PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Web Application Firewall as a Service

Um estudo prático em segurança de rede

Arthur Cavalcante Gomes Coelho

Projeto Final De Graduação

Centro Técnico Científico - CTC

Departamento de Informática

Curso de Graduação em Ciência da Computação

Rio de Janeiro, Junho de 2021

Arthur Cavalcante Gomes Coelho

Web Application Firewall as a Service

Um estudo prático em segurança de rede

Relatório de Projeto Final, apresentado ao programa Ciência da Computação da PUC-Rio como requisito parcial para a obtenção do Bacharel em Ciência da Computação.

Orientadora: Noemi Rodriguez

Rio de Janeiro

Junho de 2021.

Sumário

Agradecimentos

À minha orientadora Noemi Rodriguez por todo carinho, apoio e principalmente paciência com todos os erros de português que a obriguei a presenciar durante o desenvolvimento do projeto.

Ao meu colega de trabalho e amigo pessoal Claudio Netto, por compartilhar comigo a ideia inicial do projeto, sem o qual não seria possível sua realização.

Aos meus pais Beatriz Bergamini e Luiz Cristóvão por todo apoio e carinho em um momento tão conturbado, graças a eles tive a oportunidade que muitos não conseguem ter, um ambiente estável de vivência onde pude me desenvolver e aprender até esse ponto.

Aos meus colegas do time Tsuru, Bernardo Lins, Cezar Espínola, Paulo Sousa e Wilson Júnior que me acolheram e me ensinam todos os dias como ser uma pessoa e um profissional melhor.

Resumo

Coelho, Arthur. Rodriguez, Noemi. Web Application Firewall as a Service, Um estudo prático em segurança de rede. Rio de Janeiro, 2021. 54p. Relatório de Projeto Final – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Este projeto visa disponibilizar um sistema, para desenvolvedores de aplicações web que utilizam a plataforma Kubernetes, que possibilita a criação e configuração de Web Application Firewalls para suas aplicações. Dessa forma, cria-se uma camada de proteção à frente da aplicação com o intuito de identificar e interceptar possíveis ataques cibernéticos.

Palavras-chave

Web Application Firewall, Kubernetes, microsserviços, Docker, conteinerização, proxy reverso, segurança

Abstract

Coelho, Arthur. Rodriguez, Noemi. Web Application Firewall as a Service, Um estudo prático em segurança de rede. Rio de Janeiro, 2021. 54p. Relatório de Projeto Final – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

This project aims to provide a system that allows developers, deploying their applications in Kubernetes, to easily create and configure their own Web Application Firewalls. The goal is to create a new layer of security for their application with the intent to identify and mitigate possible cyberattacks.

Keywords

Web Application Firewall, Kubernetes, microservices, Docker, containerization, reverse proxy, security

1- Introdução

Com o crescente número de empreendimentos que disponibilizam seus serviços através da internet, dados sensíveis como cartões de crédito e credenciais de clientes são armazenados em serviços comumente denominados *back-ends* que são acessados por aplicações do tipo web que eventualmente armazenam essas informações em um banco de dados. Tais aplicações tornam-se alvos frequentes de tentativas de violação. Para o empreendimento não se tornar vítima de extorsão ou vazamento de dados, uma série de medidas de segurança podem ser implementadas para que as aplicações se tornem mais seguras.

Uma solução que vem ganhando popularidade no ramo de segurança da informação e que vem sendo utilizada e comercializada por empresas como Amazon e Google, se chama Web Application Firewall (WAF)[1]. Uma WAF funciona como a primeira linha de defesa de uma aplicação web. Além de proteger a aplicação, ela tem sido utilizada como uma forma de notificar a equipe desenvolvedora de possíveis falhas de segurança, fazendo com que a equipe tenha tempo de mitigá-las antes que a aplicação seja exposta à Internet evitando que o empreendimento tenha dados confidenciais comprometidos.

Por mais que WAFs possam disponibilizar uma camada de segurança muitas vezes essencial para organizações que lidam com dados sensíveis. É preciso estabelecer uma arquitetura de software bem definida para poder distribuir essa funcionalidade de forma uniforme em organizações que proveem serviços de software na Internet.

Este projeto tem como objetivo desenvolver um sistema com o qual o desenvolvedor seja capaz de criar e configurar sua Web Application Firewall, removendo a necessidade de qualquer conhecimento a priori de arquiteturas internas de servidores, regras de segurança e suas sintaxes. Medidas de segurança de mitigação de ataques, como *SQL Injection* e *Cross-site Scripting* (XSS), foram padronizadas de forma que suas implementações são disponibilizadas de maneira acoplável à aplicação e essa é uma das vantagens que o sistema do projeto visa disponibilizar. É apresentada ao usuário apenas a informação de que a mitigação desses tipos de ataque existe e o mesmo é capaz de escolher a configuração que mais se adequa ao seu produto, definindo o equilíbrio desejado entre segurança e desempenho.

O projeto visa desenvolver um software que seja acessível a todos os tipos de desenvolvedores e não apenas aos profissionais de segurança. Além disso, o produto deve ser customizável e extensível para que possa evoluir conforme a modernização dos ataques de segurança.

A escassez de ofertas se dá pelo fato de que a disponibilização e configuração de servidores de WAF estão fortemente acoplados à infraestrutura dos serviços da organização, fazendo com que os provedores de infraestrutura na nuvem cobrem monetariamente por tais *features* de segurança como um adicional ao serviço já prestado. Porém, soluções *open source* de infraestrutura como o Kubernetes[2][3] vem dominando o mercado e devolvendo o poder de negociação às organizações, visto que quando se utilizam de tais ferramentas, resgatam o poder de escolha de alocação física de sua infraestrutura, seja ela *on-premises* ou utilizando algum provedor de *cloud*, podendo escolher aquele com o melhor custo-benefício para seus produtos.

O Kubernetes (comumente conhecido como k8s) é um sistema de orquestração de contêineres[4] open-source que automatiza a implantação, o dimensionamento e a gestão de aplicações em contêineres. Ele foi originalmente projetado pelo Google e agora é mantido pela Cloud Native Computing Foundation. Ele funciona com uma variedade de ferramentas de conteinerização, incluindo Docker[5].

