Uma Ferramenta Adaptativa Baseada em Agentes Móveis Inteligentes para Segurança de Aplicações Web

Márcio A. Macêdo[1], Ruy J. G. B. de Queiroz[1], Julio C. Damasceno[2]

[1]Centro de Informática– Universidade Federal de Pernambuco (UFPE)
Cidade Universitária – 50.740-540 – Recife – PE – Brasil

[2]Departamento de Informática– Universidade Federal Rural de Pernambuco (UFRPE)
Rua Dom Manuel de Medeiros, S/N – 52.171-900 – Recife – PE – Brasil

{mam5,ruy}@cin.ufpe.br, julio.damasceno@deinfo.ufpe.br

Abstract. *Nowadays, we see the need for migration of desktop software to the Internet. Because of this, the number of web applications developed grows every day, and projections related to the persistence of such a steady growth are even more optimistic. However, the security of web applications is not prioritized the same way that the quest for rapid and agile software development. Based on these considerations, this paper presents an alternative to solve the aforementioned problem by using an adaptive web application firewall that also protects web applications from known attacks, and protect them from possible 0-day attacks that may arise.*

Resumo. *Nos dias atuais, observamos a necessidade de migração dos softwares desktop para a Internet. Por conta disso, cresce a cada dia o número de aplicações web desenvolvidas e as projeções relacionadas a continuidade desse crescimento são ainda mais otimistas. No entanto, a segurança das aplicações web não é priorizada da mesma forma que a busca pelo desenvolvimento de software rápido e ágil. Com base em tais considerações, esse trabalho apresenta uma alternativa para solucionar o problema supracitado através da utilização de um firewall de aplicações web adaptativo que, além de proteger as aplicações web de ataques já conhecidos, protegerá as mesmas de possíveis ataques 0-day que possam surgir.*

1. Introdução

A proteção de servidores de aplicações *web*, permitindo acesso às aplicações somente por pessoas autorizadas, é uma necessidade global. Isso se deve ao crescimento do desenvolvimento de aplicações *web*, pois programadores, empresas e corporações têm observado que a Internet é atualmente um dos meios mais importantes para se fazer comércio e isso provocou o desaparecimento de fronteiras para os negócios. Contudo, devido ao crescimento exponencial desse mercado, empresas têm contratado mão de obra inexperiente, e têm recorrido a soluções de desenvolvimento fornecidas por diversos *frameworks*, que muitas vezes escondem vários problemas de segurança.

Um dos grandes problemas que estas empresas vêm enfrentando é o crescimento dos ataques que exploram vulnerabilidades presentes em aplicações *web*. Essas vulnerabilidades em questão são criadas durante o processo de desenvolvimento e colocam as aplicações e consequentemente as empresas em risco, podendo levar ao

vazamento de informações privilegiadas, o que pode acarretar no comprometimento do modelo de negócio.

Apesar das dificuldades no tratamento de falhas de aplicações, devido ao surgimento de novas vulnerabilidades e da diversificação das técnicas de ataques a camada de aplicação [CERT.BR 2010], algumas empresas têm adotado possíveis soluções para esse problema como: o investimento em treinamento de desenvolvimento de código seguro para suas equipes, a utilização de um processo de software confiável e baseado em testes de segurança e a utilização de IDS (*Intrusion Detection Systems*) [Bace and Mell 2001]. Porém, nenhuma dessas medidas ataca o problema de forma eficaz ou pelo menos dá ao desenvolvedor a garantia de proteção da sua aplicação.

A solução apresentada neste artigo é implementada através de um *firewall* de aplicações que trabalha diretamente na camada de aplicação. Essa solução, além de funcionar como ponto único de verificação através de um *proxy* reverso, filtra os possíveis pacotes que possam apresentar ameaças a aplicação e cria uma camada de filtragem de forma segura e confiável. Além disso, a nossa solução conta com o auxilio de um agente móvel inteligente, que aprende sobre novas formas de ataque e atualiza a base de conhecimento do *firewall* de aplicações em tempo real de forma totalmente autônoma, fornecendo a nossa solução características adaptativas.

2. Conceitos Preliminares

2.1. *Web Application Firewall*

De acordo com [WAFEC 2006], *Web Application Firewall* (WAF) é uma nova tecnologia de segurança que tem o papel de proteger aplicações *web* de ataques. As soluções de um WAF são capazes de prevenir ou até mesmo neutralizar um ataque a uma aplicação, conseguindo filtrar de forma eficiente o que um IDS (Intrusion Detection Systems) e *firewalls* de rede não conseguem. Isso se deve ao fato de um WAF agir diretamente na camada de aplicação, filtrando os dados e principalmente parâmetros utilizados em uma transação HTTP [Jones and Bejtlich 2006].

