UNIVERSIDADE FEDERAL DE SANTA MARIA
COLÉGIO TÉCNICO INDUSTRIAL DE SANTA MARIA
CURSO SUPERIOR DE TECNOLOGIA EM REDES DE COMPUTADORES

UTILIZANDO FIREWALL DE APLICAÇÃO NO PROCESSO DE DESENVOLVIMENTO DE SISTEMAS WEB

TRABALHO DE CONCLUSÃO DE CURSO

Alfredo Del Fabro Neto

Santa Maria, RS, Brasil

2013

CTISM/UFSM, RS	DEL FABRO NETO, Alfredo	Graduado	2013

UTILIZANDO FIREWALL DE APLICAÇÃO NO PROCESSO DE DESENVOLVIMENTO DE SISTEMAS WEB

Alfredo Del Fabro Neto

Trabalho apresentado ao Curso de Graduação em Tecnologia em
Redes de Computadores, Área de concentração em Segurança da Informação, da
Universidade Federal de Santa Maria (UFSM, RS),
como requisito parcial para obtenção do grau de
Tecnólogo em Redes de Computadores.

Orientador: Prof. Ms. Rogério Correa Turchetti

Santa Maria, RS, Brasil

2013

Universidade Federal de Santa Maria
Colégio Técnico Industrial de Santa Maria
Curso Superior de Tecnologia em Redes de Computadores

A Comissão Examinadora, abaixo assinada,
aprova a Monografia

UTILIZANDO FIREWALL DE APLICAÇÃO NO PROCESSO DE DESENVOLVIMENTO DE SISTEMAS WEB

elaborada por
Alfredo Del Fabro Neto

como requisito parcial para obtenção do grau de
Tecnólogo em Redes de Computadores

COMISSÃO EXAMINADORA

Rogério Correa Turchetti, Me.
(Presidente/Orientador)

Celio Trois, Me. (UFSM)

Walter Priesnitz Filho, Me. (UFSM)

Santa Maria, 25 de janeiro de 2013

RESUMO

Monografia
Curso Superior de Tecnologia em Redes de Computadores
Universidade Federal de Santa Maria

UTILIZANDO FIREWALL DE APLICAÇÃO NO PROCESSO DE DESENVOLVIMENTO DE SISTEMAS WEB
AUTOR: ALFREDO DEL FABRO NETO
ORIENTADOR: ROGÉRIO CORREA TURCHETTI
Data e Local da Defesa: Santa Maria, 25 de janeiro de 2013

Os sistemas *Web* predominam na maioria dos segmentos da sociedade. Ao mesmo tempo que cresce o número de aplicações e recursos *online*, aumentam também as preocupações relacionadas à segurança da informação. Este trabalho apresenta um *firewall* de aplicação, denominado UniscanWAF, que auxilia no controle de segurança para aplicações Web através da implementação de uma *whitelist* e *blacklist*. Sua principal finalidade é detectar, interceptar e bloquear tentativas de ataques do tipo *XSS, RFI, LFI, RCE e SQL Injection* às aplicações *Web* através de uma *blacklist* e/ou somente permitir acesso a recursos expressamente cadastrados em uma *whitelist*. Para isso, uma abordagem diferente foi proposta para ser adicionada ao SDLC, de forma que "obrigue" a equipe de codificação a cadastrar os possíveis valores aceitos pela aplicação para cada vulnerabilidade desenvolvida. Qualquer valor informado que não sejam os permitidos é bloqueado. Testes foram efetuados de forma a provar as propriedades de segurança e do funcionamento da *whitelist*. A ferramenta foi comparada a uma solução amplamente utilizada, o *ModSecurity.* Constatou-se que o UniscanWAF conseguiu detectar mais ataques, sendo portanto validada a funcionalidade de *blacklist*. Por último foram realizados testes de desempenho, e observou-se que o UniscanWAF pode ser utilizado em ambientes com grande número de requisições, mas que ainda precisa passar por melhorias no processo de análise das requisições.

Palavras-chave: Web Application Firewall. Security Software Development Life-Cycle. UniscanWAF.

ABSTRACT

Monography
Superior Course of Technology in Computer Networks
Federal University of Santa Maria

Using Application Firewall in the development process of Web systems

AUTHOR: ALFREDO DEL FABRO NETO
ADVISER: ROGÉRIO CORREA TURCHETTI

Defense Place and Date: Santa Maria, January 25th, 2013

The Web Systems predominate in most of segments of society. While the number of applications and online resources are growing, the concerns related to information security are also increased. This paper presents an application firewall called UniscanWAF, which helps to control security for Web applications by implementing a whitelist and blacklist. Its main purpose is to detect, intercept and block attacks like XSS, RFI, LFI, RCE and SQL Injection to Web applications using the blacklist, and only allow access to resources explicitly registered on a whitelist. The UniscanWAF tool was compared with a solution widely used called ModSecurity, it was observed that the UniscanWAF was able to detect more attacks, and thus validated the blacklist functionality. Finally performance tests were conducted, and it was observed that the UniscanWAF can be used in environments with large number of requests, but compared to ModSecurity has underperformed.

Key words: Web Application Firewall. Security Software Development Life-Cycle. UniscanWAF.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CPU	*Central Processing Unit*
HTTP	*HyperText Transfer Protocol*
IDS	*Intrusion Detection System*
IP	*Internet Protocol*
IPS	*Intrusion Prevention System*
LFI	*Local File Include*
OWASP	*Open Web Application Security Project*
PHP	*Personal Hypertext Preprocessor*
RAM	*Random Access Memory*
RCE	*Remote Command Execution*
RFI	*Remote File Include*
SDLC	*Software Development Life-Cycle*
SQL	*Structured Query Language*
URL	*Uniform Resource Locator*
WAF	*Web Application Firewall*
XSS	*Cross-site Scripting*

SUMÁRIO

1 INTRODUÇÃO

Os sistemas *Web* predominam na maioria dos segmentos da sociedade. Atualmente, há uma grande dependência por parte de pessoas e organizações à aplicações *Web*. A maioria dos serviços, outrora disponíveis somente com presença física do envolvido, como serviços bancários e de compras e serviços do governo, estão disponíveis via *Web*. Além disso, há novos serviços, como redes sociais, *blogs*, *sites* pessoais, entre outros. Essas características proporcionam lazer e conforto aos usuários, alavancam os ganhos das empresas e tornam processos (administrativos, por exemplo) mais ágeis. Com o rápido crescimento de sistemas e aplicações sendo criados e migrados para a *Web*, questões referentes a segurança de operações realizadas neste ambiente tornam-se cada vez mais importantes.

De acordo com estatísticas presentes no *site* do Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil (CERT.br) (CERT, 2012), nos últimos cinco anos, o número de ataques *Web* aumentou de 4201 no ano de 2008, para 25557 ataques em 2012. Portanto, é cada vez mais frequente ataques destinados à aplicações *Web*.

De forma contrária as estatísticas, Nakamura e Geus (2007) mencionam que é comum organizações investirem apenas em um *firewall* tradicional. Entretanto, um filtro de pacotes convencional, que trabalha nas camadas de rede e transporte, não detecta ou evita ataques que exploram vulnerabilidades em nível de aplicação. Dessa forma, torna-se necessária uma técnica capaz de auxiliar no monitoramento de ataques às aplicações que residem no ambiente *Web*. É interessante que esta técnica não apenas identifique o ataque, mas também bloqueie o acesso ao dado requisitado em tempo de execução, pois se o ataque atingir a aplicação alvo, os dados do usuário podem ser comprometidos.

Fonseca, Vieira e Madeira (2009, tradução nossa) relatam que “embora haja uma preocupação crescente com a segurança, existem alguns fatores importantes que tornam a proteção à aplicativos *Web* uma tarefa difícil de cumprir”. Fonseca, Vieira e Madeira (2009, tradução nossa) enumeram esses fatores:

- O mercado de aplicativos *Web* está crescendo rapidamente, resultando em uma enorme proliferação de aplicações *Web*, em grande parte alimentada pela (aparente) simplicidade em desenvolver e manter aplicações.
- Aplicações *Web* são altamente expostas a ataques.
- É comum encontrar desenvolvedores e administradores de aplicações *Web* sem o conhecimento necessário, ou experiência, na área de segurança.

Existe um ambiente propenso a ataques, acrescidos de serviços disponibilizados de forma negligente (que podem ser utilizados para atacar outros serviços), e profissionais com pouca ou nenhuma experiência em construir e manter aplicações seguras. Dessa forma, para minimizar a probabilidade de ataques com sucesso, estratégias devem ser tomadas desde o desenvolvimento inicial da aplicação e mantidos, enquanto a aplicação estiver disponível.

Complementarmente, Teodoro e Serrao (2011) avaliam que problemas de segurança devem-se, dentre outros fatores, a uma fraca implementação do Ciclo de Desenvolvimento de Software (SDLC – *Software Development Life-Cycle*). Os autores citam que a pouca experiência de equipes de desenvolvimento com a construção de *software* orientado à segurança contribui para a entrega de *softwares* com vulnerabilidades. Uma ferramenta que force o programador e a equipe de desenvolvimento a prever os dados que podem ser trafegados na aplicação durante o SDLC é uma possível solução para o problema. Sumarizando, ferramentas que auxiliem na prevenção de ataques em tempo de execução e que auxiliem no SDLC, tornam-se importantes para prover um maior nível de segurança aos dados críticos presentes em diversos sistemas *Web*.

O presente trabalho apresenta a ferramenta UniscanWAF, um *Web Application Firewall* (WAF), capaz de detectar comportamentos anômalos e interceptar, com base em regras pré-definidas (*blacklist*), ações maliciosas contra sistemas *Web*. O UniscanWAF, diferentemente de outras propostas, é utilizado também durante o Ciclo de Desenvolvimento de Software, permitindo que somente recursos cadastrados durante o desenvolvimento e manutenção de determinada aplicação em uma lista confiável de acesso (*whitelist*) sejam acessíveis. A ferramenta proposta representa uma extensão ao Uniscan (ROCHA; KREUTZ; TURCHETTI, 2012), o qual é um *scanner* convencional de vulnerabilidades em sistemas *Web*. O Uniscan fornece um diagnóstico do sistema testado, identificando todas as páginas ou URLs problemáticas do sistema. Ele está disponível no SourceForge (ROCHA, 2012) e na distribuição *Linux BackTrack*. A variação do Uniscan, isto é, UniscanWAF, possibilita a adição de uma funcionalidade híbrida, ou seja, pode-se trabalhar com a política de **liberar tudo** que pertencer as regras da *whitelist* e/ou **bloquear tudo** que pertencer à *blacklist*.

O restante deste trabalho esta organizado da seguinte maneira: no Capítulo 2 são abordados trabalhos relacionados e que influenciaram na proposta deste trabalho. No Capítulo 3 são explicados alguns conceitos importantes para o entendimento do trabalho. No Capítulo 4 são explicadas a proposta e implementação da ferramenta UniscanWAF, bem como a sua integração ao ciclo de desenvolvimento de aplicações *Web*. No Capítulo 5 são apresentados os resultados dos testes efetuados. Por fim, no Capítulo 6 são apresentadas as considerações

finais e abordados alguns trabalhos futuros.

2 TRABALHOS RELACIONADOS

Há na literatura diversas propostas que visam identificar e corrigir ao máximo as vulnerabilidades presentes em uma aplicação *Web* durante todo o ciclo de vida da aplicação. Isso quer dizer que a aplicação não é verificada somente antes da sua disponibilização para os usuários, como também a cada atualização e nova funcionalidade implementada. Dentre as propostas, existem trabalhos que sugerem a adoção de um SDLC orientado à segurança, ou seja, que segurança seja considerada desde os estágios iniciais de desenvolvimento da aplicação e se estendam durante todo o SDLC e, posteriormente, em ambiente de produção.

Antunes e Vieira (2012) definem que no mínimo três fases do modelo de SDLC tradicional em cascata (*Waterfall*) devem ser modificadas para aumentar a segurança no SDLC; são elas: Implementação (*Implementation*), Teste (*Testing*) e Implantação (*Deployment*). A principal orientação citada pelos autores na fase de implementação, é utilizar as melhores práticas (*best practices*) de programação, por exemplo, efetuar a validação dos dados de entrada e saída das aplicações. Na fase de teste, os autores indicam tipos de ferramentas que auxiliam à avaliar a segurança no SDLC, como ferramentas de testes de invasão, de análise estática, de análise dinâmica e ferramentas de detecção de anomalia em tempo de execução. Por último, na fase de implantação, os autores sugerem que ferramentas como Sistema Detector de Intrusão (*Intrusion Detection System* - IDS) e *Firewall* de Aplicação *Web* (*Web Application Firewall* – WAF) podem ser utilizados para aumentar a segurança da aplicação e manter o sistema protegido de ataques.

