Uma Arquitetura de Firewall derivada do OWASP ModSecurity Core Rule Set baseada em ganchos de APIs I/O

Muryllo Pimenta de Oliveira[1], Carlo Marcelo Revoredo da Silva[2]

Universidade de Pernambuco (UPE) – Garanhuns, PE – Brasil

{muryllo.pimenta[1], marcelo.revoredo[2]}@upe.br

Abstract. *The use of cloud-based firewalls such as CloudFlare has proven to be effective in containing HTTP protocol attacks targeted at the DNS domain, although it is still possible to obtain the actual address of the protected server and attack it directly by automatically querying DNS records and histories. This article introduces a firewall architecture derived from the OWASP ModSecurity Core Rule Set based on hooks of the I/O functions in the TCP socket that allow the writing of input and output rules capable of containing attacks directed to the HTTP protocol at the actual server address. Results obtained in the controlled testing environment and in the real environment showed that the applied architecture was effective in containing attempts at remote code injections, SQL injections, brute-forces, and even DoS attacks from Russian and Chinese botnets. These results were obtained through the use of rules for vectors of common attacks extracted from the Core Rule Set (CRS) combined with the use of deviations in the so-called patterns of the Express framework functions to which the firewall is directed to be used.*

Resumo. *A utilização de firewalls baseados em nuvem como o CloudFlare demonstrou ser eficaz na contenção de ataques no protocolo HTTP direcionados ao domínio DNS. Contudo, ainda assim é possível obter o endereço real do servidor protegido e atacá-lo diretamente consultando de forma automatizada registros e históricos DNS. Diante essa lacuna, este artigo apresenta uma arquitetura de firewall derivado do OWASP ModSecurity Core Rule Set baseada em ganchos das funções de I/O no socket TCP que permitem a escrita de regras de entrada e saída capazes de conter ataques direcionados ao protocolo HTTP no endereço do servidor real. Resultados obtidos no ambiente de testes controlado e no ambiente real mostraram que a arquitetura aplicada foi eficaz na contenção de tentativas de injeções remotas de código, injeções de SQL, força bruta e até ataques DoS de botnets russas e chinesas. Esses resultados foram obtidos através do uso de regras para vetores de ataques comuns extraídas do Core Rule Set (CRS) aliado ao uso de desvios nas chamadas padrões das funções do framework Express para o qual o firewall é direcionado a ser utilizado.*

1. Introdução

A arquitetura de segurança tradicional de muitos *websites* atualmente utiliza soluções em nuvem para atuar na proteção contra ataques DoS (*Denial of Service*) e tentativas de exploração na camada de aplicação, sendo bastante eficazes na sua taxa de detecção e otimização do desempenho geral dos serviços *web* (STAUFFACHER, John, 2017).

Porém, conforme o número de aplicações em nuvem vem crescendo cada vez mais, a quantidade de sistemas disponíveis em rede com configurações inadequadas de DNS vem se tornado uma preocupação (NIXON, Allison & CAMEJO, Christopher, 2013, p. 11), haja vista, serem vulneráveis à ataques no endereço de origem dos servidores reais por meio da obtenção de informações em registros DNS e ferramentas automatizadas como o CloudFail[1] e CloudMare[2].

De acordo com o 14° volume do relatório estatístico de segurança de *websites* da WhiteHat, em 2019 cerca de 32% dos ataques e vulnerabilidades exploradas foram em aplicações *web*, elevando as aplicações *web* dentre os cenários mais aproveitados dentre os existentes, demonstrando ser um resultado alarmante (WhiteHat, 2019). Com uma previsão de atingir gastos de até $ 6 trilhões de dólares para o ano de 2021 somente com cibercrimes (MORGAN, Steve. 2019), a adoção de *firewalls* é uma das medidas amplamente adotadas para conter o avanço desses números visto que possuem boas alternativas no mercado e geralmente baixos custos de implantação.

Um dos líderes de mercado, o CloudFlare, um *firewall* e rede de distribuição de conteúdo (*Content Delivery Network* – CDN) foi projetado para ter uma fácil configuração (DICKEY, Jeff, 2013), a qual qualquer proprietário de domínio possa usar sem se submeter a alterações na infraestrutura ou no código fonte do sistema, contando com balanceadores de carga e sistemas eficientes de cache que ajudam o servidor protegido a apresentar menores latências e tempos de carregamento de páginas. Embora os *firewalls* em nuvem atinjam taxas de detecção altas e números de falsos positivos extremamente baixos com proteções de alto custo benefício (Oracle, 2019), eles possuem limitações visto que sua proteção é baseada quase exclusivamente no método de mascaramento de endereço de IP do servidor de origem e desafios *JavaScript* (Pramod, N. *et al*, 2013).

Ao longo da pesquisa resultados mostram que a maior parcela dos ataques no servidor foi originada por *botnets* (dispositivos comprometidos utilizados para atacar endereços Ipv4 disponíveis) em busca de novos vetores de ataques para ampliar sua presença na *internet*. Muitos desses ataques possuem como alvo aplicações em execução sobre ambientes *Node.js*, o que justifica a criação de um WAF específico para o *Node.js,* com foco no *framework Express*. Diante desse cenário este trabalho propõe uma arquitetura minimalista de *firewall* complementar aos baseados em nuvem, fornecendo uma taxa de proteção adicional na camada de aplicação aos servidores que possuem seus endereços de IP expostos e que são atacados diariamente por *botnets*.