Muitos serviços de nuvem oferecem uma plataforma baseada em Serviço (Paas ou Iaas), onde o Kubernetes pode ser implantado sob serviço gerenciado. Muitos fornecedores também provêm sua própria implementação de distribuição de Kubernetes.

Este projeto tem como desafio suprir a demanda de WAFs de código livre no mercado, com a proposta de que o produto final se torne uma opção tangível para que empresas e desenvolvedores possam proteger suas aplicações de forma simples, rápida e gratuita. Desenvolveremos esse serviço utilizando a plataforma de automação de desenvolvimento Kubernetes. Como objetivo secundário, o projeto visa explorar a flexibilidade para a disponibilização de softwares como serviço que essa plataforma provém.

2- Situação Atual

WAFs são implementadas como servidores de proxy reverso[6] que se estabelecem à frente de uma aplicação web. Um proxy reverso é um servidor de rede geralmente utilizado para ficar à frente de uma aplicação web. Todas as conexões originadas externamente são endereçadas para uma das instâncias da aplicação através de um roteamento feito pelo servidor proxy, que pode tratar ele mesmo a requisição (utilizando a estratégia de *caching*) ou encaminhar a requisição à aplicação.

Um proxy reverso repassa o tráfego de rede recebido para um conjunto de servidores, tornando-se a única interface para as requisições externas. Por exemplo, um proxy reverso pode ser usado para balancear a carga de um cluster de servidores Web. Assim, ele é exatamente o oposto de um proxy convencional, que age como um despachante para o tráfego de saída de uma rede, representando as requisições dos clientes internos perante os servidores externos à rede a qual o servidor proxy atende. A figura 1 visa exemplificar o funcionamento do tráfego de rede de um servidor de proxy reverso.

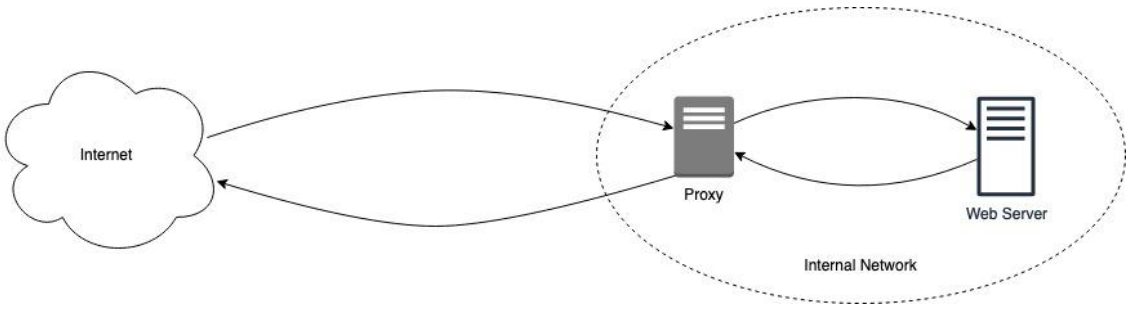


Figura 1: Tráfego de rede de um proxy reverso

As soluções de WAF atuais usualmente estão atreladas a uma infraestrutura específica (Amazon AWS e Google Cloud Engine) ou a linguagens e frameworks isolados. Uma das propostas do projeto é que o produto desenvolvido não esteja atrelado a qualquer tipo de linguagem, framework ou infraestrutura. O produto deve servir tanto para aplicações em *clouds* públicas como em clusters privados e ser completamente independente da linguagem utilizada na aplicação do desenvolvedor.

Além de um orquestrador de contêineres, o Kubernetes realiza o trabalho de abstrair os componentes de infraestrutura necessários para disponibilizar aplicações. Além disso, o Kubernetes permite que essas abstrações sejam expandidas por seus

usuários. Fizemos o uso dessas abstrações para tornar nosso produto agnóstico de qualquer infraestrutura. Ademais utilizamos expansões previamente criadas por usuários do Kubernetes para auxiliar na definição de novas abstrações, que definirão a composição de uma WAF em nosso sistema.

Existe uma relativa complexidade arquitetural para se estabelecer um ambiente de comunicação entre a aplicação e uma WAF associada a ela. Esta camada de complexidade é o que o projeto em questão visa erradicar. Durante o desenvolvimento são tomadas uma série de decisões de arquitetura e tecnologias (sempre *open source*) para que o fluxo de redirecionamento de tráfego da WAF para a aplicação aconteça. Os detalhes de arquitetura, assim como as decisões de tecnologias serão expostos a seguir.

3- Arquitetura do Sistema

O sistema possui duas arquiteturas complementares e é necessário que seu entendimento fique claro antes de se adentrar em mais detalhes sobre a implementação do código dos programas desenvolvidos e tecnologias utilizadas.

A primeira arquitetura é ilustrada na figura 2 que visa exemplificar o comportamento esperado da WAF produzida pelo software do projeto que irá interagir com a Internet e a aplicação pré-existente.

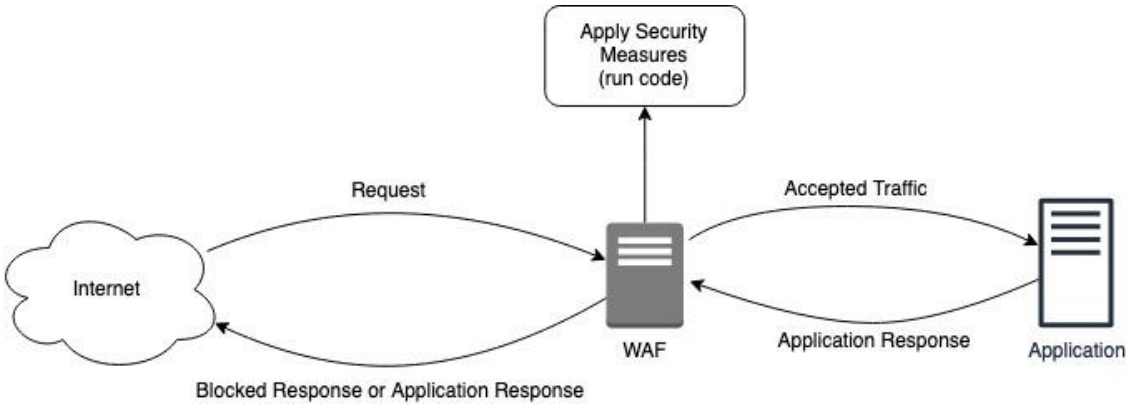


Figura 2: Comportamento da WAF disponibilizada pelo sistema

A segunda arquitetura, ilustrada na figura 3, descreve a utilização do sistema e trata de como os desenvolvedores de aplicações irão interagir com os softwares desenvolvidos no projeto.