Outra proposta é a feita por McGraw (2004). A diferença principal nessa proposta é a abrangência maior, mas uma profundidade menor em comparação aos autores anteriores. O autor trata a segurança da aplicação em todas as fases do SDLC, e não em apenas três. Em especial, o autor propõe a adição de requisitos de segurança na fase de análise de requisitos, e a adição da fase de análise e teste de riscos na fase de *Design* do sistema. Essas fases são importantes, pois ajudam o projetista a considerar segurança de aplicações durante a concepção da ferramenta. O autor define também ferramentas de análise estática e ferramentas de testes de invasão. No entanto, as fases detalhadas no trabalho de Antunes e Vieira (2012) são mais claras quanto as ferramentas que devem ser utilizadas na fase de implantação, por exemplo, ferramentas de IDS, IPS (*Intrusion Prevention System*) e WAF. Notadamente, essas ferramentas são essenciais para monitorar em tempo real os acessos ao

sistema, pois podem proteger o sistema de vulnerabilidades possivelmente não detectadas durante a execução, em determinada fase do SDLC, das ferramentas responsáveis por encontrar vulnerabilidades, por exemplo, ferramentas de teste de invasão.

Foco do presente trabalho, a classe de ferramentas citadas anteriormente (WAF, IDS e IPS) costuma ser utilizada quando uma aplicação a ser protegida encontra-se em ambiente de produção. Assim, essas ferramentas atuam somente na fase de Manutenção do ciclo de desenvolvimento, diferentemente do UniscanWAF, que contribui ativamente durante a fase de Codificação e Testes, além da fase de Manutenção. Em especial, os três tipos de ferramentas citados podem trabalhar com abordagens *blacklist* e/ou *whitelist*. Ataques conhecidos são bloqueados por uma *blacklist*, enquanto que uma *whitelist* possibilita que ataques não previstos ou desconhecidos também sejam bloqueados. Enquanto a *blacklist* pode ser gerada para mais de um serviço *Web*, em que a lista pode ser um modelo genérico que contém assinaturas que independem do que esta sendo protegido, uma *whitelist* contém informações específicas de cada serviço disponibilizado, uma vez que deve ser configurada com informações do que deve ser explicitamente permitido em cada serviço. Os conceitos de *blacklist* e *whitelist* serão explicados no Capítulo 3.

Outro ponto a ser observado no escopo deste trabalho, é que a informação que deve ser inserida na *whitelist* e que representa o que é permitido para determinado serviço varia de acordo com a aplicação em questão. Em uma *whitelist* tradicional, é comum conter valores como: nome do usuário, número do usuário, dados pessoais, dados de validação de cadastro, entre outros. Esse tipo de *whitelist* consegue delimitar que tipo de usuário pode acessar o sistema, restringindo possíveis ataques. Entretanto, não pode impedir um ataque *Web* de ocorrer, visto que um possível usuário autorizado pode realizar requisições maliciosas. O UniscanWAF possui uma *whitelist* onde é possível cadastrar quais tipos de dados podem ser aceitos para cada recurso *Web*, de forma que, por mais que um usuário seja autorizado no sistema, ele não poderá realizar requisições que não sejam as explicitamente permitidas.

Em nenhum dos modelos apresentados por Antunes e Vieira (2012) e por McGraw (2004), é obrigatório que os dados utilizados pela aplicação a ser desenvolvida sejam validados pelos desenvolvedores, sendo possível que desenvolvedores inexperientes, ou até mesmo desenvolvedores experientes mas sem conhecimento de segurança, não tratem as possíveis fontes de vulnerabilidades. Embora existam ferramentas que auxiliem no processo de encontrar vulnerabilidades em aplicações, é imperativo que seja considerado os possíveis problemas de segurança que uma aplicação pode possuir. Em um ambiente *Web*, isso é ainda mais crítico. O exemplo mais frequente é não tratar os pontos de entrada do sistema, ou seja,

os locais onde o usuário pode incluir informação. Como nas propostas pesquisadas não há nenhum mecanismo que impeça o desenvolvedor de disponibilizar o recurso, é frequente acontecer casos de funcionalidades de determinada aplicação serem disponibilizadas com vulnerabilidades. Nesse aspecto, o presente trabalho diferencia-se das propostas das ferramentas existentes atualmente, pois implementa um mecanismo que impede que o desenvolvedor torne disponível uma funcionalidade que não tenha sido explicitamente liberada. A grande vantagem é que, para liberar um recurso, deve-se definir os possíveis valores que o recurso *Web* a ser liberado aceita como parâmetros e valores para estes parâmetros. Dessa forma, o desenvolvedor é forçado a pensar em segurança.

3 CONCEITOS RELACIONADOS

Este capítulo tem como objetivo explicar conceitos referentes a um WAF, foco do presente trabalho. A construção de um WAF, no entanto, não é trivial e a gama de definições e conceitos é demasiadamente extensa para ser discutida em profundidade. Portanto, serão somente abordados os conceitos essenciais para o entendimento de como funciona um WAF e como foi projetado o WAF proposto no presente trabalho.

3.1 Definição de WAF

Segundo IPA (2011, p. 9, tradução nossa), "o WAF é um *hardware* ou *software* que protege aplicações *Web* de ataques que exploram as vulnerabilidades nas aplicações *Web*". Sua função é inspecionar as transações entre os usuários e a aplicação *Web*, baseado em regras criadas pelo operador da aplicação. Ressalta-se que "o WAF é uma medida que minimiza o impacto dos ataques, e não uma solução definitiva que elimina as vulnerabilidades em uma aplicação *Web*" e que os seguintes benefícios são esperados: proteger aplicações *Web* de ataques que tentam explorar vulnerabilidades, detectar ataques que tentam explorar vulnerabilidades e proteger múltiplas aplicações *Web* de ataques. A Figura 1, demonstra ilustrativamente como funciona um WAF.

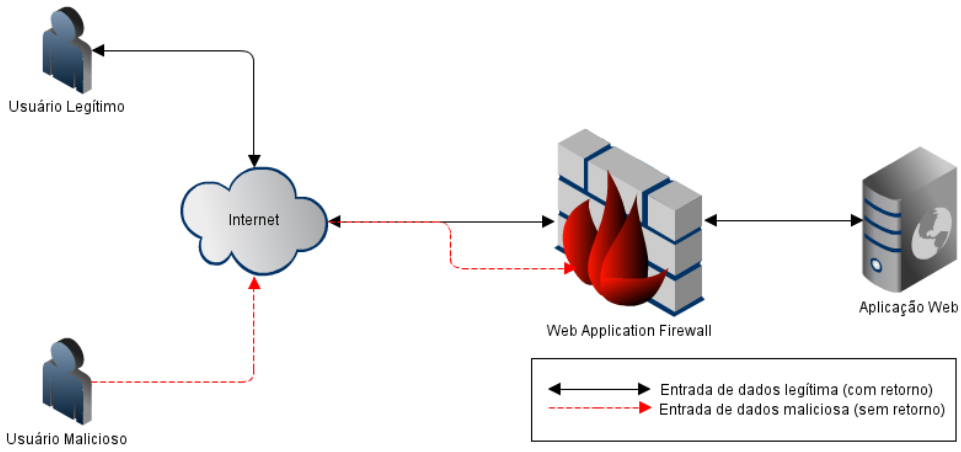

Figura 1 – Funcionamento de um WAF
Fonte: Adaptado de IPA (2011).

Observa-se na Figura 1 dois tipos de usuário. O usuário legítimo e o usuário malicioso. A seta indicada pela cor preta (e com flechas nas duas extremidades), indica o fluxo de mensagens executadas com uma mensagem válida. Nela se pode perceber que a mensagem enviada pelo usuário chega até a aplicação *Web*, que retorna o que o usuário solicitou. Já no fluxo indicado pela linha tracejada, e flecha apenas na extremidade do sentido de envio, pode-se observar que o usuário malicioso envia uma mensagem visando explorar uma vulnerabilidade, e o WAF nega o acesso, sem responder ao usuário. O fato de responder ou não as mensagens ao usuário malicioso varia de acordo com a aplicação. Por exemplo, um WAF pode decidir retornar uma página com o erro 403 (*Forbidden*) quando detectar uma requisição maliciosa. No exemplo da Figura 1, o WAF tem como política de segurança não responder às requisições que forem detectadas vulnerabilidades.

3.2 Diferença entre WAF, Firewall, IDS e IPS

No capítulo anterior, foram citadas ferramentas que poderiam ser incorporadas ao SDLC de um *software* de modo a agregar segurança. Entretanto, é preciso diferenciar essas tecnologias e posicionar onde exatamente atua um WAF, foco do presente trabalho.

É comum que muitas empresas utilizem somente um *firewall* como medida de segurança da infraestrutura de sua empresa. Segundo IPA (2011, p. 10, tradução nossa), "um *firewall* é um *hardware* ou um *software* que impõe controle de acesso baseado na informação de origem e destino (como endereço IP e portas) nos pacotes*"*. Quando o autor comenta informação, significa até a camada de transporte, no máximo. Isso significa que ataques presentes no *payload* do pacote não serão identificados por este tipo de filtro (NAKAMURA; GEUS, 2007).

Para o problema apresentado no parágrafo anterior, existem três ferramentas que podem ser utilizadas como solução. Um IPS/IDS, assim como um WAF, segundo IPA (2011, p. 11, tradução nossa) é um *software* ou *hardware* que inspeciona as transações para diversos dispositivos baseados em regras definidas pelo operador". Um IPS pode prevenir ataques que vão além do escopo de um WAF, como por exemplo, ataques que exploram vulnerabilidades do sistema operacional (IPA, 2011, p. 11). Além disso, um IPS bloqueia ataques de acordo com uma *blacklist*, uma lista de regras que define valores e padrões inaceitáveis para o sistema em questão (IPA, 2011). Um WAF é focado somente em ataques contra aplicações

Web, e pode utilizar além de uma *blacklist,* uma *whitelist*.

A diferença entre um IDS e um IPS está no fato de que o IDS visa apenas a detecção do ataque, enquanto um IPS detecta um ataque e age na prevenção de do mesmo (NAKAMURA; GEUS, 2007). Portanto, as definições e diferenças citadas para um IPS, posicionado de forma *inline* na rede (os pacotes obrigatoriamente passam pelo IPS antes de atingirem seu destino), se aplicam também a um IDS, posicionado de forma passiva na rede (os pacotes analisados são uma cópia dos pacotes destinados a rede).

3.3 Blacklist vs. Whitelist

A ação tomada por uma ferramenta que filtra pacotes em uma rede de computadores implica na política de segurança de cada empresa. Em outras palavras, pode-se prever tudo que se deseja permitir acesso através do conceito de uma *whitelist*, ou bloquear tudo que estiver especificado em regras utilizando para isso uma *blacklist*.

Segundo IPA (2011, p. 22, tradução nossa), uma *blacklist* é "uma lista de regras que definem valores e padrões inaceitáveis para transações HTTP". Isso significa, que no momento de inspecionar uma requisição, o WAF procura por valores que reflitam os cadastrados na lista. Se encontrar, detecta uma requisição maliciosa, senão é considerada uma requisição normal.

Uma *whitelist*, segundo IPA (2011, p. 22, tradução nossa), é "uma lista de regras que definem valores e padrões permitidos e aceitáveis para transações HTTP". Ou seja, se o conteúdo presente em uma requisição HTTP não corresponder com o cadastrado na *whitelist*, a transação é caracterizada como maliciosa. Caso contrário, a transação é dita normal.

Ao utilizar uma abordagem com *blacklist*, o WAF pode proteger aplicações *Web* de ataques conhecidos, devendo ser mantida atualizada frequentemente. Uma vantagem é que a mesma *blacklist* pode ser utilizada para diversas aplicações (IPA, 2011). Por outro lado, utilizando uma abordagem com *whitelist*, um WAF pode prevenir aplicações *Web* de ataques que os desenvolvedores não consideraram durante o SDLC (IPA, 2011), diferentemente de uma *blacklist*. Isso acontece porque somente o que está cadastrado na *whitelist* será processado pela aplicação *Web* alvo. Segundo IPA (2011, p. 22) é uma desvantagem de uma *whitelist*, que a cada nova atualização ou recurso adicionado, deve-se atualizar a lista. É importante ressaltar que é justamente essa característica que é utilizada para fazer com que o

desenvolvedor venha a pensar nos parâmetros que serão cadastrados na *whitelist* para cada recurso desenvolvido. Portanto, na proposta apresentada nesse trabalho, essa desvantagem não se aplica. A implementação da *whitelist* é explicada na Seção 4.4.2.

4 PROPOSTA

Prasad e Ramakrishna (2011) ressaltam que o desenvolvimento de aplicações *Web* distingue-se do desenvolvimento de aplicações tradicionais por dois motivos. O primeiro é pelo rápido crescimento dos requisitos das aplicações *Web*. O segundo é pela rápida mudança de conteúdo, ou seja, as aplicações *Web* evoluem e sofrem constantes mudanças. Não é possível especificar em sua totalidade o que uma aplicação *Web* irá conter no início do SDLC. Os autores deixam claro ainda, que o desenvolvimento de sistemas *Web* não é um evento único, como atualmente a maioria dos desenvolvedores percebem, mas sim, um processo interativo com um longo SDLC. Ou seja, por essa dinamicidade característica de sistemas *Web*, questões de segurança tornam-se ainda mais importantes. Somado a isso, devido ao ambiente em que se encontram, sistemas *Web* são alvos constantes de ataques.