A arquitetura abordada utiliza um subconjunto de regras mantidos pela comunidade da OWASP (*Open Web Application Security Project*) para vetores de ataques conhecidos chamada CRS[3] (*Core Rule Set*) utilizado no projeto ModSecurity, que foram extraídos do repositório oficial e adaptados ao motor de processamento de regras do *Minimalistic WAF*. Muitos *firewalls* ainda não utilizam ou não possuem compatibilidade com regras e vetores comuns do CRS, assim como não há nenhum WAF no *Node.js* que forneça esse suporte. A abordagem da aplicação utiliza ganchos de entrada e saída (Input/Output – I/O) e filtros de requisições HTTP que podem ser configurados

[1] Repositório CloudFail: https://github.com/m0rtem/CloudFail
[2] Repositório CloudMare: https://github.com/MrH0wl/Cloudmare
[3] Projeto OWASP Core Rule Set: https://owasp.org/www-project-modsecurity-core-rule-set/

na própria aplicação residente no servidor com uma importação e uma chamada de procedimento, respectivamente.

O trabalho está organizado em 6 principais seções, sendo a seção 2 apresentando a contextualização e os *firewalls* baseados em nuvem, o principal objeto de estudo e motivador desse trabalho através da análise de suas limitações. A seção 3 aborda a proposta e a metodologia utilizada no desenvolvimento da estrutura interna de processamento de regras e prevenções de ataques do *firewall*, enquanto a seção 4 descreve o esquema utilizado como prova de conceito, contando com a aplicação controlada e experimental da arquitetura em uma máquina virtual e também da implantação em um cenário de produção, onde os ataques recebidos são originados por *botnets* e dispositivos reais. Já a seção 5 descreve os resultados obtidos na validação enquanto a seção 6, por conseguinte, apresenta as considerações finais e trabalhos futuros.

2. Contextualização

A *Internet* não foi inicialmente planejada para ser um ambiente seguro, o que mais tarde revelou ser um problema grave quando ganhou popularidade no que mais tarde ficou conhecido como "*Boom* da *Internet"* (Peterson, L. L. & Davie, B. S., 2013). Diante desse fato, muitas correções e estudos vêm sendo feitos para contornar essa realidade. Dentre os vários estudos existentes na área da segurança, há uma grande notoriedade acerca dos *firewalls*. Criados inicialmente para controlar os protocolos e portas a serem utilizados por uma rede, hoje são uma tecnologia indispensável para a infraestrutura de inúmeras redes corporativas e domésticas (SPEED, Tim, 2003).

Os *firewalls* tiveram sua primeira aparição durante a década de 80 começando com os filtros de pacotes, tecnologia desenvolvida para inspecionar pacotes em tráfego evoluindo em *firewalls* de inspeção de estado, posteriormente evoluindo em *firewalls* de aplicação, principal objeto de estudo desse trabalho (SPEED, Tim. 2003).

2.1. Conceitos sobre Firewall

Os *firewalls* são sistemas de proteção que aplicam políticas de segurança em uma rede de computadores, atuando de forma física como um *hardware* ou lógica enquanto um *software*. Suas principais funcionalidades numa rede incluem:

- Bloqueio de portas e protocolos em conexões no fluxo de entrada e saída;
- Restrição do tráfego de rede baseado no endereço de IP;
- Recusar pacotes com sinalizadores e regiões do cabeçalho TCP adulterados;
- Bloquear tentativas de acesso não autorizado e exploração de falhas em serviços em execução;
- Políticas de controle de acesso à serviços e registro de atividades suspeitas.

De acordo com a literatura popular, há pelo menos 3 tipos de *firewalls*, sendo os *proxies* e *firewalls* de aplicação mais abordados neste trabalho. O primeiro tipo, conhecido como filtro de pacotes atua a nível de rede e controla o fluxo de conexões com base em seu protocolo e endereço de IP das partes envolvidas (SPEED, Tim, 2003). Filtros de pacote não entendem os protocolos de aplicação (ROMANOFSKI, Ernest. 2002), já o segundo tipo, *firewall* de inspeção de estado, monitora as conexões que estão sendo iniciadas, que foram estabelecidas e as que estão em etapa de finalização, salvando

dados como o endereço de IP e a porta utilizada no ciclo de vida da conexão, além de inspecionar se a informação nos cabeçalhos dos pacotes são válidas (SPEED, Tim, 2003).

O terceiro tipo de *firewall* é conhecido como *Application gateway* ou *proxy-firewall* que atua nas camadas 3, 4, 5 e 7 do Modelo OSI (*Open System Interconnection*). Esse tipo de *firewall* entende o significado dos dados trafegados e qual o protocolo de aplicação está sendo utilizado (SPEED, Tim, 2003). Os *firewalls*, no geral, possuem 3 tipos de ações principais executadas quando um pacote novo adentra a rede, são elas: *Deny*, *Accept* ou *Reject*. Como os nomes sugerem, *Deny* nega a conexão, *Accept* aceita a conexão e *Reject* rejeita a conexão.

2.2. Firewalls e suas aplicabilidades

Um levantamento realizado pela *eSecurity Planet*[4] apontou os 10 mais populares *firewalls* baseados em nuvem, dentre os quais estão contidos o Imperva Incapsula, CloudFlare e o Akamai (ROBB, Drew, 2018). O último levantamento quantitativo feito no ano de 2017, pela CloudFlare em página oficial (CloudFlare, 2017), apontou que cerca de 12 milhões de *websites* são protegidos somente pelas soluções Anti-DDoS (*Destributed Denial of Service* - DDoS). Não obstante, levantamentos feitos pela *W3Techs Web Technology Surveys* mostraram que cerca de 13,6% de todos os *websites* da amostra estudada utilizam o CloudFlare como a principal solução de proteção de seus serviços.