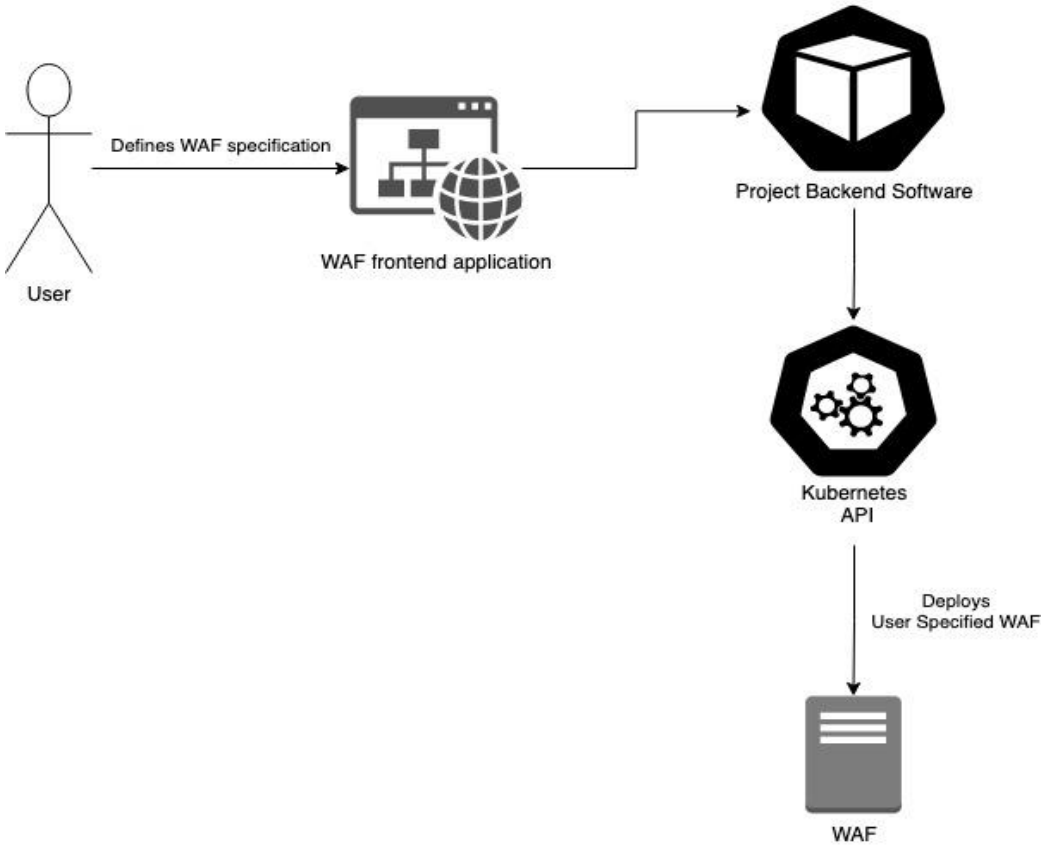


Figura 3: Interação dos componentes e usuários no sistema de forma simplificada

A figura 3 é uma simplificação da interação dos softwares desenvolvidos no projeto, e visa apenas exibir uma visão geral do todo, com foco no usuário. Adiante serão expostos diagramas que entrarão em detalhes específicos em relação a comunicação dos softwares em conjunto com os serviços do Kubernetes.

3.1- Tecnologias Utilizadas

O sistema foi desenvolvido utilizando o sistema operacional Linux, visto que todas as tecnologias de mercado a serem utilizadas têm suporte nativo ao mesmo. A aplicação é compatível com as principais distribuições Linux do mercado (Ubuntu, Mint, Arch, CentOS, etc...).

Os softwares a serem utilizados no trabalho serão:

1. Nginx Community, um servidor web *open source* escrito em C, que implementa o servidor de proxy reverso e que possibilita a execução de programas de terceiros que agregam funcionalidades ao servidor. Essas funcionalidades são denominadas módulos e as *features* de segurança do servidor são implementadas por meio da importação de

um desses módulos, chamado ModSecurity[7]. Módulos são bibliotecas, escritas em C, que devem ser compiladas em conjunto com o código fonte do servidor, fazendo com que seja necessário criar e manter uma distribuição do código do servidor com os módulos necessários pré-compilados.

Para realizar a tarefa de manter e distribuir de forma simples e fácil o código pré-compilado do servidor em conjunto com o módulo de segurança ModSecurity, escolhemos utilizar a tecnologia de conteinerização implementada pelo PaaS (Platform as a Service) Docker.

2. O Docker é uma alternativa de virtualização em que o kernel da máquina hospedeira é compartilhado com a máquina virtualizada ou o software em operação, permitindo que o desenvolvedor agregue a seu software bibliotecas e outras dependências do seu programa com menos perda de desempenho do que na virtualização do hardware (máquinas virtuais). O empacotamento e distribuição dos contêineres são feitos por meio de imagens Docker: um arquivo de configuração que determina a "receita" do sistema operacional e do software a ser executado dentro dele.

   Além do uso da tecnologia de conteinerização Docker a nível operacional, fez-se o uso da interface de versionamento de imagens chamada DockerHub, de modo a publicar, versionar e distribuir as imagens dos contêineres desenvolvidas no projeto para serem usadas de forma integrada com o Kubernetes, que faz o uso dessas imagens nativamente.