2.1.1. Implementações e configurações de um WAF

Os WAFs podem ser implementados de três formas diferentes, cada uma possuindo vantagens e desvantagens. Segundo [WAFEC 2006], as formas de implementação de um WAF são:

- No nível da camada de rede.
- Através de um *proxy* reverso.
- Diretamente no servidor *web*.

De acordo com [WAFEC 2006], os *firewalls* de aplicações *web* podem ser configurados para detectar ataques de duas diferentes formas:

- Modelo de segurança negativo.
- Modelo de segurança positivo.

2.1.2. Modelo de segurança negativo

O modelo de segurança negativo é simples de ser configurado e tem por base permitir o tráfego de todos os pacotes de solicitação, filtrando somente os que obedecem a alguma assinatura ou regra do WAF (*black list*). O sucesso dessa implementação é determinado pela eficiência com que o *firewall* de aplicação *web* consegue detectar solicitações nocivas, de acordo com sua base de regras e assinaturas.

O problema em implementar o modelo de segurança negativo, reside no risco do banco de dados de regras filtrar um grande número de falso-positivos (quando a regra filtra pacotes autênticos da aplicação) e por esse modelo ser mais suscetível a técnicas de evasão.

A vantagem de se usar o modelo de segurança negativo se deve a facilidade de configuração que o mesmo proporciona, pois serão criadas regras e assinaturas de ataques que serão aplicadas a todas as requisições.

2.1.3. Modelo de segurança positivo

O modelo de segurança positivo é complexo considerando sua configuração e implementação. Por padrão todo e qualquer tráfego é bloqueado e permite-se somente os pacotes de solicitação que respeitem a algumas regras que garantem que a solicitação é segura para a aplicação (*white list*). Essa forma de configuração é mais segura e eficiente, pois necessita de menos regras de segurança para filtragem.

O problema em configurar um *firewall* de aplicação com esse modelo se deve a grande necessidade de conhecimento sobre a aplicação e, a partir desse conhecimento, julgar o que é nocivo ou não e assim, criar uma regra totalmente personalizada para proteger a aplicação de ataques.

2.2. *Honeypots*

Honeypots são recursos computacionais estreitamente monitorados, tais como softwares, hosts servidores ou máquinas virtuais instalados em um ambiente computacional, cuja intenção é que os mesmos sejam sondados, atacados e comprometidos. Um *honeypot* funciona como uma armadilha para que seja possível aprender sobre o padrão de ataque e ferramentas empregados por invasores durante o processo de invasão. O *honeypot* não reage aos ataques, apenas armazena informações sobre o processo de ataque, desde a sondagem até a invasão, inclusive monitorando as ações do invasor depois que o mesmo invade o sistema [Provos and Holz 2007].

Dessa forma, será possível aprender mais sobre os mecanismos de ataques e usar as informações obtidas para proteger as aplicações web do ambiente. Para o invasor, o sistema parecerá comprometido e, portanto, poderá realizar várias tentativas de obtenção de informações de bancos de dados e/ou mesmo testar falhas na aplicação. Do ponto de vista do modelo de segurança sugerido aqui, varias informações importantes estão sendo coletadas a respeito do mecanismo de ataque, do comportamento do invasor, das ferramentas e que informações exatamente o invasor busca.

Para entender melhor os *honeypots*, é necessário destacar os seus tipos e particularidades, sendo assim, para uma primeira distinção vamos esclarecer que os *honeypots* podem ser hosts físicos completos, ou seja, sistemas operacionais instalados em uma máquina física com diversos serviços ativos aguardando conexões entrantes.

Por outro lado podemos implementar *honeypots* através de máquinas virtuais rodando sobre algum sistema de virtualização, onde os *honeypots* apresentam as mesmas funcionalidades implementadas em uma máquina física, no entanto, a escalabilidade desse modelo é muito maior, sua manutenção é facilitada e, ainda, o custo é bem menor do que a utilização de máquinas físicas.

Outra importante distinção dos *honeypots* é quanto ao grau de interação entre os mesmos e os invasores. Essa interação pode ser uma interação limitada ou uma interação mais completa em termos de possibilidades para os potenciais atacantes. De acordo com essa distinção, temos os *honeypots* de baixa-interação e os de alta-interação, respectivamente.