O CloudFlare é uma das maiores redes de distribuição de conteúdo do mundo (DICKEY, Jeff, 2013) que possui em sua composição, por padrão, o CRS ModSecurity da OWASP (KAUSHIK, Mehul, 2017), sendo extremamente eficaz na detecção de ameaças à aplicativos *web* além de utilizar suas próprias regras. Haja vista sua vasta amplitude e sua crescente demanda, suas maiores e mais conhecidas limitações estão no fato de atuar apenas no contexto de nuvem, ou seja, atuar apenas como um *proxy* reverso e realizar o mascaramento do endereço de origem (NIXON, Allison & CAMEJO, Christopher, 2013, p. 11). A CloudFlare vem modernizando seus planos de proteção na tentativa de atenuar o problema, inclusive com a disponibilização do sistema *Argo Tunnel* que atua na proteção e redução da superfície de ataques aos endereços de origem, mesmo que estes possuam portas abertas e acessíveis por redes externas (CloudFlare, 2020). A medida é válida e expõe uma arquitetura de tunelamento versátil e de fácil configuração, embora seja obtida com custos adicionais (CloudFlare, 2020).

2.3. Trabalhos Relacionados

Os estudos de [Clincy, V., & Shahriar, H., 2018] enaltecem a necessidade de WAFs bem configurados para evitar a falsa sensação de segurança. Neste trabalho foram utilizadas simplificações do modelo de melhoria no processo de criação de regras dos WAFs e que diminuem o número de erros de regras proposta em [Ahmad, A. *et al*, 2012] nas ações disruptivas e condições dos filtros. A iniciativa de expandir a influência do CRS ModSecurity para o ambiente Node.js foi fortemente influenciada pelo trabalho de [YARI, Imrana A. *et al*, 2019] que utilizou uma abordagem de testes semelhante com o DVWA e portou o ModSecurity para o ambiente do Apache Tomcat.

[4] https://www.esecurityplanet.com/products/top-ddos-vendors.html

Diante de iniciativas frustradas de se criar um WAF modelo *standalone* que permitisse uma implantação barata e prática na infraestrutura de aplicações que utilizam o *framework express*, surgem 3 projetos com linhas de pesquisa semelhantes embora metodologias de aplicações distintas. No ano de 2016 a primeira iniciativa promissora foi de um desenvolvedor de *software* alemão chamado Daniel Ruf. O projeto tratava-se de um *firewall* instalado diretamente na camada de aplicação e que não utilizava o conceito de *proxy* reverso ou *daemons* em execução. Por volta do ano de 2017 o projeto foi descontinuado devido à falta de atualizações do desenvolvedor, embora ainda esteja disponível no repositório oficial do NPM (*Node Package Manager*) sob o nome *node-waf*[5].

No ano de 2017, um projeto sucessor chamado *express-waf*[6] foi publicado e logo em seguida descontinuado pelo desenvolvedor. A proposta do projeto era de utilizar o MongoDB como unidade de armazenamento das regras e filtros também contando com a disponibilidade de uma camada de API própria que permita a criação de módulos de terceiros. O projeto foi disponibilizado como código aberto e encontra-se também disponível no repositório do NPM. Já em 2019, outro projeto chamado *waf-js*[7] surge com uma proposta dessa vez não tão promissora visto que que fornecia apenas um *firewall* com uma taxa de detecção muito inferior que era baseada exclusivamente em listas de palavras, termos conhecidos e expressões regulares para a busca de ameaças.

A arquitetura de *firewall* apresentada, nomeada de "WAF Minimalista" ou Mini-WAF, utilizou um subconjunto de regras exposto no repositório oficial do modelo de regras adotado no ModSecurity, o chamado *Core Rule Set* (CRS) que cataloga os principais vetores de ataques na *web*. O CRS está em desenvolvimento ativo e sob apoio da comunidade OWASP desde o ano de 2010. Sendo um dos *firewalls* mais conceituados, a arquitetura abordada neste trabalho foi motivada pela falta de portabilidade do CRS nos projetos do *Node.js* uma vez que o ModSecurity não possui uma extensão para aplicações baseadas no *express*.

3. Proposta

Conforme aborda a seção 2.3, muitos sistemas *web* necessitam de proteções contra ataques na camada de aplicação originados por *botnets* e agentes perigosos que conseguem burlar os *firewalls* baseados em *proxies* externos e redes *AnyCast*. A arquitetura abordada está dividida em 6 principais estruturas detalhadas a partir da seção 3.2 até a seção 3.6:

1. O motor e interpretador de regras;
2. As listas discricionárias de controle de acesso;
3. Os filtros de entrada e saída;
4. A fila de *callbacks* de pós operação de filtro;
5. O manipulador global de exceções não tratadas;
6. A pilha de ganchos no objeto *Response* de aplicações HTTP do Node.

Abordagens semelhantes utilizando ganchos em funções de nível de usuário nos sistemas operacionais da *Microsoft* tem sido utilizado há anos pelas empresas de antivírus

[5] https://www.npmjs.com/package/node-waf
[6] https://www.npmjs.com/package/express-waf
[7] https://www.npmjs.com/package/wafjs

para mitigar a propagação de *malwares*, elevações de privilégios e comportamentos suspeitos dos *softwares* (Joxean Koret *et al*, 2015). Não obstante, os métodos supracitados foram amplamente adotados na arquitetura de *firewall* minimalista como método para detecção de anomalias nas respostas dos pedidos HTTP.

É importante destacar que nem todos os ataques podem ser contidos por esta abordagem de *firewall*, já que a sua principal vantagem está na proteção adicional da origem em servidores eventualmente vulneráveis em camada de aplicativo. Sendo assim, as boas práticas de programação e desenvolvimento WEB seguras não devem ser negligenciadas, cabendo ao desenvolvedor projetar um sistema tolerante a falhas e seguro.