3. Como já mencionado anteriormente, utilizamos o Kubernetes, um sistema de orquestração de contêineres que automatiza a implantação, dimensionamento e a gestão de aplicações em contêineres. A decisão de utilizar o Kubernetes como orquestrador de contêineres, se justifica por ser a tecnologia de orquestração de contêineres mais bem estabelecida e difundida no mercado. Tomando como base a pesquisa realizada pela VMWare, VMWare State Of Kubernetes 2021, mais de 65% das empresas entrevistadas fazem uso de clusters Kubernetes em produção, em 2018 essa mesma pesquisa havia apontado uma

adesão de menos de 30% de clusters Kubernetes em produção. Com a grande adesão e crescimento desse software no ramo de orquestração de contêineres, encontramos consequentemente vasta documentação e suporte disponíveis.

Além de sua popularidade, o Kubernetes provê padrões de desenvolvimento e comportamento de software pré estabelecidos em sua arquitetura e que podem ser reutilizados para expandir suas funcionalidades. O projeto faz o uso de um desses padrões denominado *Operator*[8][9] (operador em português).

O software que desenvolveremos para a criação e configuração de WAFs possuem três componentes básicos, denominados: WAF Operator, WAF API e WAF Frontend. O WAF Operator é quem define e manipula as configurações de WAF no cluster, a WAF API possibilita configurar WAFs sem ter que nos comunicar diretamente com a API do Kubernetes, simplificando o processo, a WAF Frontend simplifica ainda mais o fluxo, disponibilizando uma interface gráfica para se comunicar com a WAF API. Esses três componentes se encaixam em duas categorias de software, a interface com a qual o usuário alvo irá interagir se configura na categoria de *frontend,* essa interface foi desenvolvida em Python utilizando o framework de aplicações web, Flask. Os outros dois componentes se encaixam na categoria de *backend* e foram desenvolvidos em Go (Golang). Esses componentes e sua arquitetura de comunicação serão discutidos a seguir.

Todos os componentes foram desenvolvidos de forma incremental. Utilizando o sistema de controle de versões Git, cada componente consiste em um diretório git único, um repositório, e todo o código escrito está disponibilizado na plataforma de hospedagem de código gratuita Github (www.github.com).

3.2- Detalhes da Arquitetura

A partir da explanação das tecnologias utilizadas pode-se entrar em detalhes mais concretos da arquitetura do sistema. Podemos comparar a figura 4 com a figura 2 exposta anteriormente e obter uma visão mais clara do uso de algumas das tecnologias mencionadas.

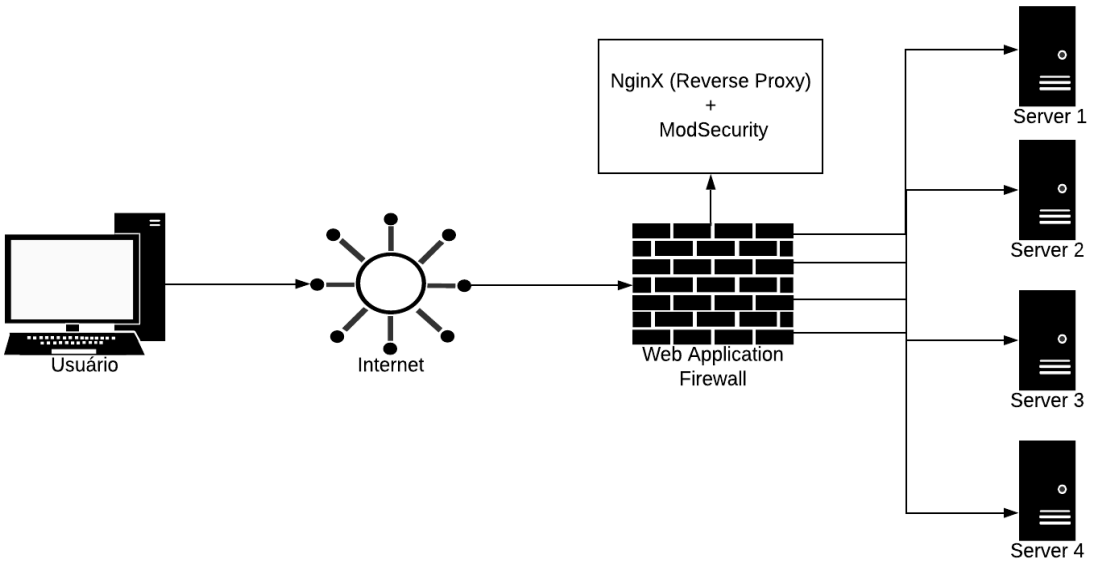


Figura 4: Comportamento da WAF com o uso das tecnologias mencionadas

Ainda sobre a figura 4, podemos observar que o balanceamento de tráfego feito pela WAF agora serve mais de uma instância da mesma aplicação. O Kubernetes nos permite escalar tanto a aplicação quanto às instâncias de WAF à sua frente.

O projeto consiste no desenvolvimento dos 3 componentes descritos brevemente anteriormente. São eles:

- WAF frontend

  A WAF Frontend é uma aplicação de *frontend* escrita na linguagem de programação python utilizando o framework Flask (um framework de desenvolvimento de aplicações web em python). Essa aplicação tem como objetivo apresentar todas as opções de configuração disponíveis relacionadas à WAF para o usuário. Uma vez que essas configurações forem definidas o usuário aperta o botão de "criar" sua instância no cluster e a responsabilidade do fluxo segue para a WAF API. A aplicação também dá a liberdade de realizar o *update* e o *delete* de instâncias previamente criadas.

  A figura 5 mostra a interface da WAF frontend exibida para o usuário.