4.1 O UniscanWAF

Com o objetivo de minimizar o efeito e as ações de ataques a sistemas *Web*, este trabalho propõe e avalia o UniscanWAF, uma ferramenta capaz de detectar comportamentos anômalos e interceptar, com base em uma *whitelist* e/ou uma *blacklist*, ações maliciosas contra sistemas *Web*. O UniscanWAF representa uma extensão do Uniscan, um *scanner* de vulnerabilidades para sistemas Web . O Uniscan envia requisições maliciosas para o sistema a ser testado e avalia a resposta enviada para determinar se o sistema é vulnerável a determinada vulnerabilidade testada. Ao final, o Uniscan gera um relatório com todas as páginas do sistema que possam conter problemas. O UniscanWAF trabalha com as mesmas regras do Uniscan, e faz uso da arquitetura modular do mesmo.

A principal diferença é que a ferramenta proposta atua de forma similar a um *proxy Web* com filtro de pacotes, ou seja, é capaz de analisar e interceptar requisições maliciosas em tempo de execução. O fato de o UniscanWAF trabalhar de forma híbrida (*blacklist* e *whitelist*) para decidir se libera acesso a recursos *Web,* provê proteção contra novos ataques através da *whitelist,* permitindo que apenas recursos nela cadastrados sejam acessados pelos usuários. A proteção contra ataques conhecidos é feita através da utilização de uma *blacklist,* ou detector

de vulnerabilidade, que atua na análise de ataques através de regras pré-definidas. Para entender como esse processo funciona, é importante conhecer o funcionamento geral do UniscanWAF, para posteriormente entender os módulos que o compõe em detalhes. A Figura 2 mostra o funcionamento da ferramenta de forma simplificada.

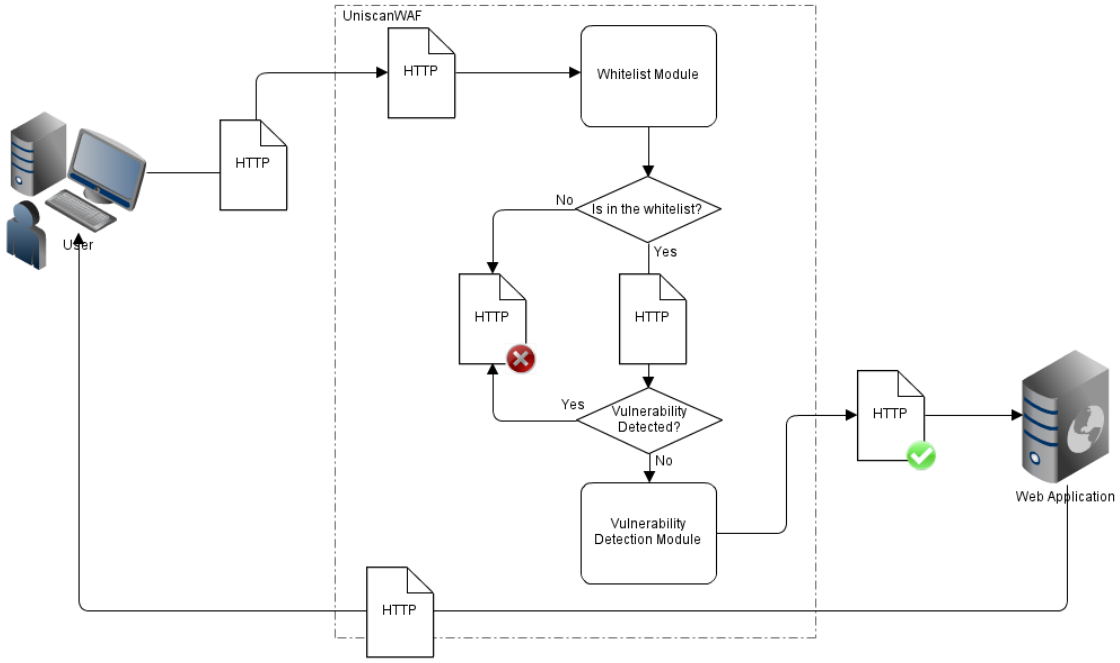

Figura 2 – Funcionamento do UniscanWAF.
Fonte: do Autor.

Como pode-se observar, o UniscanWAF fica localizado entre o usuário e a aplicação *Web*, monitorando todas as requisições HTTP com destino ao servidor *Web*. Na Figura 2, um usuário envia uma requisição HTTP com destino a uma aplicação *Web* (indicado por *Web Application*). O UniscanWAF captura essa requisição e analisa se ela está presente na lista de requisições permitidas (*whitelist*) através do Módulo *Whitelist* (indicado por *Whitelist Module*). Caso negativo, essa requisição é descartada. Opcionalmente, pode ser configurada uma página de retorno com uma mensagem de erro para ser apresentada ao usuário. Aconselha-se fortemente, que essa página não contenha informações sobre o erro, visto que um atacante pode obter vantagem das mensagens retornadas para cada erro. Caso afirmativo, a requisição é enviada para o Módulo de Detecção de Vulnerabilidade (indicado por *Vulnerability Detector Module*) que trabalha na abordagem de *blacklist* e analisa se a requisição possui algum ataque *Web* conhecido. O módulo detecta somente os tipos de ataques

cadastrados na base do UniscanWAF. Se a requisição estiver livre de ataques, é enviada à aplicação *Web*. Caso contrário é descartada.

4.2 Delimitações e considerações quanto a execução do ambiente

É importante delimitar o cenário a que a ferramenta se propõe operar, de modo a definir com clareza que tipo de ambiente ela pode ser executada. O cenário de execução proposto é composto por uma ferramenta interceptando todas as requisições HTTP entre usuários externos e aplicações *Web* hospedadas na organização. Considera-se uma ou mais aplicações *Web*, presentes em um mesmo domínio administrativo, construídas utilizando-se o modelo de SDLC *Waterfall*.

As ferramentas podem ser implementadas sob qualquer linguagem de programação. Assume-se que o programador, no momento da codificação, informou todos os parâmetros e valores relacionados a cada recurso que o mesmo implementou no modelo de dados proposto com a sintaxe correta.

Podem haver dois cenários:

- Cenário I: há uma máquina que analisa as requisições recebidas para todas as aplicações *Web* que estão em servidores separados.
- Cenário II: uma análise é realizada em cada máquina que hospeda a aplicação *Web*.

Para o primeiro caso, assume-se que não exista qualquer tipo de interceptação em qualquer sentido da comunicação entre a ferramenta desenvolvida que intercepta as requisições e as aplicações distribuídas em diferentes servidores. Para o segundo caso, e também o primeiro, assume-se que não há qualquer programa malicioso no servidor para interceptar a mensagem a nível de Sistema Operacional.

Este trabalho não trata requisições criptografadas como o uso do HTTPS. Para o tratamento e filtragem das mensagens HTTP, assume-se a existência de uma ferramenta que redirecione as mensagens destinadas à aplicação *Web* para o UniscanWAF.

4.3 Mudanças propostas no SDLC

A ferramenta desenvolvida neste trabalho foi projetada para ser utilizada de forma integrada ao SDLC, de modo que, faz-se necessário introduzi-la ao SDLC da aplicação que se deseja proteger de ataques *Web*. Essa decisão foi tomada baseada no fato de muitos desenvolvedores, no momento de projetar e codificar aplicações *Web*, não levarem em consideração questões de segurança. Dessa forma, a utilização do modelo de *whitelist* proposto visa forçar os codificadores e projetistas a pensar em segurança e em cada tipo de dado que trafega na aplicação. O presente trabalho propõe um WAF que deve ser utilizado durante o SDLC, e não somente quando a ferramenta se encontra em ambiente de produção. Isso é necessário devido a ferramenta utilizar informações que devem ser recolhidas durante a fase de Codificação para alimentar o UniscanWAF. Se as informações sobre determinado recurso não forem fornecidas, esse recurso não estará acessível aos usuários.

O modelo proposto atua em duas fases distintas do SDLC tradicional *Waterfall,* e também em uma terceira fase opcional. Na Figura 3, pode-se notar os quadros grifados que indicam as fases que o modelo proposto traz inovações no quesito segurança. As demais fases representam as fases tradicionais do modelo supracitado.

O modelo proposto correlaciona as fases de Codificação (*Code*) e Manutenção, de forma que o bom funcionamento da segunda, depende da execução correta da primeira. Essa correlação é alcançada através da implementação do UniscanWAF, uma vez que as requisições a serem permitidas pela ferramenta na fase de Manutenção são obtidas da fase de Codificação. A ferramenta é a responsável pela implementação do modelo de dados, isto é, da *whitelist* que determina se uma requisição é maliciosa.

Portanto, o UniscanWAF implementa e executa o modelo proposto em um SDLC convencional. A seguir, é explicado como o UniscanWAF é integrado ao SDLC, e como é o modelo de dados que gera a *whitelist*.

A Figura 3 ilustra todas as fases do modelo proposto integrado ao modelo tradicional *Waterfall.* Ao lado esquerdo da Figura 3 encontram-se as informações do SDLC referentes a proposta do presente trabalho e, ao lado direito encontram-se as informações referentes ao modelo clássico. Dessa maneira, as fases de Requisitos e Casos de Usos*, Design* e Planos de Testes permanecem inalteradas e devem ser seguidas de acordo com as diretrizes apresentadas no modelo utilizado pela equipe responsável.

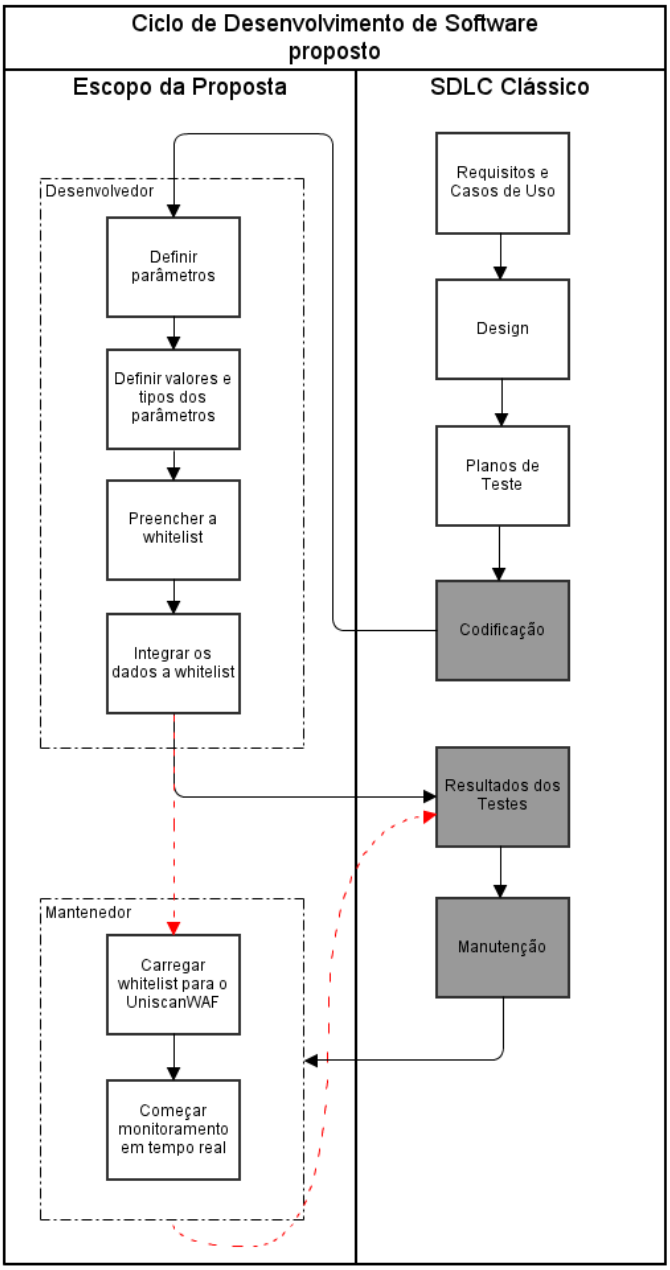

Figura 3 – Mudanças propostas no SDLC
Fonte: Adaptado de Xiong e Peyton (2010).

A partir da fase de Codificação, o presente trabalho começa a influenciar de forma direta no SDLC. Normalmente, de posse das especificações da funcionalidade, o desenvolvedor efetua a implementação e então, disponibiliza o recurso para que sejam efetuados testes. Entretanto, no modelo proposto, o desenvolvedor deve definir informações sobre a funcionalidade que ele irá implementar. Esse processo é explicado através do quadro Desenvolvedor da Figura 3. Assim que receber uma especificação de uma funcionalidade, seja

para refatoração ou implementação de uma nova funcionalidade, o desenvolvedor deve de acordo com a especificação recebida definir os **parâmetros** que a funcionalidade requer. Exemplo: uma nova funcionalidade para cadastro de dados pessoais, tal como "*/pessoas/cadastro.php*", devem conter os possíveis parâmetros para esse recurso: *nome, endereco, telefone*, *estado,* etc. Feito isso, o desenvolvedor deve procurar na especificação que tipo de dados cada parâmetro pode receber, e então definir o **tipo** que cada parâmetro pode receber. Existem três tipos de parâmetros que o UniscanWAF permite definir: valores estáticos, por exemplo, o parâmetro estado só pode receber RS, SP ou RN; expressões regulares, para definir, por exemplo, restrições e o formato do que pode ser informado, como, definir que telefone só pode receber números e que esses números devem ter nove dígitos; por último, pode ser definida uma consulta a base de dados para determinado parâmetro, com o intuito de comparar se o informado pelo usuário está no resultado da consulta esperada pela aplicação para determinado parâmetro.