3.1. Ciclo de vida de uma requisição HTTP no Express

No interpretador Node.js há uma API padrão para manipulação de requisições HTTP utilizando um único retorno de chamada (HAHN, Evan. Express In Action, 1st edition, 2016). Assim como em outras plataformas, o Node.js carrega uma camada superficial de APIs que atendem às solicitações TCP, UDP e HTTP. Em sua forma tradicional, a API utiliza apenas uma função para gerenciar todas as rotas, métodos e respostas, conforme ilustra a Figura 1.

Figura 1. Ciclo de vida de uma requisição com apenas um controlador no Node.js. Fonte: (HAHN, Evan. Express In Action, 1st edition, 2016).

De acordo com o esquema demonstrado anteriormente, é possível notar que a manipulação das solicitações provenientes da *web* se tornaria muito complexa caso um único controlador fosse utilizado chamada (HAHN, Evan. Express In Action, 1st edition, 2016). Diante deste cenário, surge o *Express* com uma proposta de manipulação baseada em pilhas de manipuladores de solicitações, conforme descreve a Figura 2.

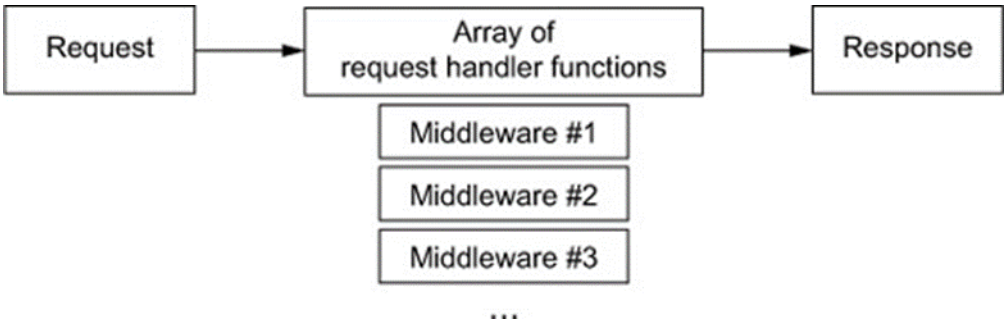

Figura 2. Ciclo de vida de uma requisição através dos middlewares no Express Fonte: (HAHN, Evan. Express In Action, 1st edition, 2016).

A abordagem orientada a pilhas de manipuladores permite a inclusão de um manipulador da arquitetura de *firewall* proposta neste trabalho e realça a característica minimalista e de baixo nível técnico necessário para a implementação da camada de proteção. De acordo com repositório oficial do projeto *Express*[8], a análise de *cookies*, cadeia de consulta, rotas e cabeçalhos é feita pelo próprio *framework*, reduzindo esforços

[8] Repositório do *Express*: https://github.com/expressjs/express

adicionais na implementação do processador e interpretador de regras do *firewall* abordado na seção seguinte.

3.2. O motor e interpretador das regras e a sua composição

No *firewall* minimalista, o motor e interpretador de regras utiliza um objeto de inicialização dividido em DACLs, filtros contendo regras adaptadas do CRS, uma fila de *callbacks* disparada a cada requisição permitida de forma explícita ou implícita e uma tabela de acesso ou pilha de acesso, conforme ilustra a Figura 3:

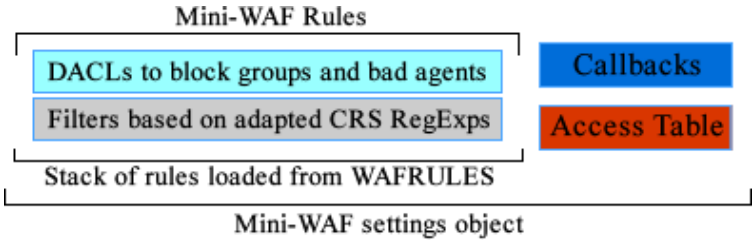

Figura 3. Estrutura do objeto de inicialização do interpretador de regras.

O esquema representa a pilha principal do interpretador de regras. As regras são divididas em DACLs e filtros, podendo conter múltiplas delas. Em teoria esse modelo difere das regras de *firewalls* de filtro de pacotes e inspeção de estado pois não há presença de DACLs. O motor de varredura das solicitações HTTP percorre todos os cabeçalhos presentes no objeto *Request.headers* exposto pelo *Express*, bem como o corpo da solicitação.

No nível das DACLs há a inspeção do endereço de IPv4 ou IPv6 utilizado para realizar a conexão. Determinados endereços ou até grupos podem ser bloqueados, com base no cabeçalho de agente de usuário, método utilizado e o próprio endereço remetente. O interpretador decide com base nos resultados das DACLs e filtros se a requisição ou a resposta serão aceitas, caso contrário, a resposta é recusada com um código HTTP 403.

Sendo exposto como uma API baseada em *middleware*, o interpretador é carregado utilizando o método $waf.WafMiddleware(wafrules.DefaultSettings)$. Por conseguinte, o interpretador das regras foi projetado internamente para atender a bandeiras binárias. Cada regra possui múltiplas opções que são extraídas do objeto passado como parâmetro no método citado e são verificadas conforme mostra a seção seguinte.

3.3. As Listas Discricionárias de Controle de Acesso do WAF

O primeiro elemento constituinte de uma regra do *firewall* proposto é a DACL. Com elas, é possível delimitar grupos que podem ou não acessar o serviço exposto na *web*. Na família de sistemas operacionais Windows NT, as DACLs são utilizadas para controlar os acessos dos usuários de forma individual à objetos, enquanto as ACLs (*Access Control Lists*) são utilizadas para descrever quais objetos do sistemas e serviços possuem acesso à determinados objetos (Kris Jamsa, Lars Klande, Hacker Proof, 2nd Edition, 2002).