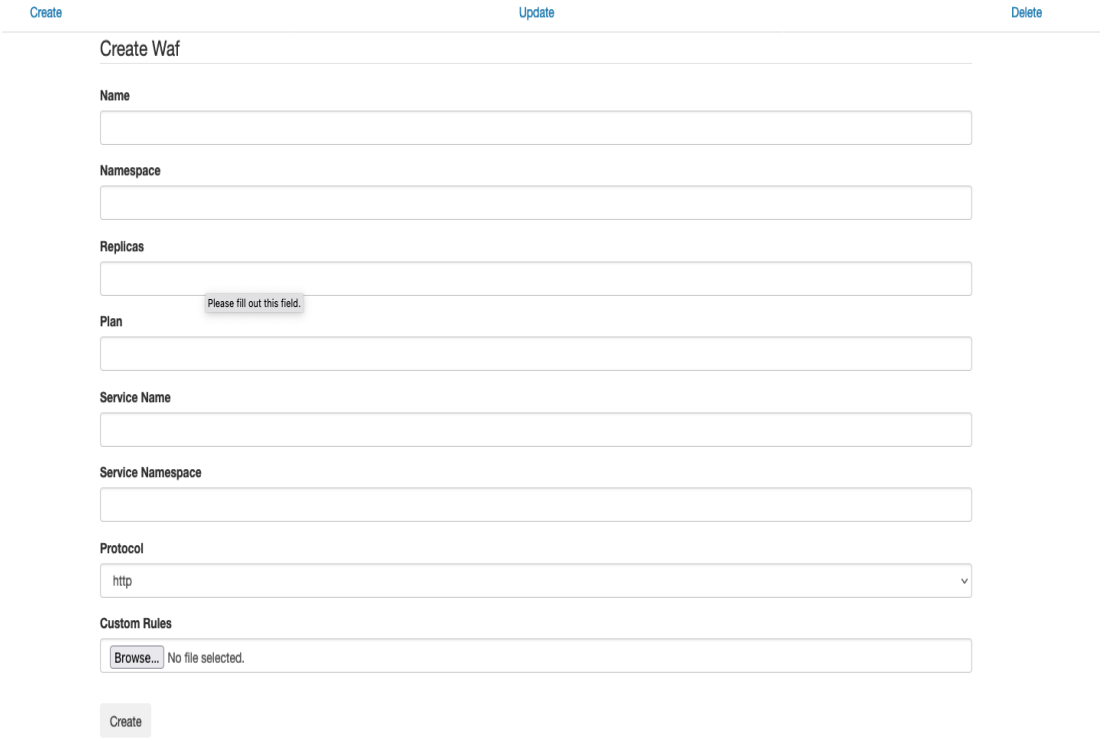


Figura 5: WAF frontend

Sobre os campos apresentados:

- Name: Nome da instância de WAF
- Namespace: *Namespace* é um conceito organizacional do Kubernetes que consiste na separação dos escopos dos objetos. Parafraseando a documentação oficial do Kubernetes, namespaces são clusters virtuais dentro de um cluster Kubernetes físico. Namespaces podem ser separados por times, por tipos de projeto, por tipos de aplicação... Cada time ou organização possui sua própria estrutura de namespaces e é necessário que o usuário especifique em qual namespace ele deseja criar sua instância de WAF. Caso ela não exista o programa retorna um erro.
- Replicas: A quantidade de instâncias redundantes que o usuário deseja que sua WAF possua, em ambientes produtivos é recomendado que se possua mais de uma réplica da sua aplicação ativa no cluster para que se minimize a possibilidade de qualquer indisponibilidade no serviço.
- Plan: O nome do plano de WAF existente no cluster, o sistema foi desenvolvido de forma que se possa criar diferentes planos de WAF.

Planos consistem em imagens Docker com Nginx Community e ModSecurity pré-compilados, e com regras de segurança pré-habilitadas para situações distintas de segurança i.e: no sistema atual é criado o plano *default* que busca cobrir as 10 falhas de segurança mais abundantes em aplicações web na internet. O sistema foi desenvolvido de forma que se possa criar planos com outros propósitos de segurança, um exemplo seria: um plano voltado apenas para ataque de DoS (Denial of Service) visto que a aplicação não possui um banco de dados por trás de sua arquitetura e só serve conteúdos estáticos… Cada plano visa englobar um cenário de segurança diferente.

- Service Name: Toda aplicação no Kubernetes é exposta dentro do cluster através de *services*. *Services* são balanceadores de requisições assim como a WAF mas de uso interno do Kubernetes. É por meio de *services* que as aplicações dentro de um cluster Kubernetes costumam se comunicar e esses *services* precisam de nomes no cluster.

  O *Service Name* requisitado no formulário se resume ao nome do *service* pelo qual a WAF criada irá repassar as requisições, ou seja, é o nome do *service* que fica à frente da aplicação a ser protegida.

- Service Namespace: além do nome do *service* é necessário que seja explicitado seu *namespace* é possível criar *services* com o mesmo nome em namespaces diferentes.
- Protocol: protocolo de camada 7 do modelo OSI utilizado na comunicação entra a WAF e a aplicação que ela protege, o usuário pode escolher entre HTTP e HTTPS.
- Custom Rules: Além das regras existentes no plano, o usuário pode acrescentar regras de segurança customizadas caso julgue necessário para sua aplicação. Basta criar um arquivo de extensão *.conf* contendo as regras que deseja acrescentar.
- Create: Uma vez que o usuário preencher todo o formulário com as especificações desejadas, a aplicação repassa a requisição para a API de backend WAF API, que recebe os dados em JSON (Javascript Object Notation) e tenta criar o objeto WAF no cluster.

- WAF API

Uma API REST que aceita requisições POST, PUT, DELETE da aplicação de *frontend*. A API é escrita em Go (golang) e possui o papel de se comunicar com a API do Kubernetes, determinando o estado desejado das instâncias de WAF que os usuários especificaram na WAF frontend. Ela recebe os dados em JSON e os transforma em objetos concretos do Kubernetes.

O Kubernetes funciona baseado em um conceito de estado desejado. Aplicações ou usuários que desejam criar, editar ou deletar objetos no Kubernetes atualizam o estado desejado no cluster por meio de uma API disponibilizada no momento da criação do cluster. A API por sua vez tem como objetivo atualizar esse estado, porém não é ela quem se encarrega de criar, editar ou deletar os objetos, nesse momento entram em cena os *Controllers* e *Operators* do Kubernetes.

- WAF Operator

  Como dito anteriormente, a responsabilidade de criar, editar e deletar os objetos do Kubernetes de acordo com o estado desejado se delega aos *Controllers* e *Operators* do Kubernetes. As diferenças entre os dois componentes citados entrarão em pauta mais adiante, mas o mais importante é que ambos os componentes são definidos por um *control loop* que observa o estado desejado do cluster e aplica mudanças, quando necessárias, visando chegar o mais próximo possível do estado desejado do cluster definido pela API do Kubernetes.