Essas informações serão utilizadas pela ferramenta UniscanWAF para montar a *whitelist,* responsável por permitir que somente recursos presentes nessa lista sejam acessíveis aos usuários (fase de Manutenção). De posse de todas as informações, o desenvolvedor deve popular o modelo de dados. Dessa forma, é obrigatório que o desenvolvedor considere o tráfego dos dados e consequentemente trate questões de segurança enquanto desenvolve as funcionalidades designadas. Isso ocorre quando o desenvolvedor precisa definir os valores que um parâmetro pode aceitar, pois é preciso fazer uma avaliação do que definir como um valor aceitável. Um exemplo comum seria decidir se um parâmetro que corresponde ao nome de um arquivo pode conter caracteres especiais ou a sequência "../".

Por fim, devem ser reunidas todas as informações geradas por todos os desenvolvedores que realizaram o processo descrito acima, para gerar um único modelo de dados consistente, que represente todos os recursos a serem protegidos. Esse é o último processo realizado pelo desenvolvedor, como pode ser observado na Figura 3 em Desenvolvedor. Se alguma atualização na aplicação for feita, o modelo de dados deve ser revisto seguindo o mesmo processo descrito nesta seção.

Seguindo o ciclo de desenvolvimento, a próxima etapa é enviar o recurso para teste, e então, quando todos os recursos tiverem sido testados, será iniciada a fase de Manutenção. Nessa fase, as informações fornecidas no modelo de dados na fase de Codificação devem ser carregadas para o UniscanWAF, que montará a *whitelist* com os valores obtidos desse modelo. É nessa fase também, que são carregadas as regras de vulnerabilidades implementadas pelo UniscanWAF.

Opcionalmente, ao invés de enviar os recursos para teste logo após acabar a implementação da funcionalidade e o preenchimento do modelo de dados para gerar a *whitelist*, pode-se enviar essas informações diretamente para o UniscanWAF montar a *whitelist* e realizar os testes com o UniscanWAF já em funcionamento (processo indicado pelas flechas pontilhadas na Figura 3). Ou seja, os testes podem ser realizados com a presença da ferramenta entre o recurso a ser acessado e a ferramenta de testes (no caso de uma ferramenta de testes de invasão). Dessa forma, a ferramenta de testes realizaria os testes somente sob os recursos permitidos na *whitelist* e somente para os parâmetros e os valores permitidos para esses recursos.

Um ponto importante a ser esclarecido é como acontece a manutenção, tanto da *whitelist*, quanto da *blacklist,* e a vantagem de mantê-las atualizadas. A *whitelist* influencia nas fases de Codificação, Testes e Manutenção. Esse processo foi explicado nesta seção. Assim como a *whitelist*, a *blacklist* também possui papel importante no SDLC. A diferença é que a abrangência é um pouco menor. A *blacklist* atua somente na fase de Manutenção, pois como explicado anteriormente, possui regras que independem de questões de implementação, e portanto, podem ser utilizadas para proteger diversas aplicações sem conhecer nenhuma característica de implementação. A *blacklist*, que contém um conjunto de regras responsáveis por efetuar a detecção de ataques, no entanto, precisa ser atualizada frequentemente para que seja capaz de detectar novos ataques.

Basicamente, a *whitelist* é utilizada para refletir aspectos de novas implementações e/ou mudanças na aplicação, ao passo que a *blacklist* vai possuir mudanças quando forem detectadas novas regras a serem incluídas no ambiente para novas vulnerabilidades descobertas, ou o caso de necessitar uma atualização de uma regra.

A combinação dessas duas abordagens fornece algumas vantagens. Por exemplo, mesmo que a *blacklist* não esteja atualizada, pode haver o caso da requisição ser bloqueada pela *whitelist*, fornecendo uma segurança maior. Da mesma forma, se um desenvolvedor cadastrar um tipo de dados permitido para um parâmetro erroneamente, e a informação possibilitar um ataque, o ataque pode ser bloqueado pela *blacklist* conforme as regras contidas nela. Na seção 4.4 são explicados os módulos do UniscanWAF responsáveis por implementar os conceitos de *whitelist* e *blacklist* utilizados nessa seção.

4.4 Arquitetura do UniscanWAF

O UniscanWAF possui uma arquitetura modular. Isso significa que cada funcionalidade da ferramenta foi separada em módulos que são anexados a um núcleo do sistema, responsável por gerenciar esses módulos.

O UniscanWAF atua como um intermediário, similar a um *proxy*, entre o usuário e o servidor *Web*. Sendo assim, ele é capaz de analisar e repassar ou não as requisições para a aplicação *Web*, conforme a *whitelist* e as regras habilitadas para detecção de vulnerabilidades. A arquitetura do UniscanWAF é representada na Figura 4.

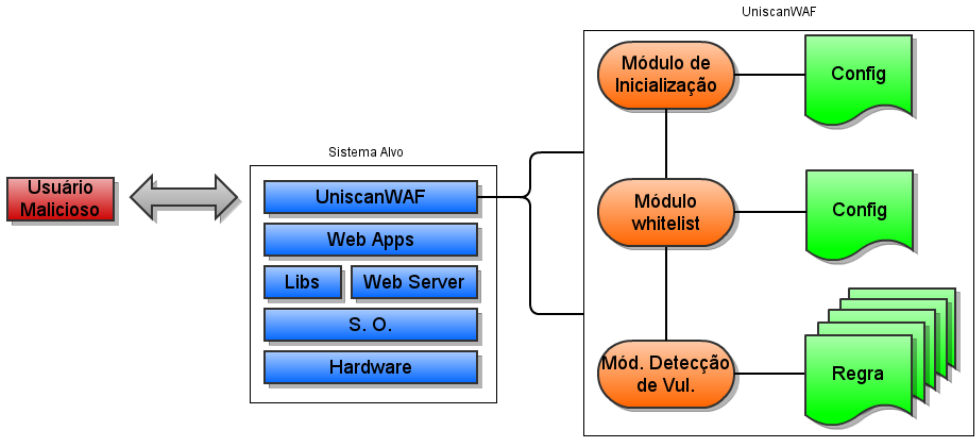

Figura 4 – Representação da arquitetura do UniscanWAF
Fonte: Adaptado de Rocha (2012).

Como pode ser observado, a arquitetura do UniscanWAF é composta por três módulos: o Módulo de Inicialização, o Módulo de *Whitelist* e o Módulo de Detecção de Vulnerabilidade. O Módulo de Detecção de Vulnerabilidade, adicionalmente, é composto por diversas regras responsáveis por identificar um ataque em uma requisição HTTP. O fato dessas regras serem acopladas ao Módulo de Detecção de Vulnerabilidade, e não estarem presentes diretamente nele, confere à ferramenta as características de extensibilidade e flexibilidade, pois não é preciso alterar o código-fonte do módulo para adicionar novas regras para novos ataques, ou editar regras de acordo com a necessidade do usuário. A Figura 5 mostra um fluxograma que explica o funcionamento da ferramenta UniscanWAF.

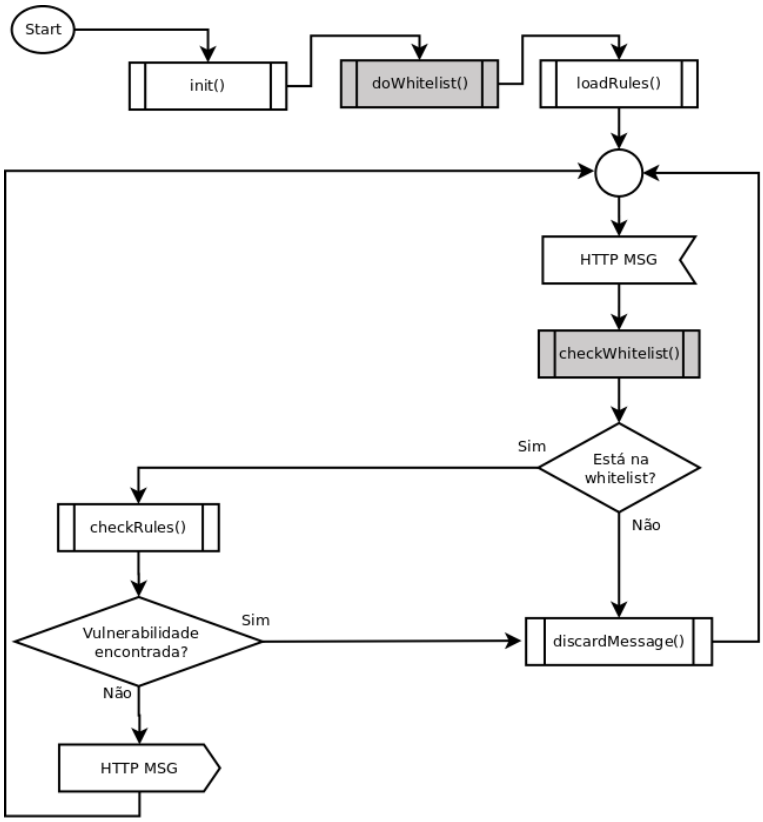

Figura 5 – Fluxo de Funcionamento do UniscanWAF.
Fonte: do Autor.

A Figura 5 mostra que no momento que a ferramenta é inicializada, a sub-rotina *init()* carrega todos os parâmetros de configuração (por exemplo, a localização do arquivo de *log* ou a porta TCP que o UniscanWAF utiliza para comunicação). Logo, a sub-rotina *doWhitelist()* é responsável por carregar a *whitelist* para a memória a partir de um arquivo de configuração. Isso é importante porque diminui o tempo de análise das requisições, uma vez que não é mais necessário fazer acesso a disco, uma operação sabidamente custosa a memória principal. Após, a sub-rotina *loadRules()* carrega as regras disponíveis para detecção de vulnerabilidades em requisições HTTP. Os passos descritos aqui simplesmente preparam a ferramenta para operar sobre as requisições HTTP recebidas.

Completado esses passos, o sistema fica aguardando por mensagens HTTP. Para a ferramenta receber essas mensagens é feito um redirecionamento de todo o fluxo HTTP para a porta que o UniscanWAF opera. Assim que receber uma requisição HTTP, essa requisição é examinada para ver se está presente na *whitelist*. Isso é feito através da sub-rotina *checkWhitelist()*. Caso não esteja presente, a sub-rotina *discardMessage()* fica responsabilizada de descartar a mensagem HTTP, sem retornar nada para o requerente. Caso

pertença, a mensagem HTTP ainda é analisada através das regras que procuram por vulnerabilidades conhecidas, por meio da sub-rotina *checkRules()*. Se nenhuma vulnerabilidade for encontrada, a mensagem é encaminhada ao destino original antes de ser analisada pelo UniscanWAF. Caso contrário, a mensagem é descartada pela sub-rotina *discardMessage()*. Logo após, a ferramenta passa a esperar por novas requisições.

4.4.1 Módulo de inicialização

Ao Módulo de Inicialização compreende a função de inicializar a ferramenta de acordo com os parâmetros definidos no arquivo de configuração. Algumas das possibilidades de configuração são: a ação tomada pelo módulo de Detecção de Vulnerabilidades no momento da detecção, que pode ser configurado para encerrar a conexão ou executar um *script*, por exemplo, gerar um relatório técnico, colocar o *host* de origem em quarentena, ou mesmo obter maiores informações com a execução de outras ferramentas que possibilite um diagnóstico detalhado. Outros exemplos de parâmetros de configuração incluem a localização do arquivo de *log* e a porta que o UniscanWAF utilizará para comunicação.

4.4.2 Módulo de whitelist

O Módulo de *Whitelist* tem por finalidade ler o arquivo que contém os recursos e os parâmetros que são permitidos, e carregar esses dados para a memória. A segunda função do módulo é comparar as requisições recebidas com os valores que foram carregados em memória. Se a requisição estiver presente na *whitelist*, o acesso ao recurso é permitido, caso contrário o acesso é negado. O algoritmo de funcionamento é explicado na subseção 4.4.2.2. Entretanto, para entendê-los é necessário analisar a composição da *whitelist*.

4.4.2.1 Composição da *whitelist*

Para a construção da *whitelist* considera-se um ou mais sistemas Web $S = \{s_1, s_2, ..., s_n\}$, com $s > 0$, cujos recursos Web fazem parte de cada um destes sistemas, onde S_n[1] $= \{r_1, r_2, ..., r_n\}$ possuam a si associados um conjunto de parâmetros $P = \{p_1, p_2, ..., p_n\}$, onde $p \geq 0$. Cada elemento de P, tem associado a si um ou mais tipos de parâmetros $T = \{t_{static}, t_{regexp}, t_{querydb}\}$, onde $\exists\ t_{static}\ v\ \exists\ t_{regexp}\ v\ \exists\ t_{querydb}$, em que t_{static} refere-se a parâmetros que não mudam na aplicação, t_{regexp} refere-se a parâmetros que especificam um conjunto de parâmetros aceitáveis e $t_{querydb}$ são parâmetros que fazem referência a uma consulta em uma base de dados. Além disso, cada tipo T, tem a si associados valores V, para $v \neq \emptyset$. Ou seja, se existirem parâmetros, os valores referentes a eles, bem como seus tipos, devem ser informados. Todas essas informações são especificadas durante a fase de codificação do sistema, e devem ser informadas pelo programador a cada recurso adicionado, para que então seja gerada a *whitelist.* A *whitelist* determinará os recursos que podem ser acessados e os parâmetros e valores que podem ser fornecidos pelo usuário.