No contexto do *firewall* proposto, as DACLs atuam sobre 3 dados na solicitação HTTP que identificam o cliente, são eles: [1] o endereço de IP do remetente; [2] o método utilizado; [3] o cabeçalho de agente de usuário. A atuação de uma DACL configurada

para bloquear solicitações com o método *DELETE*, considerado inseguro, é ilustrada pela Figura 4 (RFC 7231, R. Fielding *et al*, 2014).

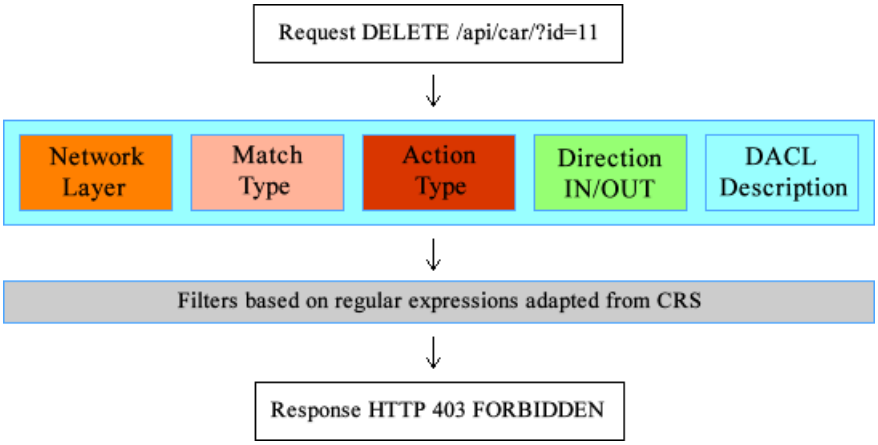

Figura 4. Estrutura de uma das DACLs do firewall ilustrada na cor azul.

A DACL é composta pelos seguintes sinalizadores binários:

- *Network Layer*: protocolo de camada de rede utilizado (IPv4, IPv6 ou ambos);
- *Match Type*: tipos de correspondência que devem acionar a ação da DACL;
- *Action Type*: tipo de ação que deve ser tomada (bloquear, permitir ou auditar);
- *Direction*: direções em que a DACL deve atuar (entrada, saída ou ambos);
- *Description*: a descrição em forma de texto da regra, utilizada para identificar o evento quando ocorrido.

O interpretador das regras é o principal ator haja vista ser quem realiza a inspeção da solicitação e da resposta, analisando de forma recursiva, cada DACL uma após a outra. Sempre que uma DACL tem sua condição correspondida na solicitação e a ação pretendida é a de bloqueio, o interpretador continua com a verificação até atingir o final da pilha de DACLs e filtros ou encontrar uma DACL ou filtro que permita explicitamente a solicitação. Nesse caso, ações de permissão explícita são mais valorizadas que ações de bloqueio, o que implica afirmar: uma única DACL na pilha que permita explicitamente a solicitação anulará todas as DACLs ou filtros de I/O que a neguem o acesso.

3.4. Os Filtros de I/O do WAF

A camada de proteção que sucede a pilha de DACLs é a pilha de filtros de I/O. Os filtros, na arquitetura do WAF minimalista, são objetos contendo fragmentos de *exploits* (instruções utilizadas para explorar uma vulnerabilidade) utilizados para identificar comportamentos nocivos à aplicação (MORENO, Daniel. p. 204-206. 2017). O *firewall* vem com dezenas de filtros prontos com vetores comuns extraídos do CRS.

Conforme demonstrado na seção anterior, as DACLs atuam sob grupos de clientes e redes, já os filtros trabalham de forma diferente pois atuam sobre as características da solicitação e da resposta, sendo analisados: [1] cabeçalhos; [2] cadeia de consulta; [3] cadeia de parâmetros; [4] número de tentativas; [5] corpo da solicitação; [6] *cookies*; [7] extensões de arquivos recebidos. A Figura 5 ilustra a estrutura interna da segunda camada de proteção.

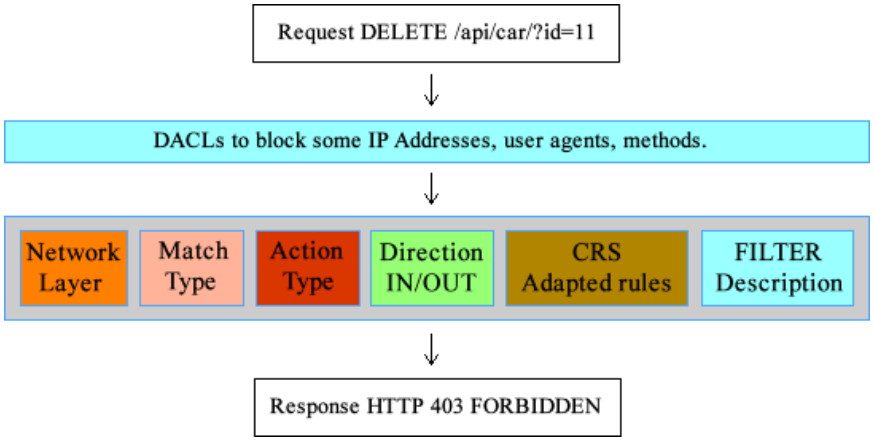

Figura 5. Estrutura de um dos filtros do firewall ilustrado na cor cinza.

O Filtro de I/O é composto pelos seguintes sinalizadores binários, muito semelhantes aos sinalizadores da DACL:

- *Network Layer*: protocolo de camada de rede utilizado (IPv4, IPv6 ou ambos);
- *Match Type*: tipos de correspondência que devem acionar a ação o filtro;
- *Action Type*: tipo de ação que deve ser tomada (bloquear, permitir ou auditar);
- *Direction*: direções em que o filtro deve atuar (entrada, saída ou ambos);
- *CRS Adapted rules*: cadeia de regras em formatos de texto puro ou expressões regulares que foram adaptadas de subconjuntos de regras do CRS;
- *Description*: a descrição em forma de texto da regra, utilizada para identificar o evento quando ocorrido.