  Controllers e *Operators* são programas que rodam ininterruptamente dentro do cluster. Ambos são responsáveis por gerenciar um ou mais objetos (caso o objeto pelo qual são responsáveis possua dependências de outros objetos), objetos esses que sempre possuem um escopo bem delimitado.

  Agora vamos às diferenças, ou melhor à diferença, entre *Controllers* e *Operators. Controllers* gerenciam objetos previamente definidos dentro do cluster. No caso dos *services do Kubernetes* mencionados anteriormente, existe um *Controller* criado por *default* durante a criação do cluster que se encarrega dos estados desses objetos. Um *Operator* se encarrega das mesmas responsabilidades de um *Controller,* podemos e devemos dizer que um *Operator* é um *Controller* no Kubernetes. Sua única diferença é que enquanto o *Controller* gerencia objetos previamente definidos, um *Operator* se encarrega de criar as definições de novos objetos e gerenciá-los dentro do cluster, esses objetos novos podem e provavelmente serão, compostos por outros objetos previamente definidos no cluster.

  A partir dessa introdução de funcionalidades do padrão *Operator* conclui-se que o componente crucial que possibilita definir no cluster o que consiste uma WAF e gerencia todas as instâncias de WAF criadas no cluster é o Operator, denominado no projeto como WAF Operator. Esse componente de software realiza a ponte entre o conceito abstrato de WAF para um componente concreto no cluster. O WAF Operator foi desenvolvido em Go com a ajuda de um framework de criação de *Operators* chamado *kubebuilder*, desenvolvido por um dos grupos mantenedores do Kubernetes, denominado API Machinery.

A partir da descrição arquitetural dos componentes desenvolvidos podemos melhor exemplificar o funcionamento do sistema simplificado na figura 3, mostrada anteriormente na figura 6:

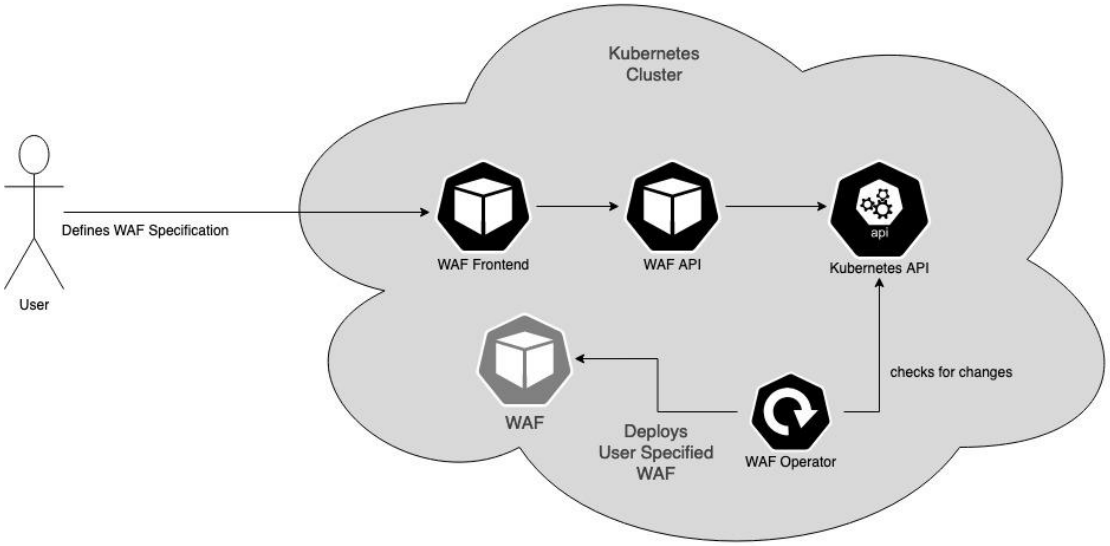


Figura 6: Arquitetura de comunicação dos componentes do sistema

4- Desenvolvimento

Cada componente foi desenvolvido seguindo uma ordem de prioridade, onde o componente mais crucial é implementado e testado primeiro para depois seguir para o desenvolvimento do próximo componente. Dessa forma, o primeiro componente desenvolvido foi o WAF Operator, seguido pela WAF API e finalmente, o WAF Frontend. Cada um possui um processo de desenvolvimento e desafios distintos, de forma que suas implementações tenham englobado diferentes áreas da ciência da computação. O desenvolvimento de cada um é exposto nos tópicos a seguir.

Caso o leitor queira acompanhar a descrição do desenvolvimento enquanto observa o código dos componentes. Os repositórios contendo o código de cada um dos componentes estão disponíveis na plataforma Github e seus *links* são expostos a seguir.

1. WAF Operator: https://github.com/arthurcgc/waf-operator
2. WAF API: https://github.com/arthurcgc/waf
3. WAF Frontend: https://github.com/arthurcgc/waf-frontend

4.1- WAF Operator

Como já introduzido anteriormente, o padrão *Operator* possibilita que se possa expandir os tipos de aplicações que podem ser executadas no Kubernetes. Podemos resumir seu propósito com a citação: “A Kubernetes Operator helps extend the types of applications that can run on Kubernetes by allowing developers to provide additional knowledge to applications that need to maintain state.” - Jonathan S. Katz, diretor de sucesso do consumidor & comunicações na empresa Crunchy Data.

A figura 7 busca exemplificar o funcionamento do *Operator* descrito anteriormente.