Dentro dessa configuração podem surgir dois cenários para a geração da *whitelist*: ser gerada uma única *whitelist W* que contemple todos os sistemas da organização, onde W é um conjunto unitário $W = \{w_1\} \mid w_1 = \{r_1, r_2, ..., r_n\}$. Ou, uma *whitelist W* para cada sistema da organização, em que $W = \{w_1, w_2, ..., w_n\} \mid w_i = \{r_{i1}, r_{i2}, ..., r_{im}\}$, sendo $0 < i \leq m$.

4.4.2.2 Algoritmo de detecção da *whitelist*

A Figura 6 demonstra o algoritmo utilizado para verificar se determinada requisição HTTP está ou não presente na *whitelist*.

É importante notar que na linha 2 é inicializada a variável status em *true.* Essa variável é responsável por determinar se uma requisição está na *whitelist*, valor booleano *true*, ou em caso negativo, valor booleano *false*. O algoritmo fica esperando por novas mensagens HTTP. Assim que receber uma mensagem (linha 5), é extraído o conjunto de parâmetros r' para um recurso a ser acessado (linha 6). Logo após, é extraído o conjunto de valores dos parâmetros

1 Para cada sistema *Sn* há uma Whitelist *W1* composta pelos mesmos recursos de *S1*, sendo assim, a partir de agora, utiliza-se *W* para referir ambos conjuntos.

p' da requisição (linha 7) e o sistema ao qual é destinada a requisição (linha 8). Ressalta-se que o subscrito *s*, refere-se ao identificador do sistema Web ao qual a requisição foi enviada. Se o conjunto de parâmetros extraídos da requisição for vazio, ou seja, não existirem parâmetros na requisição, deve-se validar se é permitida uma requisição sem parâmetros (linha 10). Caso não esteja prevista na *whitelist*, a validade dos parâmetros que compõe um recurso *r* deve ser contemplada se, e somente se, todos os elementos que compõem *r* forem validados. Ou seja, como a *whitelist W* segue o formato de $W_1(r_{11}(p_{111},...,p_{11m}), r_{12}(p_{121},...,p_{12m}), r_{1n}(p_{11n},...,p_{1nm}))$, deve-se validar os valores que a compõe. Assim, é implementado um laço que valida cada parâmetro do conjunto de parâmetros r' informados pelo usuário e cada valor do conjunto de valores p' informados pelo usuário (linha 14 à 23).

```
1   received ← null
2   status ← true                          {true if present in whitelist, false if not present}
3   cobegin
4     loop
5       received ← received(httpMessage)
6       r'[] ← getResource(received)
7       p'[] ← getParam(received)
8       s = getHost(received)
9
10      if  r' ⊂ Ø ∧ (r_s ⊂ Ø ∧ r_s ⊂ (W_s)^rn_r0)  then
11        status ← true
12      else
13        i ← 0
14        loop
15          if  r_i' ⊄ (W_s)^rn_r0  then
16            status ← false
17          else
18            if  p_i' ⊄ (R_s)^pn_p0  then
19              status ← false
20            endif
21          endif
22          i ← i +1
23        endloop
24      fi
25    endloop
26  coend
```

Figura 6 – Processo para averiguar se uma requisição HTTP é um ataque.
Fonte: do Autor.

O laço iniciado na linha 14 é responsável por percorrer cada parâmetro do conjunto de parâmetros r' informados pelo usuário e verificar se determinado parâmetro faz parte da lista de parâmetros aceitáveis $(W_s)^{rn}_{r0}$ (linha 15). Caso negativo, a variável de controle status

recebe o valor false, indicando que o acesso ao recurso desejado não será disponibilizado devido ao parâmetro não estar relacionado como valor aceitável na *whitelist* (linha 16). Caso o parâmetro pertença ao conjunto de parâmetros aceitáveis $(W_s)_{r0}^{rn}$, deve-se verificar o valor p' informado pelo usuário como valor para o parâmetro (linha 18). Em outras palavras, deve-se verificar se o valor p' informado pelo usuário faz parte do conjunto de valores aceitos $(R_s)_{p0}^{pn}$ para o parâmetro acima validado. Caso pertença, a variável de status permanece com o valor atual. Caso não pertença, o acesso ao recurso é bloqueado, através da configuração da variável status para false (linha 19).

É possível observar que o algoritmo trabalha com considerações de refutação, isto é, se os casos analisados não forem contemplados pelas condições, a mensagem requisitada é válida sendo portanto repassada à aplicação *Web*.

4.4.3 *Blacklist*: módulo de detecção de vulnerabilidades

O módulo de Detecção de Vulnerabilidades é responsável por realizar a análise de todas as requisições dos clientes que tenham como destino o servidor *Web* a ser protegido. Para cada regra há um arquivo correspondente, que determina as ações a serem executadas pela ferramenta para detectar a existência ou não de uma vulnerabilidade na requisição que está sendo analisada. Com isso, novas regras podem ser adicionadas ao sistema sob-demanda, conferindo flexibilidade e simplicidade ao sistema. Na versão atual, o UniscanWAF é capaz de analisar e processar as vulnerabilidades *XSS (Cross-site Scripting)*, *LFI (Local File Include)*, *RFI (Remote File Include)*, *RCE (Remote Command Execution)* e *SQL-Injection*.

5 TESTES E RESULTADOS

Para garantir que as funcionalidades apresentadas no UniscanWAF realmente atendam os requisitos que a ferramenta se propunha, foram realizados testes a fim de validá-la. Existem duas funcionalidades da ferramenta que devem ser validadas: o módulo de *whitelist* e o módulo correspondente à *blacklist*, ou módulo de detecção de vulnerabilidade. Esta seção mostrará a metodologia e o resultado dos testes para cada módulo de forma separada.

5.1 Validações para implementação da Whitelist

É necessário comprovar a validade da *whitelist* para garantir que somente será permitido o acesso aos recursos nela previstos, portanto, seguem algumas relações que comprovam o modelo de *whitelist* proposto.

Cada recurso[2] r pode ser acessado por um número pré-definido de parâmetros, formando W_1 da seguinte forma: $W_1(r_{11}(p_{111},...,p_{11m}),\ r_{12}(p_{121},...,p_{12m}),\ r_{1n}(p_{11n},...,p_{1nm}))$. Cada parâmetro p_1 pode ser validado da seguinte forma:

$$\textbf{\textit{Lema 1:}}\ \forall\ p_{111} \wedge p_{111}' \in N,\ 0 < p_{111} < 2 \wedge 0 < p_{111}' < 2 \rightarrow p_{111} = p_{111}'$$

Prova. Para os valores dos parâmetros individuais serem validados, considere p_{111} ser um parâmetro definido em um recurso r_{1n}, deixe p_{111}' ser um parâmetro a ser validado. Para p_{111}' ter sua validade, p_{111}' deve ser igual a p_{111}, previsto em r_{1n}. Do ***Lema 1***, pode-se concluir que p_{111} e p_{111}' possuem o valor 1, logo $p_{111}=p_{111}'$.

Além da validade de cada parâmetro, os elementos de W_1 devem ser validados, isto ocorre através da seguinte forma:

Teorema 1: quaisquer que sejam os elementos de r_{11} e r_{11}', tem-se:

2 Neste trabalho, um recurso significa uma funcionalidade oferecida pelo sistema que implique na passagem, ou não, de parâmetros pelo usuário.

$$r_{11} \subset W_1,\ r_{11} \subset r_{11}' \wedge r_{11}' \subset r_{11} \rightarrow r_{11}'=r_{11}$$

Prova. A validade dos parâmetros que compõe um recurso r_{1n}, deve ser contemplada se e somente se todos os elementos que compõem r_{1n} forem validados. A validade individual de cada elemento de W_1 é verificada através da igualdade dos subconjuntos ($r_{11} \subset W_1$) existentes em W_1. Considere os recursos r_{11} e r_{11}', sendo que r_{11} está previsto em W_1 e r_{11}' é o conjunto validado na requisição do usuário, se e somente se, os elementos de r_{11}' existirem em r_{11}, ou seja, os subconjuntos devem ser iguais, mas seus elementos não necessitam estar na mesma ordem. Então, se r_{11} está contido em r_{11}' e r_{11}' está contido em r_{11}, logo comprovamos que r_{11} e r_{11}' são iguais.

Outra validade é realizada para os dados absolutos recebidos pela *whitelist*. Considerando que estes dados não possuem informações variadas, e que portanto não possuem parâmetros, a validade ocorre da seguinte forma:

$$r_{11} \subset \varnothing,\ \forall\, r_{11} \subset W_1 \wedge \forall\, r_{11}' \subset \varnothing \nrightarrow r_{11}' \not\subset L_{r\_drop}$$

Prova. Considerando que r_{11} seja o recurso disponibilizado em W_1, e que r_{11}' é o recurso acessado pelo usuário, como r_{11} é um conjunto vazio e está contido em W_1, r_{11}' para ser equivalente, deverá ser vazio. Logo, r_{11}' não fará parte na lista de recursos bloqueados em L_{r_drop}, pois será validado em W_1.

No ***Lema 1***, utilizou-se p para se referir aos parâmetros de um recurso r_n, ou seja, p são elementos de r_n. Entretanto, esses parâmetros possuem obrigatoriamente elementos tal que $P \not\subset \varnothing$. Logo, os elementos de P devem ser validados:

$$x \in P \wedge x \neq \varnothing,\ P' \subseteq P \nrightarrow x' = x$$

Prova. Sendo P' um conjunto unitário que contém o parâmetro x' digitado pelo usuário, e x o valor correspondente ao conjunto de valores que são aceitáveis para determinado parâmetro P. Dessa forma, x' será igual a um valor x se P' for um subconjunto próprio de P, visto que o conjunto P' é unitário. Então x' será igual a um valor x de P.

Portanto, prova-se que somente será aceito o valor de um parâmetro se ele estiver presente no conjunto de valores aceitáveis.

5.2 Testes no módulo de detecção de vulnerabilidades

O objetivo é apresentar os primeiros resultados de avaliação da funcionalidade de detecção de vulnerabilidades do UniscanWAF e comparar os resultados com o *ModSecurity* (MODSECURITY, 2013), um WAF *open source* projetado para trabalhar junto a servidores *Web Apache*. O *ModSecurity* por si só, não consegue prover um nível satisfatório de proteção para aplicações *Web*. Para tanto, a ferramenta necessita de regras. Essas regras são responsáveis por procurar padrões conhecidos na identificação de ataques em requisições HTTP.

Por padrão o *ModSecurity* não instala nenhuma regra, de modo que é preciso instalar separadamente um conjunto que proveem regras para serem utilizadas. Esse conjunto é denominado *OWASP ModSecurity Core Rule Set*. A OWASP (*Open Web Application Security Project*) é uma organização focada em melhorar a segurança de *softwares* e é parceira da Trustwave (TRUSTWAVE, 2013) no desenvolvimento desse conjunto de regras.

5.2.1 Vulnerabilidades Web detectadas pelo UniscanWAF

Para validar a eficiência do detector de vulnerabilidades proposto, foi criada uma aplicação teste contendo as cinco vulnerabilidades propostas no cenário deste trabalho. Cada vulnerabilidade, acompanhada de um exemplo de código vulnerável, é descrita a seguir.

5.2.1.1 *Cross-site Scripting*

A vulnerabilidade XSS ocorre quando uma aplicação inclui dados fornecidos pelo usuário sem a validação desses dados (OWASP TOP 10, 2013). O impacto desse tipo de

ataque inclui o eventual roubo de sessão. Aplicações que não tratam os dados de entrada adequadamente estão sujeitas a esse tipo de ataque. Esse tipo de vulnerabilidade passa, com uma certa frequência, desapercebido pelos desenvolvedores. De maneira a amenizar esse problema, estratégias de segurança presentes no SDLC são recomendadas (CHESS; MCGRAW, 2004).

```php
<?php
print "Pagina: " . urldecode($_SERVER["REQUEST_URI"]);
?>
```

Figura 7 – Código-fonte PHP vulnerável à XSS.
Fonte: do Autor.

Na Figura 7 é apresentado um exemplo de código-fonte PHP vulnerável ao ataque XSS. No caso em questão, o trecho de código imprime o recurso que foi requisitado. Consequentemente, ao requisitar uma URL como "*http://sistemaWeb.com.br/index.php?name=teste*", será exibido "Pagina: */index.php?name=teste*". Entretanto, há outras possibilidades de ações, como no caso de uma URL similar a "*http://sistemaWeb.com.br/index.php?nam e=<script>alert('XSS')</script>*", que, no caso, irá criar uma caixa com o conteúdo "*XSS*" e um botão "*OK*". Isso significa que um atacante consegue executar ações e códigos diversos, tornando o ataque potencialmente comprometedor para o sistema *Web*, ou ambiente alvo, como o sistema hospedeiro do servidor *Web*.