As expressões regulares passam por adaptações antes de se tornarem legíveis pelo interpretador de regras. O CRS introduz o conceito de paranoia que não é utilizado na arquitetura, além de impor regras específicas para determinados cabeçalhos HTTP, como o *Referer*. Uma regra de proteção contra ataques de inclusão local de arquivos (*Local File Inclusion* – LFI), por exemplo, escrita como $@rx\ (?:\hat{}|[\backslash/])\backslash.\backslash.(?:[\backslash/]|\$)$ no CRS pode ser traduzida para o formato do Mini-WAF obtendo-se a expressão regular $/(\backslash.\backslash.(\backslash/|\backslash\backslash)|\backslash.\backslash.\%(2F|5C))+/$.

3.5. Os Callbacks de pós operações e os manipuladores de exceções

Os *callbacks* de pós operações de filtro são executados exclusivamente quando o interpretador das regras conclui todas as varreduras especificadas pelos sinalizadores binários dos filtros. Cada *callback* possui uma declaração similar à $postFltCallback(reqRef, resRef): void$ que quando executado recebe duas referências aos objetos de requisição e resposta, respectivamente. Os *callbacks* possuem um novo método adicionado à referência do objeto original de resposta declarado $Drop(): void$ que não recebe parâmetros opcionais e pode ser utilizado para descartar uma requisição recebida, impedindo que a camada de aplicação interprete-a caso varreduras adicionais sejam feitas na requisição utilizando lógica proprietária da aplicação, levando em consideração regras de negócio do sistema.

O processo de registro de um retorno de chamado de filtro é semelhante ao processo feito em *drivers* de filtro de rede com funções de *callout* nos sistemas Microsoft Windows (Microsoft, 2019). A biblioteca base do *firewall* expõe duas APIs de registro e

remoção do *callback* personalizado, respectivamente. A API de registro, declarada como $WafRegisterCallback(wafObj, lpCallback)$ recebe dois parâmetros, sendo o primeiro uma referência para o objeto de inicialização utilizado pelo interpretador das regras explanado na seção 3.2 enquanto o segundo parâmetro contém uma referência para a função que deverá ser executada após a operação de varredura de filtro concluída com sucesso onde nenhuma regra bloqueie a solicitação de entrada. A função retorna um identificador único tal qual um *handle* em APIs do sistema Windows, entretanto, na forma de um identificador único universal (Mark E. Russinovich *et al*, 2011).

Já a função de remoção do *callback*, declarada como $WafUnregisterCallback(wafObj, callbackUuid)$ recebe o mesmo primeiro parâmetro especificado na API de registro enquanto o segundo é o identificador do retorno de chamada registrado. Além dos *call-backs* personalizados, o *firewall* conta com o seu próprio manipulador de exceções globais no aplicativo que é registrado também como um *callback* padrão, ou seja, pré-configurado para impedir que a aplicação protegida seja derrubada por exceções não tratadas.

3.6. Os Ganchos nas APIs I/O

Essa seção aborda a técnica utilizada pelo *firewall* para contornar o problema da falta de uma API para interceptar os eventos de escrita através do objeto de resposta fornecido pelo Node.js. O *firewall* implementa uma rotina de engate, que utiliza o conceito de método *stub*, utilizado para substituir um trecho de código existente. O método $Hook(targName, ptrStub, ptrParent)$ que recebe três parâmetros é utilizado no processo de substituição de 7 funções de escrita contidas no objeto de resposta. A referência das funções originais é armazenada em propriedades dentro do objeto de resposta que contém um decorador de nome $\_\_unhooked\_\_original\_\_$, sendo o objeto real substituído pela referência $ptrStub$.

Uma vez que todos os ganchos foram definidos, as operações de escrita utilizando os métodos são submetidos à análise de todos os filtros com os sinalizadores binários **MATCH_HEADERS** e **MATCH_PAYLOAD**. O primeiro tipo de gancho é realizado na função $res.header()$ que lida com o envio de dados ao cliente e sua referência original é armazenada na propriedade $res.\_\_unhooked\_\_original\_\_header$ e só é analisada caso o sinalizador binário de análise de cabeçalhos esteja definido. As demais funções recebem ganchos que só sofrem análise quando o filtro define o sinalizador de análise de carga útil ou *payload,* sendo elas: [1] *json*; [2] *jsonp*; [3] *write*; [4] *set*; [5] *send*; [6] *end*.

4. Validação do Firewall

A validação do *firewall* foi dividida em 2 principais cenários, o de ambiente controlado onde os testes foram divididos em módulos vulneráveis do DVWA e o de ambiente real (não controlado) onde um servidor *web* foi implantado num dispositivo IoT (*Internet of Things*) sob uma placa *Tinkerboard R/BR.* Visando comprovar a eficácia da arquitetura de *firewall* proposta, a placa esteve conectada à *internet* executando dois serviços HTTP e HTTPS, respectivamente, ambos com o *Minimalistic* WAF instalado. O servidor, acessível a partir de um nome de domínio protegido pela CloudFlare foi analisado numa ferramenta chamada CloudFail que conseguiu obter com êxito o endereço real do servidor, o qual foi alvejado por ataques de injeções de SQL.