Os *honeypots* de baixa-interação são sistemas que emulam serviços de rede, aplicações e alguns aspectos limitados de um servidor real. O seu objetivo, ao implementar um subconjunto de funcionalidades de um sistema completo, é coletar informações sobre um comportamento particular de ataque, por exemplo, como o acesso a um determinado arquivo pode ser obtido, que ferramentas foram utilizadas e se algum *worm* foi instalado. As informações obtidas através do monitoramento dos ataques são muito mais quantitativas que qualitativas, pois o atacante tem uma interação limitada devido aos poucos serviços e funcionalidades de rede disponíveis. A vantagem da sua utilização é a sua facilidade de implementação e de manutenção [Provos and Holz 2007].

Por outro lado, um *honeypot* de alta-interação é um sistema de computação convencional, assim como um servidor de comércio eletrônico, um roteador, ou até mesmo um switch. O importante é que esse sistema não gera tráfego para a rede, apenas carga local devido aos processos dos serviços que rodam localmente. Esse tipo de honeypot é um sistema completo, mas altamente vulnerável a ações maliciosas. Essa consideração é importante para a detecção de ataques, pois cada interação com esse tipo de honeypot é suspeita. Portanto, todo o tráfego de e para o *honeypot* é monitorado e contabilizado em arquivos de log.

2.3. Agentes Móveis

Os agentes móveis tem se tornado uma tecnologia emergente para o desenvolvimento de aplicações e sistemas em ambientes abertos, distribuídos e heterogêneos, como, por exemplo, a Internet [Wangham 2004]. O termo agente é usado em diversas disciplinas da ciência da computação, sendo um conceito estudado e aprofundado em inteligência artificial, sistemas distribuídos, sistema de aprendizado adaptativo, sistema especialista, algoritmos genéticos e em estudos sobre ambientes sociais on-line e colaborativos.

Apesar de o paradigma de agentes móveis se apresentar como uma tecnologia nova, o interesse nos agentes móveis não está motivado somente na tecnologia, mas também pelos benefícios que eles fornecem para a criação de sistemas distribuídos. Lange (1998) apresenta sete boas razões para se utilizar agentes móveis, são elas:

- Redução da Carga da Rede;
- Diminuição da Latência da Rede;
- Encapsulamento de Protocolos;
- Execução Assíncrona e Autônoma;

- Adaptação Dinâmica;
- Naturalmente Heterogêneos;
- Robustos e Tolerantes a Falhas.

3. Trabalhos Relacionados

3.1. *ModSecurity*

O *ModSecurity* [ModSecurity 2010] é o WAF mais conhecido e utilizado na Internet, sendo um projeto *Open Source* que é mantido pela *Breach Security*. "O *ModSecurity* é um *firewall* de aplicações *web* que funciona interligado ao container *Apache* como um módulo acoplado, e trabalha de duas formas, diretamente no servidor ou como um *proxy* reverso. O mesmo fornece proteção contra uma série de ataques a aplicações *web* e permite a monitoração do tráfego HTTP, registro e análise em tempo real" [ModSecurity 2010].

O *ModSecurity* geralmente usa o modelo de segurança negativo e, por ser o mais conhecido e também o primeiro *firewall* de aplicações *Open Source*, o mesmo possui vários outros projetos relacionados, também *Open Source*, que ajudam a melhorar essa solução.

4. JShield *Security Appliance*

O jShield *Security Appliance* é a nossa solução para segurança de aplicações web, que tem por finalidade impedir que atacantes comprometam a segurança desse nicho de aplicações. Essa solução é composta por três ferramentas, que são:

- jShield *Web Application Firewall;*
- Loki *Honeypot*;
- Savant *Mobile Agent*;

A seguir, apresentaremos as três ferramentas separadamente, observando todas as características das mesmas. Após isso, será abordado a arquitetura completa do jShield *Security Appliance* definindo o funcionamento das 3 ferramentas em conjunto e sua comparação com trabalhos relacionados.

4.1. JShield *Web Application Firewall*

A nossa solução de WAF denominada jShield [Macêdo *et al.* 2010] é uma ferramenta desenvolvida na linguagem de programação Java. Um dos motivos para escolha desta linguagem foi a camada de abstração que Java oferece ao gerenciamento do protocolo HTTP, além da necessidade de um *proxy* reverso robusto e estável para os *containers* que rodam aplicações desenvolvidas nessa linguagem como, por exemplo, o Tomcat da *Apache Software Foundation* [Macêdo *et al.* 2010].