5.2.1.2 *Local File Include*

A vulnerabilidade LFI torna possível a inclusão de arquivos locais. Este ataque pode expor arquivos e dados dos sistemas hospedeiros da aplicação *Web*.

```php
<?php
$arquivo = $_GET['arg'];
include($arquivo);
?>
```

Figura 8 – Código-fonte PHP vulnerável à LFI e RFI.
Fonte: do Autor.

Na Figura 8 é ilustrado um exemplo de código-fonte PHP vulnerável ao ataque LFI. Pode ser observado que o código recebe um parâmetro de nome "*arg*" via método GET. O valor é atribuído à variável "*$arquivo*", que, logo após, é incluída no código sem nenhuma verificação. Desse modo, um usuário poderia requisitar uma URL como "*/arquivo.php?arg=/etc/passwd*". Neste caso, o arquivo "*/etc/passwd*" será exibido para o usuário.

5.2.1.3 *Remote File Include*

Esta vulnerabilidade é similar a LFI. A principal diferença reside no fato de incluir um arquivo hospedado em outro servidor *Web* ao invés do sistema local. Isso representa um risco de segurança maior, pois o arquivo remoto pode ter sido intencionalmente criado e preparado pelo atacante, aumentando suas possibilidades de ações maliciosas de forma relativamente simples e prática.

A vulnerabilidade RFI ocorre devido a falta de validação e tratamento correto dos dados de entrada. O código ilustrado na Figura 8 permite a inclusão remota de arquivos. Um exemplo de URL maliciosa poderia ser "*/arquivo.php?arg=http://servidorWebDoAtacante.com.br/comandos.txt*". Este arquivo pode conter qualquer sequência de comandos e construções reconhecidas pela linguagem PHP e pelo sistema hospedeiro da aplicação alvo. Consequentemente, uma vulnerabilidade RFI é crítica e permite ao atacante realizar diferentes tipos de ações no sistema alvo.

5.2.1.4 *Remote Command Execution*

A vulnerabilidade de RCE permite a um atacante executar remotamente comandos no sistema alvo. Num código PHP, através do comando *system()* sem o devido tratamento dos dados inseridos pelo usuário, é possível executar comandos do sistema operacional.

```php
<?php
$user = $_GET['arg'];
system("echo $user >> users_forum.txt");
?>
```

Figura 9 – Código-fonte PHP vulnerável à RCE.
Fonte: do Autor.

A Figura 9 apresenta um exemplo de código-fonte PHP vulnerável. O objetivo do código é inserir usuários no final do arquivo "*users_forum.txt*", como "*/rce.php?arg=user4*". Neste caso, apenas o usuário "*user4*" será acrescentado no arquivo indicado. Porém, um atacante pode ir muito além. A chamada de sistema *system()* invoca um *shell* para a execução do comando. Como consequência, qualquer composição de comandos possível em um *shell*, utilizando o separador ";", pode ser invocada pelo atacante. Este, por exemplo, poderia requisitar a URL "*/rce.php?arg=user4;cat /etc/passwd;*", o que provocaria a inclusão do "*user4*" no arquivo "*users_forum.txt*", bem como a leitura do conteúdo do arquivo "*/etc/passwd*".

O risco de segurança aumenta em situações onde os servidores *Web* são executados em modo *root*. Nesses casos, os sistemas hospedeiros podem ser facilmente comprometidos por um atacante, estendendo as implicações de segurança para além da aplicação *Web* vulnerável.

5.2.1.5 *Sql Injection*

SQL Injection pode ser definida como a inserção de código SQL (*Structured Query Language*) malicioso através de dados de entrada de uma aplicação. Se bem sucedido, esse

tipo de ataque pode obter acesso aos dados do banco de dados, modificar esses dados, eventualmente executar operações de administrador de banco de dados, e, em alguns casos, executar comandos do sistema operacional (OWASP ATTACK CATEGORY, 2013). O tratamento desses dados, por parte de programadores, pode evitar esse tipo de ataque.

```php
<?php
$par1 = $_GET['username'];
$par2 = $_GET['password'];
$query = "SELECT * FROM usuario WHERE username= '$par1'".
                                    "AND password= '$par2'";
?>
```

Figura 10 – Código-fonte PHP vulnerável à SQL Injection.
Fonte: do Autor.

A Figura 10 apresenta um exemplo de código PHP vulnerável a *SQL Injection*. No exemplo, caso a URL de requisição for "*/show.php?username=user&password=12345*", serão apresentadas as informações do usuário "*user*", com senha "*12345*", cadastradas no banco de dados. Entretanto, se a URL apresentar o formato "*/show.php?username=user&password=1' OR '1*", serão listadas informações de todos os usuários presentes no banco de dados da aplicação *Web*. Cabe observar que os parâmetros "*username*" e "*password*" não sofrem nenhum tipo de tratamento e são inseridos diretamente na pesquisa (*query*) SQL. Deste forma, a operação "*1' OR '1*" é sempre verdadeira, retornando os dados de todos os usuários registrados.

5.2.2 Metodologia de teste e resultados

Os testes foram divididos em duas baterias. Na primeira etapa (bateria I) foram realizados testes individuais, ou seja, com as regras, das ferramentas *ModSecurity* e UniscanWAF, especificamente habilitados para cada uma das vulnerabilidades. Em ambientes reais são utilizados um conjunto de regras comumente. Para o caso de testes, é preferível isolar e testar uma regra por vez primeiramente, no intuito de garantir que nenhuma regra influencie no resultado da outra e, consequentemente, no resultado da detecção como um todo. Na segunda etapa (bateria II), foram realizados testes de detecção com todas as regras

habilitadas em ambas as ferramentas. Foram feitas essas duas baterias para que fosse possível avaliar a eficiência de cada regra, e a eficiência geral da ferramenta.

As vulnerabilidades XSS, LFI, RFI, RCE e SQL Injection foram implementadas em PHP. Foram utilizados os exemplos apresentados na Seção 5.2.1. A aplicação com os códigos vulneráveis foi hospedada em uma máquina virtual com sistema operacional *Linux Ubuntu 10.04.4 LTS*, servidor *Web Apache* versão 2.2.14, PHP versão 5.3.2-1ubuntu4.17, *Perl* versão 5.10.1, *ModSecurity Stable* versão 2.2.6 (com as regras de detecção *Core Rules* versão 2.2.5) e o UniscanWAF.

Para permitir a vulnerabilidade RFI, foram ativadas as seguintes variáveis de configuração do PHP: *register_globals*, *allow_url_fopen* e *allow_url_include*. Uma vez que a comunicação e integração entre sistemas é algo necessário e cada vez mais explorada por diferentes organizações, a ativação desses parâmetros é cada vez mais comum (ROCHA, 2012). Em um primeiro momento, realizou-se testes no *ModSecurity*, e em seguida, no UniscanWAF. Os detalhes desse processo são descritos a seguir.

5.2.2.1 *ModSecurity*

Os resultados da primeira bateria de testes estão sumarizados na Tabela 1. A coluna Regra *ModSecurity* refere-se ao arquivo responsável pela regra, a coluna Vulnerabilidade refere-se a vulnerabilidade avaliada e tratada no arquivo e a coluna *Status* indica se a vulnerabilidade foi ou não encontrada. Nota-se que o *ModSecurity* conseguiu detectar apenas duas das cinco vulnerabilidades, indicando que as regras referentes as vulnerabilidades não detectadas encontram-se deficientemente implementadas para os arquivos referenciados pela coluna Regra *ModSecurity* da Tabela 1, ou não implementadas, no caso da vulnerabilidade RCE. A empresa *Trustwave* (TRUSTWAVE, 2013) apresenta regras adicionais que contemplam o ataque RCE, mas que devem ser adquiridas comercialmente.

A segunda bateria de testes foi realizada com todas as regras habilitadas no *ModSecurity*. O resultado dos testes pode ser visualizado na Tabela 2. Pode-se observar que a ferramenta detectou mais vulnerabilidades com todas as regras habilitadas. De fato, o arquivo "*modsecurity_crs_40_generic_attacks.conf*" possibilita a detecção dos ataques de LFI e RFI, anteriormente não detectados pelos arquivos responsáveis exclusivamente por essas regras. Este arquivo concentra várias regras para ataques *Web*, entretanto, na documentação não fica

claro quais tipos de ataques o arquivo em questão contempla.

Tabela 1 – Testes individuais de vulnerabilidades

Regra ModSecurity	Vulnerabilidade	Status
modsecurity_crs_41 _xss_attacks.conf	XSS	✔
modsecurity_crs_46 _slr_et_lfi_attacks.conf	LFI	
modsecurity_crs_46 _slr_et_rfi_attacks.conf	RFI	
inexistente[3]	RCE	
modsecurity_crs_41 _sql_injection_attacks.conf	SQL-Injection	✔

Fonte: do Autor.

Observou-se um comportamento diferente para diferentes entradas de parâmetros na URL para a vulnerabilidade RFI. Quando o formato da URL é similar a "*http://sistemaWeb.com.br/arquivo.php?arg=http://10.1.1.1/index.html*", ou seja, com um endereço IP como parâmetro, o *ModSecurity* identifica a tentativa de exploração da vulnerabilidade. No entanto, quando o formato da URL é similar ao seguinte "*http://sistemaWeb.com.br/arquivo.php?arg=http://www.dominiovalido.br/index.html*", ou seja, sem um endereço IP como parâmetro, mas com uma URL válida, o *ModSecurity* não detecta a tentativa e exploração RFI do atacante. Entramos em contato com o suporte do *ModSecurity* para descobrir o motivo dos resultados, mas não obtivemos resposta até o momento.

5.2.2.2 UniscanWAF

Para o UniscanWAF foi utilizada a mesma metodologia dos testes anteriores (*ModSecurity*). Os resultados obtidos foram os mesmos nas duas baterias de testes e podem ser observados na Tabela 2.

3 Arquivo não encontrado para esta vulnerabilidade.

5.2.2.3 Resultados

A Tabela 2 apresenta o resultado dos testes efetuados com o UniscanWAF e o *ModSecurity* (com todas as regras habilitadas) no cenário proposto. Como pode ser observado, o UniscanWAF conseguiu detectar todas as vulnerabilidades apresentadas, enquanto que o *ModSecurity* não detectou a vulnerabilidade RCE e teve problemas com a vulnerabilidade RFI, conforme descrito anteriormente.

Tabela 2 – Comparação entre o UniscanWAF e o ModSecurity

Ferramenta	**XSS**	**LFI**	**RFI**	**RCE**	**SQL-Injection**
UniscanWAF	✔	✔	✔	✔	✔
ModSecurity	✔	✔	✔ [4]		✔

Fonte: do Autor.

Os resultados apresentados permitem concluir que o UniscanWAF é capaz de detectar em tempo real requisições que visam explorar vulnerabilidades das aplicações *Web*. Além disso, detecta mais vulnerabilidades que ferramentas similares, como é o caso do ModSecurity e possui uma arquitetura modular e flexível.

5.3 Testes de desempenho

Em um WAF e diversos outros sistemas, tão importante quanto a função de detectar ataques, é o desempenho do sistema para realizar essa tarefa, onde podem ser considerados vários aspectos como, utilização de recursos, tempo de resposta, entre outros. Um WAF deve cumprir com a função de detectar ataques às aplicações *Web* de forma rápida e ocupando a menor quantidade de recursos possíveis. Neste sentido, um WAF que tenha a capacidade de detectar diversos ataques, mas consuma muitos recursos da máquina hospedeira torna-se

4 Detectou parcialmente a vulnerabilidade.

inviável, pois é muito custoso. Da mesma forma, um WAF que ocupe poucos recursos mas não consiga detectar os ataques a que se propõe torna-se ineficiente, pois não cumpre sua função básica. Ou seja, um sistema que consiga uma melhor relação entre consumo de recursos e tempo hábil para detecção, surge como solução ideal para implantação. Para tanto, propõe-se avaliar o desempenho da ferramenta em dois cenários diferentes, com distintos propósitos. Os cenários de testes, bem como seus resultados são explicados a seguir.

Os testes foram efetuados utilizando as mesmas configurações descritas na Seção 5.2.2 para o servidor *Web*. Quanto ao *hardware* envolvido, a máquina do servidor *Web* foi virtualizada com uma CPU. A máquina virtual foi executada em um computador com um processador Intel(R) Core(TM) i5-2450M. A quantidade de memória RAM reservada a máquina virtual foi de 2 GB. Para coletar dados de utilização de CPU do servidor *Web* foi utilizado o software *Sar* (SYSTAT, 2011). A máquina responsável por realizar as requisições, simulando clientes, é equipada com um processador AMD Phenom(tm) II X2 B55 com 4 GB de memória RAM. Para realizar as requisições para o servidor *Web* foi utilizada a ferramenta *Httperf* (SYSTAT, 2011).

