com duas listas modificadas (denominadas de *pass.txt* e *match.txt*), simulando dois tipos de usuários, um não malicioso (*Pass*) e outro malicioso (*Match*). Enquanto que a lista do usuário não malicioso contém apenas cargas úteis que não exploram nenhuma vulnerabilidade, a lista do usuário malicioso contém *payloads* XSS comuns, que são detectados e bloqueados pelos WAFs.

Como pode ser observado na Tabela 1, cujos dados representam a média de um mil requisições (em ms), a latência aumenta com o número de regras. Na configuração padrão, há uma diferença de latência entre os WAFs, que pode ser explicada pelo número de regras iniciais de cada WAF, com exceção do ShadowDaemon. Apesar deste WAF possuir o menor número de regras (apenas 26), ele apresenta a maior latência. Os WAFs ModSecurity, Naxsi e xWAF iniciam com 170, 60 e 49 regras, respectivamente.

Tabela 1. Tempo resposta (em ms) da aplicação ao cliente

	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

Para todos os WAFs, a latência das requisições contendo *payloads* detectados (*Match*) é igual ou menor que a latência para as requisições dos usuários não maliciosos. Essa diferença aumenta com a quantidade de regras ativas no sistema. Isto ocorre devido

ao fato de uma requisição normal (*Pass*) precisa ser analisada e processada por todas as regras ativas. Diferentemente, uma solicitação maliciosa (*Match*) é bloqueada na primeira regra que identificar o ataque. Em alguns casos, a diferença entre *Pass* e *Match* pode ser significativa. Por exemplo, no caso do ModSecurity, com 10.000 regras ativas, a latência percebida pelo usuário não malicioso pode ser 200% maior que a latência percebida pelo usuário malicioso.

Considerando 1.000 regras adicionais, as soluções Naxsi e xWAF apresentaram os maiores aumentos percentuais (31% e 26%, respectivamente) para usuários normais (*Pass*). Se consideramos 10.000 regras adicionais, a solução Naxsi continua com o maior aumento percentual (205%), seguida pelo ModSecurity, que aumentou percentualmente em 188%. Como pode ser observado, o número de regras possui um impacto significativo na latência das requisições às aplicações *web* protegidas pelos WAFs.

3.2. Acurácia de Detecção

A Tabela 2 apresenta os dados dos testes realizados com os 1500 *payloads*, contendo a quantidade de cargas úteis detectadas pelos WAFs em cada categoria de vulnerabilidade. Podemos observar que os mecanismos dos WAFs são mais eficientes em detectar ataques que visam injetar códigos ou comandos na aplicação, visto que tratam-se de ataques mais ativos e que possuem um padrão de exploração bem conhecido. Enquanto isto, vulnerabilidades relacionadas a má configuração, exposição de dados sensíveis e uso de componentes vulneráveis são menos detectadas. Este cenário é, em partes, resultado da menor disponibilidade de padrões de configuração dos WAFs para esses tipos de ataques. Além disso, a forma de explorar esses tipos mais específicos de vulnerabilidades varia de forma mais significativa quando comparada com padrões de exploração de vulnerabilidades mais tradicionais (*e.g.*, SQL *Injection*, XSS, LFI), o que dificulta de sobremaneira a elaboração de regras e o desenvolvimento de mecanismos de detecção eficientes.

Tabela 2. 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

O mecanismo que obteve a melhor acurácia na detecção de falhas, em sua configuração padrão, foi o ModSecurity, que conseguiu detectar com sucesso mais de 70% dos *payloads* utilizados nos testes. Já o xWAF apresentou o pior desempenho, isto

é, detectou pouco mais de 300 ataques na aplicação, totalizando apenas 20% de todos os *payloads*. Vale ressaltar que o xWAF foi projetado primariamente para detectar injeções de SQL e XSS, o que explica o seu desempenho geral.

Os parâmetros de otimização e configuração detalhada de regras do WAF podem afetar a eficácia na detecção de ataques, como é o caso do ModSecurity. Para os 1.500 *payloads*, ao ativarmos as configurações avançadas, como o nível 4 de paranóia e a inclusão de cargas úteis específicas, a taxa de detecção aumenta para 100%. Entretanto, vale ressaltar que esse resultado não garante que todo e qualquer tipo de ataque ou *payload* será detectado pelo ModSecurity.

No caso do ModSecurity, o aumento dos níveis de paranóia também implica em um aumento no índice de falsos positivos [SINGH et al., 2018]. Além disso, em um ambiente de produção, é praticamente impossível atingirmos uma taxa de detecção constante 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 paranóia 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`.