O jShield funciona através de um *proxy* reverso, podendo utilizar tanto o modelo de filtragem negativo baseado em regras de *black list* (configuração padrão), como também o modelo de filtragem positivo utilizando regras específicas para cada tipo de requisição, baseando-se em regras de *white list* (deve ser configurado). O jShield trabalha com um ponto único de verificação, que tem como função receber todas as

solicitações HTTP direcionadas aos servidores *web* e filtrar pacotes potencialmente nocivos às aplicações, redirecionando esse tráfego de pacotes suspeitos para os *honeypots* a fim de documentar o ataque.

Além disso, o jShield também faz a verificação dos dados enviados da aplicação para o cliente, onde o mesmo filtra os dados da transação para evitar que a "aplicação ataque o usuário". Um exemplo que podemos citar são os ataques *cross-site scripting* persistidos, onde o atacante insere o código malicioso no banco de dados, por exemplo.

Outra característica interessante do jShield é que o mesmo implementa um algoritmo que ativa subclasses de regras referente as grandes categorias de ataques baseadas no top 10 de ataques a aplicações web [OWASP 2010], aumentando assim a capacidade da ferramenta em detectar e prevenir ataques nessas categorias. A Figura 1 retrata o funcionamento do jShield.

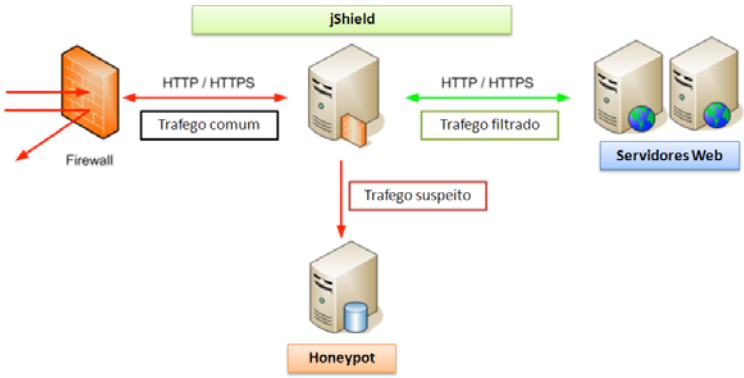

Figura 1. Esquema de gerenciamento de tráfego e filtragem do jShield.

Primeiramente, o *firewall* de rede recebe todas as solicitações HTTP provenientes da internet que foram endereçadas para os servidores *web*, e redireciona essas solicitações para o jShield.

Após receber os pacotes, o jShield normaliza os dados da transação a fim de evitar tentativas de evasão, ao mesmo tempo que verifica os parâmetros e métodos da solicitação HTTP em busca de assinaturas ou padrões de ataques.

Depois de receber e analisar cada pacote do tráfego comum repassado pelo *firewall* de rede, o jShield decide se o pacote é nocivo ou não para a aplicação *web*. Caso a solicitação não seja nociva, o jShield encaminha os pacotes para os servidores web de forma totalmente transparente para a aplicação.

Caso a solicitação seja nociva, por padrão, o jShield descarta o pacote sem encaminhar a solicitação do atacante para nenhum local abortando a conexão. Outras possibilidades seriam: redirecionar a requisição para uma página de acesso negado; encaminhar a requisição para um aplicativo *honeypot*, que poderá receber e documentar toda a tentativa de ataque para, por exemplo, analisar e gerar novas regras e assinaturas de ataques com base nas inferências realizadas nesses *logs*.

4.2. Loki *Honeypot*

O Loki é um *honeypot* de baixa iteratividade que armazena somente situações julgadas por ele como suspeita. Esta ferramenta irá capturar o tráfego através do *proxy* reverso,

que estará recebendo as transações direcionadas a uma aplicação vulnerável. Esse método de captura analisa os pacotes de informação que serão direcionados às aplicações *web*, e então, através de regras para detecção de anomalias que serão configuradas para cada página da aplicação vulnerável de forma específica, o Loki irá documentar todas as tentativas de intrusão a essa aplicação, o que auxiliará o Savant a detectar novos padrões de ataques.

As regras específicas, para cada página da aplicação que o Loki usará para detectar anomalias, serão escritas em XML obedecendo a uma representação de dados muito simples, que facilitará tanto a configuração e programação de novas regras, quanto a leitura das mesmas pelo *honeypot*.

Inicialmente, quando o *proxy* reverso receber a transação direcionada a aplicação vulnerável, irá solicitar ao Loki para que o mesmo carregue as regras específicas para página da aplicação especificada na transação.