Foram escolhidas três faixas de valores que representam o número de requisições por segundo enviadas ao servidor *Web*. O número de requisições por segundo (req/s) enviados ao servidor *Web* nos três casos de teste foram escolhidos de forma empírica. Inicialmente idealizou-se os valores de 100 req/s, 200 req/s e 300 req/s. Realizando os experimentos, pode-se notar um comportamento anômalo nos dados coletados para o UniscanWAF. Para um número de 100 req/s, os valores coletados para o tempo de resposta eram maiores do que os valores coletados para 200 req/s. No mesmo cenário, o *ModSecurity* mantinha tempos de resposta proporcionais ao número de requisições. Investigando o problema, notou-se que no momento que o UniscanWAF estava sendo executado na máquina virtual, a máquina virtual solicitava CPU para a máquina física, executava as regras e liberava a CPU. Isso foi percebido ao analisar o gráfico de utilização das CPUs da máquina física.

Em síntese, a máquina virtual recebe a requisição e solicita um processador para a máquina física. A máquina física aloca um processador para a máquina virtual. Com o processador alocado, a máquina virtual aloca o processador para o processo do UniscanWAF, Como o UniscanWAF executa muito rápido para requisições pequenas e libera o processador. A máquina virtual, por sua vez, informa a máquina física para liberar o processador. Como a máquina física é multiprocessada, ela escolhe outro processador quando a próxima requisição é enviada. Esse processo é repetido a a cada nova requisição.

Dessa forma, para comparar as ferramentas, foi testado empiricamente o valor de

requisições por segundo necessárias para que o UniscanWAF utilizasse a totalidade da CPU, tal qual o *ModSecurity*. O número de requisições por segundo encontrados foram 250 req/s, 300 req/s e 350 req/s. Utilizou-se estes valores pois foi o menor número de requisições onde a máquina virtual não desalocava o processador.

Para cada faixa de requisições por segundo (250 req/s, 300 req/s e 350 req/s) foram realizadas a partir da máquina que simula clientes, por meio da ferramenta *Httperf,* 10.000 requisições para o Apache. Esse processo foi repetido 10 vezes para os seguintes casos: Apache com o UniscanWAF, Apache com o ModSecurity e somente o Apache. Uma média do resultado dessas 10 execuções foi efetuada para obter os dados apresentados nas tabelas a seguir. A seguir são descritos os cenários e os resultados de cada um.

5.3.1 Cenário de testes I

No primeiro cenário de testes, o objetivo é avaliar o impacto agregado no processamento da máquina que hospeda o servidor *Web* de testes, ou seja, analisar o quanto de processamento adicional é acrescido pela utilização dos WAFs analisados. Para tanto, foi coletada a porcentagem de uso do processador na presença do UniscanWAF, do ModSecurity e foi comparado ao do servidor *Web* Apache sem nenhum WAF. Os resultados obtidos são ilustrados na Figura 11.

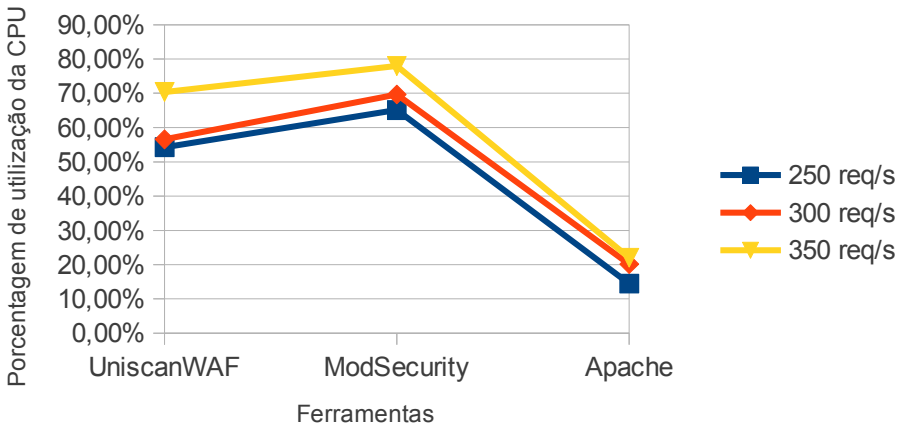

Figura 11 – Porcentagem de utilização da CPU por ferramenta
Fonte: do Autor.

As configurações do ModSecurity e do UniscanWAF utilizadas foram as configurações padrão, ou seja, foi efetuada a instalação de cada ferramenta e nenhuma alteração foi realizada. O resultado do adicional de processamento causado por cada ferramenta pode ser observado na Tabela 3.

Tabela 3 – Processamento adicional causado em relação ao Apache

Ferramenta	**250 *req/s***	**300 *req/s***	**350 *req/s***
UniscanWAF	274,86 %	180,37 %	219,43 %
ModSecurity	349,79 %	244,77 %	253,93 %

Fonte: do Autor.

Como observado na Tabela 3, pode-se notar que existe um acréscimo considerável no processamento comparando-se o ambiente sem a presença de um WAF, somente com o servidor Apache, e um ambiente com o *ModSecurity* e com o *UniscanWAF*. Pode-se concluir que a quantidade de requisições influencia significativamente no processamento, uma vez que a quantidade de operações realizadas por cada uma das ferramentas é baseada em cada requisição recebida. Os dados devem ser interpretados em relação ao Apache na Figura 11, pois a Tabela 3 informa a porcentagem adicional de processamento que cada ferramenta teve.

Para os 250 req/s, 300 req/s e 350 req/s, o UniscanWAF foi a ferramenta que menos adicionou processamento ao hardware avaliado, tendo adicionado em média, 224,9% de processamento adicional. O ModSecurity, no entanto, adicionou cerca de 282,86%. Proporcionalmente, o UniscanWAF possui um ganho de cerca de 20,49% em processamento. Concluindo, o UniscanWAF introduz um processamento menor se comparado ao *ModSecurity* na configuração padrão.

5.3.2 Cenário de testes II

O Cenário de Testes II tem por objetivo avaliar comparativamente o UniscanWAF e o *ModSecurity*. A Figura 11 mostra o gráfico com o resultado coletado para o processamento do *ModSecurity* e do *UniscanWAF*, juntamente com o Apache. Esse resultado foi medido através

da ferramenta *Sar* no momento em que foram enviadas as 10.000 requisições, simulando o cliente, pela ferramenta *Httperf*.

Analisando os resultados, pode-se observar que o UniscanWAF possui um desempenho cerca de 17,41% superior em relação ao *ModSecurity,* considerando-se a média entre as três medições. Entretanto, neste cenário, o *ModSecurity* possui um conjunto maior de vulnerabilidades previstas para detecção, além das cinco analisadas nesse trabalho e implementadas pelo UniscanWAF. Essa diferença confere um custo maior de processamento, pois a cada nova requisição o conjunto de regras a ser verificado é maior. Por esse motivo, um terceiro cenário é proposto onde serão analisadas somente requisições para uma determinada vulnerabilidade.

Deve-se analisar também o tempo de resposta e se todas as requisições foram respondidas. A Tabela 4 mostra o tempo de resposta para os três casos, enquanto que a Figura 12 mostra a quantidade de requisições respondidas.

O tempo de resposta é muito importante para definir se as requisições estão sendo respondidas em tempo hábil. Segundo Pemberton (2007), um tempo de resposta aceitável para um ambiente *Web* está em torno de 1 segundo, sendo 10 segundos um tempo considerado como inaceitável. A Tabela 4 sumariza os tempos de resposta encontrados para as duas ferramentas, bem como para o servidor *Web* sem nenhum WAF.

Tabela 4 – Avaliação do tempo de resposta

Ferramenta	**250 *req/s***	**300 *req/s***	**350 *req/s***
UniscanWAF	186,14 ms	714,58 ms	781,54 ms
ModSecurity	2709,02 ms	2820,4 ms	2852,9 ms
Apache	0,7 ms	0,7 ms	0,7 ms

Fonte: do Autor.

Podemos observar que o tempo de resposta do UniscanWAF é inferior ao do *ModSecurity* para todos os níveis de requisições analisados*,* o que revela que a arquitetura modular do UniscanWAF, juntamente com a implementação *multi thread* possibilita bons resultados. No mesmo sentido, a Figura 12 ilustra o desempenho das ferramentas no que se refere ao número de respostas recebidas e, observa-se que o UniscanWAF, novamente, consegue responder em média 44,29% de requisições a mais do que o *ModSecurity,*

considerando-se a média entre as três medições. Os números apresentados nesta tabela são referentes a quantidade de requisições enviadas de um total de 10.000 requisições.

Os resultados apresentados podem conduzir a uma dedução a respeito do *ModSecurity*: embora o *ModSecurity* possua muitas regras para diversas vulnerabilidades, a ferramenta não consegue um bom desempenho sob altas taxas de requisições. Isso pode ser ocasionado devido as requisições ficarem na fila de espera enquanto uma outra requisição é analisada. Dessa forma, explica-se o fato de o processador ficar com taxas de consumo altas, pois está sendo utilizado constantemente. Isso ocasiona o alto número de respostas não retornadas também. Essa análise possibilita alegar que o ModSecurity necessita de mais recursos de *hardware* em comparação com o UniscanWAF para um cenário nos mesmos moldes do descrito.

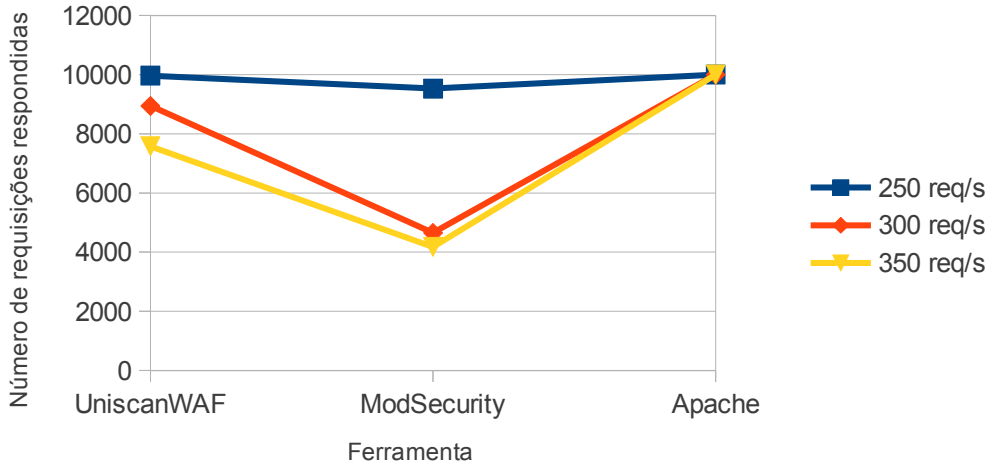

Figura 12 – Quantidade de requisições respondidas de um total de 10.000
Fonte: do Autor.

Para o caso em que deseja-se proteger de ataques de *XSS, LFI, RFI, RCE* e *SQL Injection*, e deseja-se utilizar a configuração padrão da ferramenta, seja por motivos de tempo, ou por não estar familiarizado com a configuração da ferramenta, ou até mesmo por não possuir um *hardware* com grande desempenho, o UniscanWAF é a ferramenta mais indicada entre as duas, pois adiciona o menor nível de processamento e possui o menor tempo de resposta. Além disso, permite a implementação da *whitelist*, já comentada neste trabalho. Ou seja, por hipótese pode-se concluir que o UniscanWAF consegue com a *whitelist*, bloquear um

maior número de vulnerabilidades do que aquelas descritas neste trabalho, desde que a requisição seja bloqueada na *whitelist*, sem a dependência da *blacklist*. Adicionalmente, é a ferramenta que consegue responder ao maior número de requisições, no caso do ambiente avaliado.

5.3.3 Cenário de testes III

No cenário 3 são feitas as mesmas medições do Cenário de Testes II. Entretanto, são efetuadas somente com a vulnerabilidade de LFI habilitada em ambas ferramentas. Cada ferramenta possui um conjunto com 357 regras habilitadas para essa vulnerabilidade. A Figura 13 mostra o consumo de CPU para ambos WAFs.

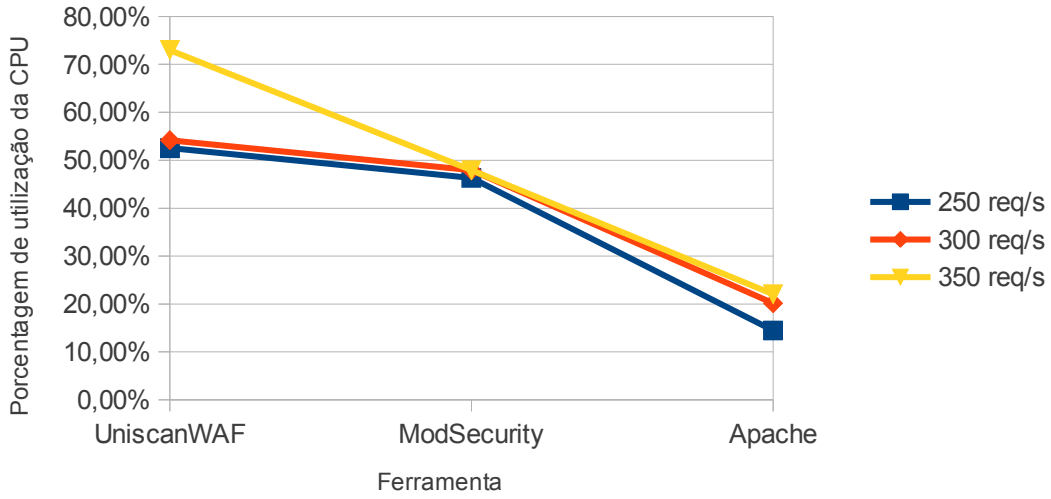

Figura 13 – Porcentagem de utilização da CPU com a vulnerabilidade LFI habilitada
Fonte: do Autor.