4. Demonstração

A demonstração da ferramenta será realizada através de um ambiente virtual composto por quatro servidores de aplicação, cada um configurado com um WAF distinto (ModSecurity, Naxsi, ShadowD e xWAF) interceptando as requisições. O ambiente estará hospedado em dispositivo próprio dos autores. O funcionamento da ferramenta será demonstrado através dos seguintes passos: (a) apresentação dos parâmetros de configuração e execução da ferramenta; (b) apresentação da aplicação *web* vulnerável; (c) apresentação das configurações básicas dos quatro WAFs; e (d) demonstração da ferramenta para testes de latência de requisições e acurácia de detecção dos WAFs.

5. Considerações Finais

Propomos e implementamos a ferramenta WAFCheck, que tem como objetivo automatizar e auxiliar o processo de avaliação de latência e acurácia de detecção de WAFs. A ferramenta permite a carga de *payloads* de ataques conhecidos e o teste sistemático de cada carga útil através de requisições HTTP para a aplicação protegida pelo WAF. Como forma de demonstrar a sua aplicabilidade, a WAFCheck foi utilizada para avaliar o impacto do número de regras na latência das requisições e a acurácia de detecção de quatro WAFs. A ferramenta desenvolvida está disponível como projeto de código aberto no GitHub (`https://github.com/felipemelchior/WAFCheck`). Acreditamos que a ferramenta, apesar de estar em um estágio inicial, tem o potencial de contribuir com as avaliações empíricas de latência, acurácia de detecção e vazão de WAFs. Além disso, ela pode servir de instrumento didático em disciplinas aplicadas de segurança computacional. Sugestões de melhoria e contribuições de desenvolvimento são bem vindas.

Referências

ACUNETIX (2019). Web application vulnerability report. `https://bit.ly/36LaeNG`.

CIMPANU, C. (2019). Security bug would have allowed hackers access to google's internal network. https://zd.net/2F8Mju8.

CLINCY, V. and SHAHRIAR, H. (2018). Web application firewall: Network security models and configuration. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 835–836.

FERRAO, I. G. (2018). Análise black-box de ferramentas de segurança na Web. Trabalho de conclusão de curso, Universidade Federal do Pampa. Orientador: Diego Kreutz.

FERRAO, I. G., de MACEDO, D. D. J., and KREUTZ, D. (2018). Investigação o do impacto de frameworks de desenvolvimentode software na segurança de sistemas web. In *3o Workshop Regional de Segurança da Informação e de Sistemas Computacionais*.

HACKERONE (2019). The hackerone top 10 most impactful and rewarded vulnerability types. https://www.hackerone.com/resources/top-10-vulnerabilities.

MELCHIOR, F., KREUTZ, D., and FIORENZA, M. (2019). Web Application Firewalls (WAFs): o impacto do número de regras na latência das requisições Web. In *4o Workshop Regional de Segurança da Informação e de Sistemas Computacionais*.

MELCHIOR, F., KREUTZ, D., and FIORENZA, M. (2020a). Web Application Firewalls (WAFs): o impacto do número de regras na latência das requisições Web. *Revista Eletrônica Argentina-Brasil de Tecnologias da Informação e da Comunicação*, 3(1). Edicão especial da versão extendida dos melhores trabalhos do WRSeg 2019.

MELCHIOR, F. H., KREUTZ, D., FIORENZA, M., FLORA, F., FERRAO, I., FERNANDES, R., ESCARRONE, T., and MACEDO, D. (2020b). Introdução à Web Application Firewalls (WAFs): Teoria e prática. In *Minicursos da XVII Escola Regional de Redes de Computadores*. SBC. https://doi.org/10.5753/sbc.5929.0.5.

MOOSA, A. and ALSAFFAR, E. M. (2008). 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*, pages 52–59. ACM.

OWASP (2020). Top 10 2020. https://bit.ly/395k2Uh.

RAO, G. R. K., PRASAD, R. S., and RAMESH, M. (2016). Neutralizing cross-site scripting attacks using open source technologies. In *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, pages 24:1–24:6. ACM.

RAZZAQ, A., HUR, A., SHAHBAZ, S., MASOOD, M., and AHMAD, H. F. (2013). Critical analysis on web application firewall solutions. In *2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS)*, pages 1–6.

SINGH, J. J., Samuel, H., and Zavarsky, P. (2018). Impact of paranoia levels on the effectiveness of the modsecurity web application firewall. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 141–144.

SROKOSZ, M., RUSINEK, D., and KSIEZOPOLSKI, B. (2018). 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)*, pages 391–395.