Após isso, o Loki receberá do *proxy* reverso o conteúdo da transação para analisar se o mesmo respeita ou não as regras definidas para essa página. A Figura 2 descreve esse processo.

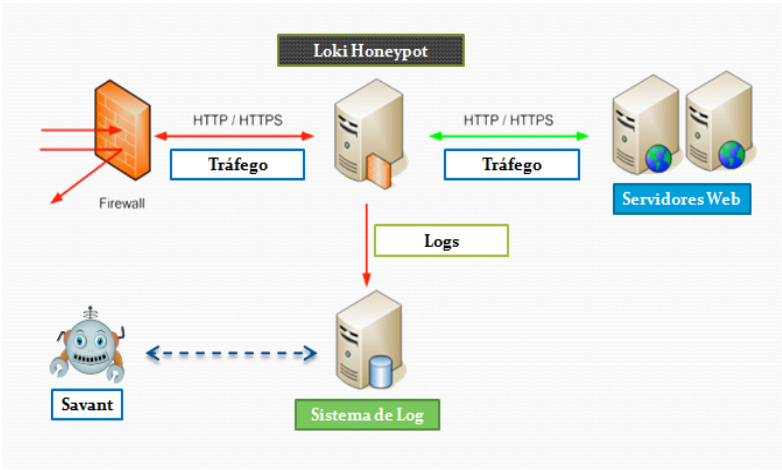

Figura 2. Funcionamento do Loki *Honeypot*.

Caso a transação não respeite as regras, o Loki irá considerar a transação como uma possível tentativa de ataque e irá documentar todos os detalhes da mesma. Para fazer essa documentação será criado um documento XML que servirá como repositório de *logs*, onde serão armazenadas todas as informações inerentes aos ataques detectados pelo Loki Honeypot.

Utilizando essa estratégia, conseguiremos, perfeitamente, detectar as tentativas de ataques que serão feitas às aplicações *web* vulneráveis utilizadas como *honeypots*, além de representar esses dados da melhor forma possível, para que o Savant possa extrair informações e, com isso, gerar novas regras de produção para o jShield.

4.3. Savant *Mobile Agent*

Savant é um agente móvel implementado na linguagem de programação Java, utilizando o *framework* de desenvolvimento para agentes móveis chamado de Aglets [IBM 1996]. Aglets etimologicamente é a união das palavras *applets* e *agent* e foi desenvolvido nos laboratórios da IBM do Japão. Aglets são agentes móveis baseados na linguagem de

programação Java, em outras palavras, são objetos móveis que podem visitar nós de uma rede de computadores.

Uma vez executando em um computador, um Aglet pode parar sua execução repentinamente ou pode ser despachado para outro computador e retomar sua execução neste novo computador, pois o Aglet leva consigo seu código e seu estado. Os Aglets são executados dentro de um contexto-aglet, o qual é o espaço de trabalho dos Aglets e tem como responsabilidade gerenciar os agentes em execução. Em 7 de julho de 2000 a IBM disponibilizou o código fonte do sistema sob licença pública, aprovada como uma iniciativa de código aberto.

Esse *framework* nos proporcionou várias facilidades como, por exemplo, o fornecimento de um contexto (ambiente para funcionamento do agente), protocolo de comunicação/transferência de agentes (ATP – *Agent Transfer Protocol*) e as funcionalidades necessárias para a vida desse agente como criação, migração, reobtenção (ação de chamar o agente móvel de volta ao nó origem) e eliminação.

O Savant *Mobile Agent* tem por função principal navegar por contextos que contenham o Loki *Honeypot*, com finalidade de analisar os *logs* gerados por essa ferramenta, gerar regras de assinaturas dos ataques detectados, e transportar as regras úteis geradas de forma autônoma para o jShield, onde essas regras serão adicionadas a sua base de conhecimento. A Figura 3 descreve o funcionamento do Savant.

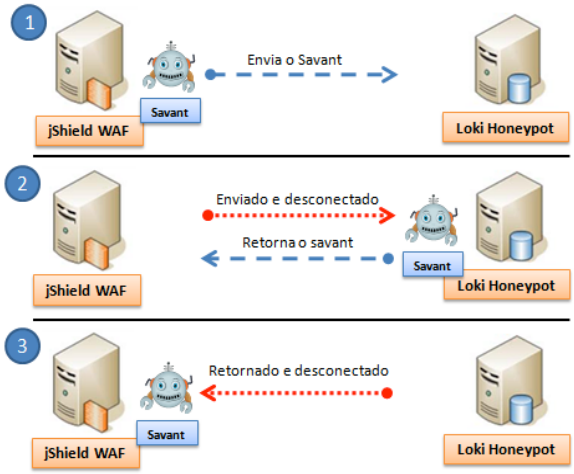

Figura 3. Arquitetura de funcionamento do Savant *Mobile Agent*.