4.1. Validação em ambiente controlado

O ambiente controlado que se encontra disponível no repositório do GitHub é uma imagem de Sistema operacional Lubuntu (LXQt 20.04) modificada com o Apache e o conhecido DVWA instalados. A imagem vem com um *script* de instalação padrão do ambiente que realizará a configuração e instalação da versão de testes mais atual além de confirmar se os serviços necessários para o experimento estarão em execução. Conforme a proposta da arquitetura do *firewall,* ele deverá atuar como um *proxy* local e redirecionar as requisições legítimas da porta 80, padrão do HTTP, para a porta 8081 onde o DVWA efetivamente aceita e processa as requisições. A Figura 6 ilustra como o *firewall* atuará na proteção da aplicação DVWA comprometida.

Figura 6. Esquema de atuação do Mini-WAF na proteção da aplicação DVWA.

O atacante, neste caso o pesquisador, deve efetuar o *pentest* na aplicação DVWA através do *localhost*, simulando assim um ataque direcionado ao IP externo da rede onde o serviço a ser protegido roda por trás do *firewall* minimalista. A aplicação vulnerável vem com o nível de dificuldade definido, por padrão, como "impossível" embora possa ser alterado conforme a necessidade do pesquisador.

4.2. Validação em ambiente real

No ambiente real, o *firewall* foi instalado e colocado em execução em um servidor com um nome de domínio protegido pelo CloudFlare que encontra-se vulnerável visto que foi propositalmente configurado anteriormente para apontar os endereços reais do servidor, o que ocasionou o armazenamento da origem em registros DNS (NIXON, Allison & CAMEJO, Christopher, 2013, p. 11). Uma ferramenta chamada CloudFail foi utilizada no processo descrito a seguir para extrair o endereço de origem e efetuar tentativas de injeções SQL no parâmetro *id* da página *index.php*.

1. O nome de domínio é escaneado pela ferramenta, verificando se o domínio está sob proteção da rede *AnyCast* da CloudFlare;
2. Caso confirmada a presença de um WAF a ferramenta dnsdumpster é utilizada para obter informações a partir de DNS mal configurado;
3. A ferramenta verifica cada endereço de IP obtido na etapa anterior pela enumeração dos nomes de domínio e com uma requisição do tipo *GET* com o cabeçalho *Host* adulterado;
4. Para cada resposta obtida nas requisições da etapa anterior, é possível identificar se o servidor responderá com conteúdo semelhante ao apresentado quando acessado diretamente pelo nome de domínio protegido pelo WAF. Nesse caso, a ferramenta realiza uma comparação textual e caso identificada uma grande aproximação entre as respostas, a ferramenta pressupõe com uma margem de erro que encontrou o endereço de origem.

Para mitigar a falha demonstrada anteriormente, práticas podem ser adotadas como a não utilização de subdomínios com referência direta à origem, evitar conexões diretas aos *Hosts* virtuais direto no servidor de origem e implementar políticas de filtro de serviços SMTP pois estes podem ser utilizados para obter o endereço de origem. Aliando às práticas anteriores, todas as conexões devem ser feitas exclusivamente pelo WAF baseado em *Cloud*, ou seja, a aplicação deve verificar se a requisição é originada pelo *gateway*. Visando a comprovação da ferramenta em um cenário real, ela também foi exposta em serviços que já estavam em execução, como em uma API de coletada de dados estatísticos da pandemia do coronavírus[9].

5. Resultados

Conforme visto na seção 4.2, o *firewall* foi exposto em um cenário de aplicação real por 3 meses, começando em maio até julho do ano de 2020. O sistema recebeu atualizações com frequência sempre que novos vetores de ataques foram identificados, acumulando um total de 34 versões diferentes. Diante do cenário, o dispositivo IoT conectado à rede foi atacado 3015 vezes, sendo a maior parcela dos ataques originadas por *botnets* russas, chinesas, alemãs e estadunidenses. Após um filtro no histórico de ataques da ferramenta, as requisições recebidas que foram enquadradas na classificação C08 (negação de serviço) foram removidas, resultando em 753 ataques com intervalos médios ou longos. Os resultados foram obtidos seguindo uma análise quantitativa na etapa de testes de cenário real e classificados, conforme destaca a Tabela 1.

Tabela 1. Classificação e quantitativo de ataques bloqueados durante os testes.

Classificação do Ataque	Bloqueados	Percentual
C01. Remote shellcode execution	120	15,93%
C02. Remote code execution	51	6,77%
C03. Path traversal	10	1,32%
C04. Known botnets	43	5,71%
C05. Remote file inclusion	82	10,88%
C06. Cross-site scripting	3	0,39%
C07. Data exposure	1	0,13%
C08. Denial of Service	3	0,39%
C09. Admin SQL Injection bypass	16	2,12%
C10. Advanced SQL Injection	280	37,18%
C11. Blind SQL Injection	69	9,16%
C12. RLIKE or WAITFOR based SQL Injection	49	6,50%
C13. Negative number SQL Injection	23	3,05%
C14. Null-byte based SQL Injection	2	0,26%
C15. Null character bypass file upload	1	0,13%

Os endereços de IP russos: 5.101.0.209 e 195.54.160.135 foram os mais detectados com tentativas de execuções remotas de código. Com 8 tentativas por dia, em média, as duas *botnets* procuram por endereços com as portas 80 e 443 disponíveis na tentativa de injetar código em aplicações comprometidas que utilizam o *framework* ThinkPHP. De acordo com o histórico de tentativas não-autorizadas da ferramenta, dos 753 ataques originados por *botnets*, 106 foram exclusivamente dos endereços

[9] API Covid-19 disponível em: https://documenter.getpostman.com/view/11185707/SzmmVvBZ

195.54.160.130 e 195.54.160.135. A Figura 7 ilustra o firewall detectando uma tentativa real de injeção de código na máquina através da *string* de consulta.