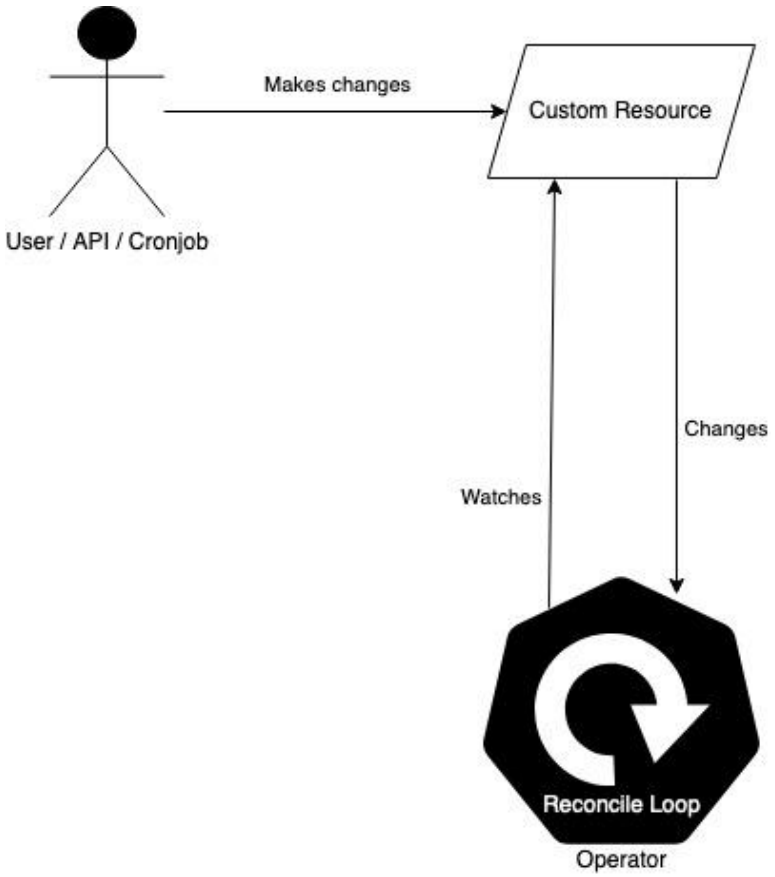


Figura 7: Padrão *Operator* no Kubernetes

O *Operator* irá definir o que compõe um objeto do tipo WAF e, através de seu loop de controle (*Reconcile Loop* na figura 7*)* manterá o estado de todas as instâncias o mais próximo possível de seu estado desejado no cluster.

A partir da definição de que será usado um *Operator* para representar as instâncias de WAF, é necessário definir como o *Operator* será criado. Atualmente para criar um *Operator* no Kubernetes é necessário declarar diversos arquivos de configuração para que o Kubernetes identifique o software como um. A tarefa de criar todos os arquivos necessários sem algum tipo de template para se basear seria demorada e muito suscetível a erro. Para evitar esse tipo de problema existem atualmente dois frameworks populares para a criação de *Operators*: operator-sdk desenvolvido pela equipe da Red Hat e o kubebuilder que é desenvolvido por um dos chamados Kubernetes-sigs (Kubernetes Special Interest Groups) o *API Machinery*. Optou-se por utilizar o kubebuilder, por ser um software *open source* e estar mais próximo, em termos organizacionais, ao desenvolvimento do próprio Kubernetes.

Os passos utilizados para se criar o WAF Operator utilizando o *kubebuilder* são análogos aos apresentados abaixo e serão discutidos brevemente.

```
mkdir -p ~/go/src/github.com/arthurcgc/waf-operator
cd ~/go/src/github.com/arthurcgc/waf-operator
kubebuilder init --domain waf.arthurcgc.waf-operator --repo
waf.arthurcgc/waf-operator
```

Cria-se o repositório do projeto e o inicializamos, criando os arquivos básicos que definem um *Operator.*

```
kubebuilder create api --group waf --version v1 --kind Waf
```

Esse passo cria os arquivos de declaração dos objetos customizados que o *Operator* instancia dentro do cluster. A seguir entraremos em mais detalhes sobre suas definições e como são compostas.

Uma vez definido como o *Operator* será criado é necessário definir o que compõe uma WAF, o *Custom Resource* da figura 7. Uma WAF se resume em um servidor de proxy reverso com uma lógica de detecção de ataques em seu código. Para representar o servidor de proxy reverso do projeto, utilizamos o software de código livre Nginx Community, uma solução muito utilizada pela comunidade Kubernetes, a ponto de fazer parte de exemplos da documentação oficial do mesmo. Além de fácil integração com o Kubernetes, ele possui um módulo desenvolvido pela equipe especializada em segurança, Spider Labs, chamado ModSecurity que se encarrega da execução de regras de segurança em nosso servidor de proxy reverso.

O projeto visa facilitar a vida do desenvolvedor de software de modo que este não precise ter conhecimento a priori das melhores práticas de segurança de rede para alcançar um nível de proteção de sua aplicação. Baseado nisso, tomou-se a decisão de disponibilizar por padrão, no momento de criação de uma nova WAF, um servidor que já possua proteção contra as 10 ameaças mais comuns às aplicações web da internet. Para isso tomou-se como base o projeto de segurança OWASP Top 10 [10], um projeto que pertence à *Open Web Application Security Project* (OWASP), que é caracterizada por uma comunidade disposta a criar e disponibilizar de forma gratuita artigos, metodologias, documentação, ferramentas e tecnologias no campo da segurança de aplicações web. O projeto OWASP Top 10 visa definir um consenso geral sobre as falhas de segurança mais críticas das aplicações web [11]. Convenientemente para o desenvolvimento deste projeto de WAF, a equipe Spider Labs trabalhou em conjunto com a OWASP para criar o OWASP ModSecurity Core Rule Set, um conjunto de regras genéricas de prevenção do OWASP Top 10 para o ModSecurity.