No primeiro estágio, o Savant é transferido para o contexto que está localizado no servidor do Loki *Honeypot*, nesse instante o objeto do agente é serializado e enviado pela rede, e é totalmente desconectado do contexto origem. No segundo estágio, o agente é reconstruído no contexto destino, onde executará sua principal função, que é dividida em 3 etapas: analisar os logs do Loki *Honeypot*, minerá-los a fim de gerar novas regras para o jShield, baseando-se nesses *logs* e adicionar essas regras em sua lista de regras. Após terminar esse processo, o objeto do agente é novamente serializado e devolvido para o contexto de origem. Quando isso acontece, o Savant é totalmente desconectado do contexto em que se encontra. No terceiro estágio, o agente é reconstruído mais uma vez, e executará a ação de atualizar a base de dados do jShield com as novas regras, de forma totalmente autônoma.

Para classificar as tentativas de ataques, o Savant faz uso de técnicas de sensibilidade ao contexto baseado em um conhecimento pré-estabelecido sobre tipos de ataques existentes. Dessa forma, a ferramenta poderá classificar as possíveis tentativas de invasão detectadas, o que também o ajudará a minerar o conhecimento existente nos *logs*.

Na etapa de mineração, primeiramente, é analisado o conteúdo dos parâmetros de uma solicitação registrada como tentativa de ataque pelo Loki, a ferramenta irá executar 2 passos: normalizar todos os dados (técnicas anti-evasão) e analisar se algum parâmetro da solicitação possui palavras reservadas baseando-se em uma ontologia de possíveis padrões de ataque. Essa ontologia mapeia palavras reservadas as suas respectivas classes de ataques.

Caso o parâmetro possua palavras reservadas, o Savant irá considerar a solicitação como uma possível tentativa de ataque. Após isso, a solicitação passa por uma nova análise, por meio da qual será verificado se as regras atuais do jShield conseguem filtrar a possível tentativa de invasão. Esse teste é verificado utilizando a lista de regras que o Savant conhece.

Caso o teste dessa etapa seja considerado negativo, a ferramenta irá classificar a tentativa de ataque baseando-se nas palavras reservadas detectadas durante a primeira etapa. Em seguida, será gerada uma nova regra de filtragem para o modelo negativo do jShield, utilizando os dados da solicitação interceptada pelo Loki. Para gerar essa regra, utilizaremos as palavras reservadas existentes nesse *log* e *tokens* que representem operações lógicas. No próximo passo, a ferramenta adiciona a nova regra em sua lista de regras conhecidas, onde aguardará o agente móvel ser transportado de volta ao contexto onde se encontra o jShield, para então atualizar a base de regras do mesmo.

5. Teste de Desempenho: JShield vs ModSecurity

Para validar a nossa pesquisa, foram realizados vários testes de carga e estresse no jShield e no ModSecurity (Ambos configurados como *proxy* reverso). O objetivo desses testes foi comparar o jShield com o ModSecurity nos quesitos desempenho (vazão), número de usuários simultâneos que os WAFs testados suportam (escalabilidade), e também a perda de velocidade que ocorre nas transações que passam pelos mesmos (latência). Para executar esses testes utilizamos a ferramenta Apache JMeter [JMeter 2010].

O Apache JMeter, é uma aplicação *desktop* desenvolvida na linguagem de programação Java. Sendo largamente utilizando na realização de testes de carga em aplicações cliente/servidor, também podendo ser usado para simular cargas de trabalho em um servidor, rede, aplicações ou mesmo em objetos. O seu desenvolvedor original foi Stefanno Mazzochi, membro da *Apache Software Foundation*, mas hoje a ferramenta, que é *Open Source*, é resultado do trabalho de milhares de pessoas [JMeter 2010].

Para efetuar os testes de desempenho, utilizamos o *website* da faculdade CEUT (http://www.ceut.com.br/). Os testes foram divididos em três etapas, das quais o itinerário do teste era acessar a página e fazer a solicitação de uma notícia no banco de dados, sempre visando o pior caso. No primeiro caso, foram utilizados 10 usuários acessando a aplicação simultaneamente. No segundo caso, foram utilizados 20 usuários

simultâneos. E por final, no terceiro caso, foram utilizados 40 usuários simultâneos acessando a aplicação. Cada etapa do teste foi repetida 1000 vezes, e com os resultados das mesmas foi feito uma média ponderada para chegar aos valores finais obtidos.