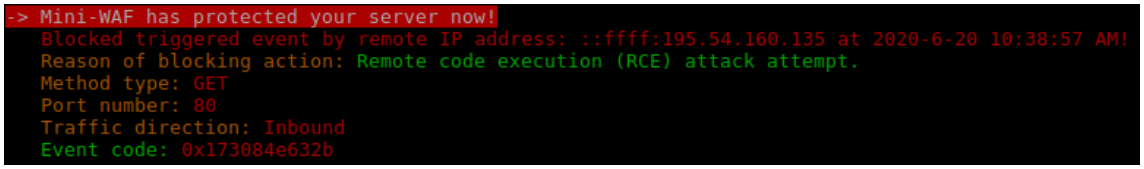

Figura 7. Tentativa de execução remota de código bloqueada pelo Mini-WAF.

O *firewall* demonstrou ao longo do seu período de testes no ambiente real uma alta resiliência, visto que notificou também sobre exceções não-tratadas no servidor que poderiam derrubar toda a aplicação. Seu módulo *built-in* de manipulador de exceções universais conseguiu impedir que exceções de qualquer gênero, causadas por eventos intencionais ou não-intencionais ocasionassem uma negação de serviço.

6. Conclusões e Trabalhos Futuros

Esse trabalho apresentou uma arquitetura de *firewall* para oferecer a portabilidade de um subconjunto seleto de regras para vetores de ataques comuns contidos no CRS. O modelo de *firewall* baseado em ganchos de funções de escrita aliado ao padrão de projeto de aplicação *web* baseado no uso de *middlewares*, tal como o *framework express*, demonstrou ser extremamente versátil para a escrita do WAF, uma vez que tornou o processo simplista e barato quando levada em consideração a implantação.

Várias das soluções que tentaram prover uma ferramenta de proteção às aplicações baseadas no *express* falharam, conforme aponta a seção 2.3. De acordo com os resultados obtidos pelo Mini-WAF, a arquitetura demonstrou ser promissora e será mantida em repositório público recebendo atualizações para aumentar a quantidade de regras portadas do CRS, e ampliar o poder de detecção da ferramenta para os aplicativos feitos em Node.

Como trabalhos futuros, a ferramenta terá seu banco de dados de regras ampliado e mais vetores de ataques conhecidos, portados do CRS pelo processo de tradução de regras descrito na seção 3. Além da expansão do banco de regras, a ferramenta ganhará um conjunto de *drivers* para difundir o trabalho da arquitetura, com possíveis candidatos os *frameworks* baseados em Java, Python e C# como o *Spring Boot*, *FastAPI*.

Referências

Ahmad, A, *et al* (2012). “Formal reasoning of web application Firewall rules through ontological modeling”, July.

ARNFELD, Tom. (2017) “How we made our DNS stack 3x faster”, The CloudFlare Blog, July, available at: https://blog.cloudflare.com/how-we-made-our-dns-stack-3x-faster/.

Clincy, V., & Shahriar, H. (2018). “Web Application Firewall: Network Security Models and Configuration”. July.

CloudFlare. (2020) “Argo Tunnel”, July, available at: https://www.cloudflare.com/pt-br/products/argo-tunnel/.

DICKEY, Jeff. (2013) “Instant CloudFlare Starter”, 1st Edition, June.

FIELDING, R. *et al*. (2020) “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content”, RFC 7231 – Internet Engineering Task Force (IETF), July.

HAHN, Evan. (2016) “Express in Action: Writing, building, and testing Node.js applications”, 1st Edition, July.

KAUSHIK, Mehul. (2017) “Cloudflare Web Application Firewall Review”, CualHost, July, available at: https://www.cualhost.com/cloudflare-web-application-firewall-review/.

KLANDER, Lars. (1997) “Hacker Proof: The Ultimate Guide to Network Security”, July.

KORET, Joxean & BACHAALANY, Elias. (2015) “The Antivirus Hacker’s Handbook”, June, p. 174.

Microsoft. (2019) “Windows Filtering Platform Callout Drivers”, Microsoft Developer Network, July.

MORGAN, Steve. (2019) “2019 Official Annual Cybercrime Report”, Herjavec Group, June.

MORENO, Daniel. (2017) “Pentest em Aplicações Web”, 1th edition, June.

NIXON, Allison & CAMEJO, Christopher. (2013) “DDoS Protection Bypass Techniques”, Integrallis Inc, June.

Oracle. (2019) “5 Reasons Why You Need a Cloud-based Web Application Firewall”, July, p. 2 – 3.

Pramod, N. *et al* (2013) “Limitations and Challenges in Cloud-Based Applications Development”, July.

Peterson, L. L. & Davie, B. S. (2013) “Redes de Computadores – Uma abordagem de sistemas”, Elsevier, 5th Edition, July.

ROBB, Drew. (2018) “Top 10 Distributed Denial of Service (DDoS) Protection Vendors”, eSecurity Planet, July, available at: https://www.esecurityplanet.com/products/top-ddos-vendors.html.

ROMANOFSKI, Ernest. (2002) “A Comparison of Packet Filtering vs Application Level Firewall Technology”, SANS Institute, June, p. 1-6.

RUSSINOVICH, Mark *et al*. (2011) “Windows Sysinternals Administrator's Reference”, 1st Edition, July.

SPEED, Tim. (2003) “Internet Security”, Elsevier, 1st Edition, July.

STAUFFACHER, John. (2017) “Web Application Firewalls: A Practical Approach”, 1st Edition, June.

WhiteHat Security. (2019) “Application Security Statistics Report”, vol. 14, June, p. 2.

W3Techs. (2020) “Usage statistics of reverse proxy services for websites”, W3Techs Web Technology Surveys, July, available at: https://w3techs.com/technologies/overview/proxy.

YARI, Imrana A. *et al*. (2019) “Towards a Framework of Configuring and Evaluating ModSecurity WAF on Tomcat and Apache Web Servers”, ICECCO, July.