Observa-se que quando deixamos os dois WAFs na presença de apenas uma vulnerabilidade, o *ModSecurity* aumenta o desempenho e consome menos recursos. Isso pode ser notado na Tabela 5, em que os tempos de resposta sofrem uma redução significativa. Além disso, por não ter tantas regras para processar (357 regras), o *ModSecurity* conseguiu

responder a todas as requisições que lhe foram efetuadas, como mostrado na Tabela 6.

Tabela 5 – Avaliação do tempo de resposta com a vulnerabilidade LFI habilitada

Ferramenta	**250 *req/s***	**300 *req/s***	**350 *req/s***
UniscanWAF	48,1 ms	73,3 ms	538,8 ms
ModSecurity	1,6 ms	1,5 ms	1,5 ms

Fonte: do Autor.

Analisando o consumo de CPU, tempo de resposta e requisições respondidas, em uma comparação mais igualitária, onde sabe-se exatamente o que cada ferramenta possui habilitada (neste caso, somente a detecção da vulnerabilidade de LFI), nota-se a grande diferença de desempenho em todos os quesitos avaliados. Observa-se claramente a redução do consumo de CPU em relação ao cenário anterior. Mas sem dúvida, o dado que mais impacta é a redução do tempo de resposta. Neste cenário, com apenas uma vulnerabilidade habilitada, o ModSecurity fica com um desempenho 56,25% superior ao tempo de resposta do Apache executando sozinho. Entretanto, o tempo de resposta ainda é baixo.

Tabela 6 – Quantidade de requisições respondidas com a vulnerabilidade LFI habilitada

Ferramenta	**250 *req/s***	**300 *req/s***	**350 *req/s***
UniscanWAF	10000	10000	8686,3
ModSecurity	10000	10000	10000

Fonte: do Autor.

Portanto, para este último cenário, o desempenho do ModSecurity se mostrou superior ao do UniscanWAF. Isso indica também, que a configuração padrão do ModSecurity muitas vezes pode ser demasiadamente pesada para alguns *hardwares*.

5.3.4 Discussão sobre os resultados

É importante ressaltar que as duas soluções propostas obtiveram tempos de resposta coerentes com utilizações em ambientes com um número de requisições por segundo razoável. Nenhuma das ferramentas chegou a um tempo de resposta considerado inaceitável por Pamberton (2007). O tempo mais longo foi de pouco mais de 2 segundos para o ModSecurity, valor bem abaixo do dito como inaceitável. Outro ponto a ser levado em consideração é que os testes foram feitos sob fortes cargas de requisições para o *hardware* em questão. No ambiente utilizado para testes aconteceram alguns imprevistos, que foram relatados anteriormente. Futuramente, poderão ser realizados os mesmos testes em um ambiente simulado para evitar distorções percebidas no ambiente relatado neste trabalho. Seria interessante a avaliação de valores maiores de requisições, como por exemplo 1000 req/s e 2000 req/s, para avaliar se o comportamento do UniscanWAF permanece linear. Porém, também de forma empírica, notou-se que a máquina responsável por enviar as requisições descartava pacotes no momento do envio com valores muito elevados, no caso superiores a 370 requisições. Dessa forma, as requisições não chegavam ao destino, e não podiam ser avaliadas pela ferramenta *Sar*.

No quesito desempenho constatou-se que o *ModSecurity* possui melhor desempenho do que o UniscanWAF em um cenário onde as duas ferramentas possuem as mesmas configurações (Cenário de Testes III). O que deve ficar claro é que o ModSecurity é mais rápido que o UniscanWAF, mas não quer dizer que detecta mais vulnerabilidades. Para o conjunto de vulnerabilidades estudados neste trabalho, o oposto pode ser percebido.

Resumindo, para o caso em que deseja-se proteger de ataques de *XSS, LFI, RFI, RCE* e *SQL Injection*, e não ajustar a configuração da ferramenta o UniscanWAF é a ferramenta mais indicada. Quando desempenho for um fator determinante, e for necessário proteger-se contra mais tipos de ataque (sem criar novos detectores, o que é disponível nas duas ferramentas) o *ModSecurity* é a alternativa mais viável.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Este trabalho apresentou um *Web Application Firewall* denominado UniscanWAF, cuja principal função é detectar e interceptar em tempo real a exploração de vulnerabilidades, por parte dos atacantes, endereçadas à aplicações *Web*. O UniscanWAF possui mecanismos híbridos de detecção de vulnerabilidades, implementados através de *whitelist* e *blacklist*. Isso possibilita a proteção contra novos ataques, através da *whitelist*, e contra ataques conhecidos através da *blacklist*.

Como visto, uma das principais causas das vulnerabilidades em aplicações *Web* é o não tratamento dos dados informados pelo usuário. Diferentemente dos WAFs existentes atualmente, o UniscanWAF foi projetado para ser integrado ao SDLC de aplicações *Web*, para isso, aplicando o conceito de *whitelist*. A proposta atua incorporando segurança ao SDLC, mais especificamente em duas fases. Na fase de Codificação é realizado o preenchimento da *whitelist* e na fase de Manutenção, onde a *whitelist* é utilizada para bloquear acessos com dados não previstos pela equipe de desenvolvimento. Além disso, a ferramenta pode ser incorporada à SDLC tradicionais, uma vez que não influencia na fase de projeto das aplicações, apenas na fase de desenvolvimento. Vale ressaltar que o funcionamento da *blacklist* não tem qualquer relação às etapas de desenvolvimento das aplicações, diferentemente da *whitelist*.

Para a fase de execução da aplicação *Web* foi incorporada ainda a funcionalidade de *blacklist,* que objetiva a análise das requisições HTTP em busca de uma dessas cinco vulnerabilidades descritas: XSS, SQL Injection, RFI, LFI e RCE. A *blacklist*, também chamado de Módulo de Detecção de Vulnerabilidades, foi comparada ao ModSecurity, e demonstrou-se que o UniscanWAF é capaz de detectar mais vulnerabilidades dentro do conjunto analisado. Com a implementação da *whitelist* juntamente com a *blacklist*, além de adicionar maior segurança à aplicação *Web*, é possível trabalhar com filosofias distintas de segurança, através da alteração das sequencias de execução de cada lista. Em outras palavras, o gerente pode desejar, antes de analisar a *whitelist*, saber se a requisição é um ataque, para só depois analisar a requisição através de parâmetros de implementação. Também pode analisar a *whitelist* antes, independente de querer saber se é um ataque, pois caso a requisição não contemple todos os requisitos da *whitelist*, será automaticamente descartada, sendo ou não um ataque. É uma diferença sutil, mas que implica na realização de uma análise *offline* dos dados

capturados e registrados em *log*.

No referencial bibliográfico pesquisado não identificou-se propostas que utilizassem uma abordagem em *whitelist* como apresentado neste trabalho, de forma que seja possível definir, baseado na especificação do *software*, quais dados (parâmetros e valores de parâmetros) são permitidos para cada recurso. A utilização desse modelo visa forçar as equipes de desenvolvimento validarem os dados que trafegam na aplicação, sob pena de não terem a funcionalidade desenvolvida disponível caso não o façam. Tal funcionalidade foi comprovada no Capítulo 5, garantindo que somente requisições, parâmetros e valores previstos pelos desenvolvedores sejam aceitos.

Outro ponto forte da ferramenta é a arquitetura modular empregada. Como os detectores de vulnerabilidade são acoplados ao Módulo de Detecção de Vulnerabilidade, caso um usuário do UniscanWAF queira desenvolver uma regra para uma vulnerabilidade não prevista na ferramenta, não é preciso refatorar o núcleo do sistema. Entretanto, ressalta-se que esta característica da ferramenta é resultado da versão do Uniscan, não sendo fruto do presente trabalho. Complementarmente, o UniscanWAF foi concebido como uma extensão do Uniscan, dentro dos conceitos de flexibilidade, simplicidade e extensibilidade desta ferramenta, um *scanner* de vulnerabilidades de sistemas Web.

Testes de desempenho foram realizados como forma de analisar o comportamento da ferramentaUniscanWAF em um ambiente com grande número de requisições, e foi comparado ao *ModSecurity*. O UniscanWAF mostrou que pode ser utilizado em ambientes com grande número de requisições, ficando dentro da média de 1 segundo de tempo de resposta para a maioria dos casos. Entretanto, comprovou-se que o ModSecurity possui um desempenho bastante superior ao UniscanWAF em um cenário de igualdade de configurações.

O teste de desempenho foi de grande importância para delimitar os rumos que o desenvolvimento da ferramenta deve tomar. Os trabalhos futuros serão focados em aumentar a performance da ferramenta, que mostrou-se capaz de detectar vulnerabilidades de igual para igual com uma solução amplamente utilizada, dentro do conjunto de vulnerabilidades propostos. Paralelamente, novos detectores de vulnerabilidades serão incorporados para aumentar a abrangência da ferramenta.

Adicionalmente, pretende-se, como trabalho futuro, realizar os mesmos testes em um ambiente simulado com o objetivo de evitar qualquer distorção como percebido no ambiente utilizado pelo trabalho.

Ajustados esses quesitos, pretende-se modificar o UniscanWAF como um *Firewall* de Aplicação *Web* Distribuído. Para isso, seria utilizada uma *whitelist* global em que diversos

WAFs compartilhariam a sua *whitelist* global. Nessa sentido, para que uma requisição fosse permitida, ela deveria receber a aprovação, total ou parcial, dos WAFs pertencentes à federação de WAFs.

REFERÊNCIAS

ANTUNES, N.; VIEIRA, M. **Defending against Web Application Vulnerabilities.** Computer , v. 45, n.2, p.66-72, 2012.

CERT.br. **Estatísticas dos Incidentes Reportados ao CERT.br**. Disponível em: <http://www.cert.br/stats/incidentes/>. Acesso em: 5 out. 2012.

CHESS, B.; MCGRAW, G. Static analysis for security. IEEE Security Privacy , v. 2, n. 6, p. 76–79, 2004.

FONSECA, J.; VIEIRA, M.; MADEIRA, H. **Vulnerability & attack injection for web applications.** International Conference on Dependable Systems & Networks, v., p. 93-102, 2009.

IPA (INFORMATION-TECHNOLOGY PROMOTION AGENCY, JAPAN). **Web Application Firewall (WAF)**: A Handbook to Understand Web Application Firewall. Japão, 2011.

MCGRAW, G. **Software Security**. IEEE Security & Privacy, v. 2, n. 2, p. 80-83, 2004.

NAKAMURA, E. T.; GEUS, P. L de. **Segurança de redes em ambientes cooperativos**. São Paulo: Novatec Editora, 2007.

OWASP TOP 10. In: THE OPEN web application security project. Disponível em: <https://www.owasp.org/index.php/Top_10_2010-A2>. Acesso em: 02 jan. 2013.

OWASP ATTACK CATEGORY. In: THE OPEN web application security project. Disponível em: <https://www.owasp.org/index.php/SQL_Injection >. Acesso em: 02 jan. 2013.

PEMBERTON, S. **Open Standards Rich Internet Applications**. 2007. Palestra realizada em Berlin em 26 de nov. 2007. Disponível em: <http://www.w3.org/2007/Talks/09-26-steven-berlin/>. Acesso em: 15 jan. 2013.

PEREIRA , P. R. **Validação Empírica de Modelos de Desempenho de Servidores Web .** 2011. 60 f. Monografia (Bacharelado em Ciência da Computação) – Universidade do Estado de Santa Catarina, Joinville, 2011.

PRASAD, A. V. K. and RAMAKRISHNA, S. **Mining for Web Engineering.** International Journal of Computer Trends and Technology (IJCTT), p. 151–156, 2011.

ROCHA, D. **Uniscan**: Um Scanner De Vulnerabilidades para Sistemas Web . 2012. 95 f. Monografia (Bacharelado em Ciência da Computação) – Universidade Federal do Pampa, Alegrete, 2012.

ROCHA, D. UNISCAN - um scanner devulnerabilidades para sistemas web. Disponível em: <http://uniscan.sourceforge.net/>. Acesso em: 01 Jan. 2013.

ROCHA, D.; KREUTZ, D.; TURCHETTI, R. **Uma ferramentalivre e extensível para detecção de vulnerabilidades emsistemas web.** 7a Conferencia Ibérica de Sistemas y Tecnologías de Información, v., p. 747-752, 2012.

SYSTAT. **Sysstat**. França, 2011. Versão 10.0.3. Sistema operacional Linux e páginas de manuais.

TEODORO, N.; SERRAO, C. **Web application security**: Improving critical web-based applications quality through in-depth security analysis. International Conference on Information Society (i-Society), p. 457–462, 2011.

TRUSTWAVE. Trustwave. Disponível em: <https://www.trustwave.com/ >. Acesso em: 01 Jan. 2013.

XIONG, P.; PEYTON, L. **A Model-Driven Penetration Test Framework for Web Applications .** Eighth Annual International Conference on Privacy, Security and Trust , p. 173–180, 2010.