Os testes foram realizados em uma máquina com processador dual core, 2 Gb de RAM e uma conexão com a internet do tipo 3G com a velocidade de 128 Kbps. Na maquina foram instalados e configurados um Apache Tomcat com o jShield devidamente configurado para o modelo de filtragem negativo contendo 30 regras de filtragem em sua *black list*. E um servidor Apache com o ModSecurity devidamente configurado para o modelo de filtragem negativo contendo também 30 regras em sua *black list*. Os Gráficos 1 e 2 demonstram a interpretação dos dados obtidos nos testes realizados.

Com os resultados do primeiro teste (ver Gráfico 1), podemos observar que com 10 usuários simultâneos, o tempo de resposta através do jShield foi levemente superior ao tráfego comum e ao ModSecurity, pois a média por solicitações foi aproximadamente 5 segundos mais eficiente que a média por solicitações do tráfego comum e 7 segundos mais eficiente que o ModSecurity. Porém, a vazão de dados do tráfego com jShield, obteve aproximadamente 4 operações a menos por minuto que o tráfego comum e 1 operação a mais que o ModSecurity (Gráfico 2).

No segundo resultado (ver Gráfico 1), observamos que com 20 usuários simultâneos, o tempo de resposta através do jShield foi muito superior, pois a média por solicitações, foi aproximadamente 27 segundos mais eficiente que a média por solicitações do tráfego comum e 30 segundos mais eficiente que o ModSecurity. Já a vazão dos dados foi aproximadamente 0,3 operação por minuto inferior ao tráfego comum e 0,5 operações superior ao ModSecurity (ver Gráfico 2), portanto, foram considerados tecnicamente iguais.

E por final, no terceiro caso (ver Gráfico 1), observamos que com 40 usuários simultâneos, o tempo de resposta através do jShield também foi considerado muito superior, pois a média por solicitações foi aproximadamente 25 segundos mais eficiente que a média do tráfego comum e 29 segundos mais eficiente que o ModSecurity. No entanto, a vazão dos dados foi aproximadamente 0,4 operação por minuto inferior ao tráfego comum e 0,2 operações superior ao ModSecurity (ver Gráfico 2), portanto, mais uma vez, considerados tecnicamente iguais.

Então, com os resultados obtidos, podemos concluir que as solicitações sendo encaminhadas pelo jShield, são mais eficientes quando se trata de tempo de resposta, e não causam prejuízos consideráveis ao número de operações por minuto da aplicação que está sendo protegida. Isto se deve pelo jShield possuir um algoritmo de *cache*, o que ajudou bastante nos resultados dos testes.

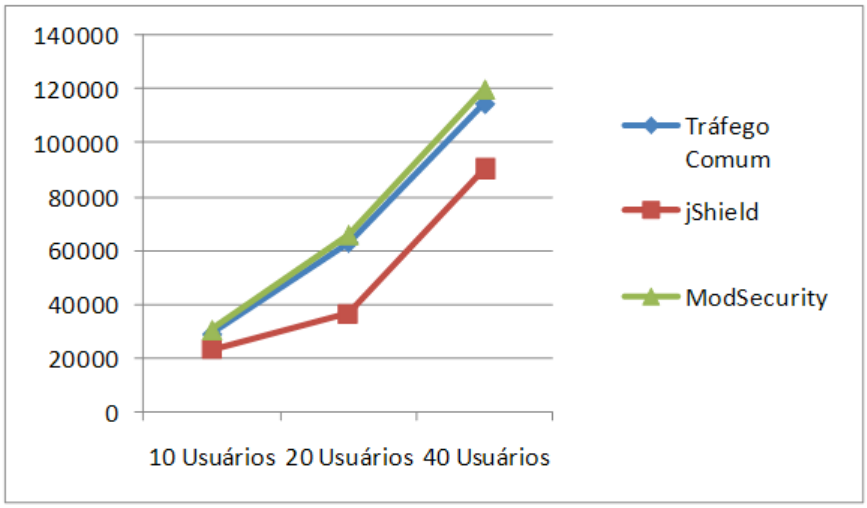

Gráfico 1. Representação gráfica do tempo de resposta.

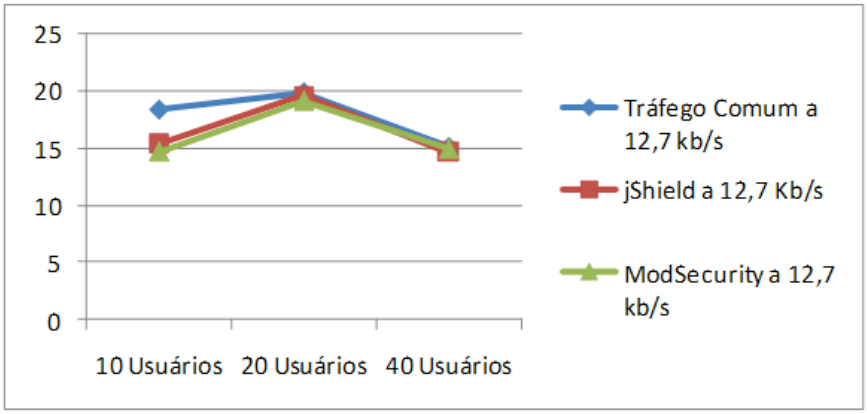

Gráfico 2. Representação gráfica de operações por minuto.

6. Conclusões

O jShield é um *firewall* de aplicação que implementa algumas características dos sistemas autonômicos e, devido a sua arquitetura de aplicação, o mesmo pode ser expandido para evoluir da categoria de software adaptativo para a de software autonômico. As funcionalidades oferecidas pelo jShield tentam tirar, senão completamente, mas grande parte da responsabilidade com a análise de segurança das aplicações web das mãos dos desenvolvedores. Assim, a sempre prioritária preocupação com as regras de negócios das aplicações pode continuar sendo o foco da equipe de desenvolvimento.

Como demonstrado nos testes, o jShield além de tornar a aplicação mais segura, agiliza o tempo de resposta das solicitações graças ao algoritmo de *caching* implementado, ao mesmo tempo que não há perdas significantes na vazão dos dados que passam pela ferramenta, tornando-o uma ferramenta extremamente atrativa pelos seus resultados.

E, por fim, a grande contribuição deste trabalho é a combinação de agentes móveis (Savant *Agent Mobile*) em *honeypots* (Loki *Honeypot*) para minerar *logs*

gerados pelos mesmos e realizar inferências para aperfeiçoar as regras do jShield de forma totalmente autônoma, tornando-o mais robusto sem, no entanto, torná-lo um gargalo evitando a configuração de regras excessivas de filtragem de pacotes.

Referências

Bace, R., Mell, P. (2001). Intrusion Detection Systems. NIST - National Institute of Standards and Technology. http://www.snort.org/docs/nist-ids.pdf.

CERT.BR (2010). Estatísticas dos Incidentes Reportados ao CERT.br. http://www.cert.br/stats/incidentes/index.html. Julho.

IBM. (1996) The Aglets Portal. http://aglets.sourceforge.net/old_site/links.html, Julho.

JMeter (2010). JMETER: Uma Aplicação desktop projetada para testes de carga e medidas de performance. http://jakarta.apache.org/jmeter/. Julho.

Jones, K. J. and Bejtlich, R. Rose, C. W. (2006). *Real Digital Forensics – Computer Security and Incident Response*, Addison-Wesley.

Lange, D. (1998). Mobile objects and mobile agents: The future of distributed computing?. ECOOP'98 - Object-Oriented Programming. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.7189&rep=rep1&type=pdf.

Macêdo, M. A.; Queiroz, R. G. and Damasceno, J. C. (2010). JShield: Uma Solução Open Source para Segurança de Aplicações Web. *VI Simpósio Brasileiro de Sistemas de Informação – SBSI*, Anais SBSI 2010.

ModSecurity (2010). ModSecurity: open source web application firewall. http://www.modsecurity.org. Julho.

OWASP (2010). The ten most critical web application security vulnerabilities. Open Web Application Security Project - OWASP. http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf.

Provos, N. and Holz, T. (2007) *Virtual Honeypots – From Botnet Tracking to Intrusion Detection*. Addison-Wesley.

WAFEC (2006). Web Application Firewall Evaluation Criteria. Web Application Security Consortium. http://www.webappsec.org/projects/wafec/v1/wasc-wafec-v1.0.pdf.

WANGHAM, M. (2004) Esquema de Segurança para Agentes Móveis em Sistemas Abertos. Tese de Doutorado, Universidade Federal de Santa Catarina, 2004. http://www.tede.ufsc.br/teses/PEEL1050.pdf.