O OWASP ModSecurity Core Rule Set é a base utilizada para a criação de todas as WAFs do projeto, sempre que uma nova instância de WAF for criada no cluster ela é constituída de uma imagem Docker com Nginx Community em execução utilizando o módulo de ModSecurity com a lógica de regras de segurança, em conjunto com as regras pré estabelecidas pelo OWASP ModSecurity Core Rule Set. Dessa forma toda WAF criada possui proteção contra as 10 mais populares vulnerabilidades por padrão e o usuário tem o poder de estendê-las ou removê-las como quiser. A representação dessa imagem no Kubernetes foi chamada de Waf Plan e é declarada pelo *Operator*. Para criar tal plano, denominado como *default* no projeto basta especificar os campos necessários definidos pelo *Operator* no arquivo de configuração utilizado para declarar objetos no Kubernetes um *.yaml* como o a seguir:

```
apiVersion: waf.arthurcgc.waf-operator/v1
kind: WafPlan
metadata:
  name: default
  namespace: frontend
spec:
  image: arthurcgc/modsecurity
  description: "default waf plan"
  resources:
      limits:
```

```
      memory: "128Mi"
      cpu: "500m"
```

Os campos acima são definidos dentro do código do *Operator*, onde é preciso seguir uma hierarquia de arquivos. Em suma é declarado dentro do diretório raiz do projeto WAF Operator um diretório chamado *api* onde iremos declarar através do código, todos os objetos a serem definidos pelo *Operator.* O nome *api* se dá pelo fato de estar-se declarando novas especificações de objetos para a API do Kubernetes. O fragmento do código que declara o plano da WAF pode ser observado a seguir:

```
type WafPlanSpec struct {
   Image string `json:"image,omitempty"`

   Description string `json:"description,omitempty"`

   Resources corev1.ResourceRequirements `json:"resources,omitempty"`
}

type WafPlan struct {
   metav1.TypeMeta   `json:",inline"`
   metav1.ObjectMeta `json:"metadata,omitempty"`

   Spec WafPlanSpec `json:"spec,omitempty"`
}
```

No fragmento de código em Go acima declaramos duas *structs*. A estrutura WafPlan é composta pela *struct* WafPlanSpec, que representa os campos necessários para se declarar um plano. Os campos TypeMeta e ObjectMeta são necessários na definição de qualquer objeto do Kubernetes; neles declaramos o nome e o *namespace* do objeto.

Ainda sobre o fragmento de código do plano, pode-se observar que no campo *image* fazemos menção à uma imagem denominada *arthurcgc/modsecurity,* a imagem Docker previamente mencionada, mantida e tratada pelo autor do projeto. Dentro dessa imagem existem as regras pré-estabelecidas pela OWASP Top 10 e os softwares necessários para o funcionamento do servidor. Imagens Docker tem suas "receitas" definidas por Dockerfiles, arquivos que declaram o que deve ser executado dentro do container. O Dockerfile da imagem mencionada é relativamente simples:

```
FROM owasp/modsecurity:nginx

RUN apt-get update && apt-get install -y curl

# ModSecurity configuration files
```

```
ADD modsecurity /usr/local/waf-conf
```

Importamos a imagem base do OWASP Top 10, instalamos o *curl* (um software de requisições http(s) cuja necessidade será mencionada adiante) e adicionamos as configurações necessárias para que a biblioteca ModSecurity compile corretamente em conjunto com o software Nginx Community.

A especificação do objeto WAF dentro do cluster é feita de forma análoga à do plano explicitado anteriormente.

```
type Waf struct {
   metav1.TypeMeta   `json:",inline"`
   metav1.ObjectMeta `json:"metadata,omitempty"`

   Spec   WafSpec   `json:"spec,omitempty"`
   Status WafStatus `json:"status,omitempty"`
}
```

Definimos o esquema padrão de objeto Kubernetes. Parte da composição do objeto WAF está contida em WafSpec, onde definimos todos os campos pertinentes à composição de uma WAF em nosso cluster:

```
type WafSpec struct {

   Replicas *int32 `json:"replicas,omitempty"`

   WafPlanName string `json:"planName"`

   Rules Rules `json:"rules,omitempty"`

   Bind Bind `json:"bind,omitempty"`

   Service *nginxv1alpha1.NginxService `json:"service,omitempty"`
   ExtraFiles *nginxv1alpha1.FilesRef `json:"extraFiles,omitempty"`
}
```

Podemos observar que WafSpec é composta por outros objetos previamente declarados. O WafSpec é um objeto complexo que se adentrarmos em todos os seus campos explicitando suas informações fugiremos do escopo do relatório. O que vale aproveitar do fragmento de código acima é o entendimento de que ele define uma estrutura necessária para a declaração de uma WAF, onde podemos observar campos comuns aos definidos na interface WAF Frontend.

Uma vez declarado ao Kubernetes o que define um objeto WAF, deve-se implementar seu loop de controle, a interseção entre os padrões *Controller* e *Operator* previamente mencionada, que se encarrega de checar o estado dos objetos WAF

dentro do cluster. Assim como se declarou a definição dos objetos do cluster seguindo uma hierarquia onde eles eram definidos dentro da pasta *api,* deve-se declarar o código do controlador dentro da pasta *controllers*. O código do loop de controle se encontra dentro do arquivo *waf_controller.go.* Assim como Java, Go possui o conceito de interfaces. Para implementar um *Operator* em Go, precisamos declarar uma estrutura própria e implementar os métodos da interface *Controller* declarada no código do Kubernetes. Apenas dois métodos são declarados pela interface, são eles: Reconcile() (o loop de controle) e SetupWithManager() (a função que define como *Controller,* ou no caso o *Operator*, é inicializado).

A estrutura que define o *Operator* é definida no fragmento de código:

```
type WafReconciler struct {
   client.Client
   Log    logr.Logger
   Scheme *runtime.Scheme
}
```

Essa estrutura define um *client*, o cliente utilizado para se comunicar com a API do Kubernetes, uma implementação de logger que eventualmente exibe informações ao *Standard Output Stream* (stdout) e um *Scheme*, que define métodos de serialização para a API do Kubernetes e não é muito importante para o entendimento do funcionamento do projeto.

A implementação do loop de controle é exposta a seguir:

```
func (r *WafReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
   ctx := context.Background()
   _ = r.Log.WithValues("waf", req.NamespacedName)

   instance, err := r.getWafInstance(ctx, req.NamespacedName)
   if err != nil {
       return ctrl.Result{}, err
   }

   plan, err := r.getPlan(ctx, instance)
   if err != nil {
       return ctrl.Result{}, err
   }

   rendered, err := r.renderTemplate(ctx, instance, plan)
   if err != nil {
       return reconcile.Result{}, err
   }

   wafCM, err := newWafConfig(instance)
   if err != nil {
```

```
        return reconcile.Result{}, err
    }
    err = r.reconcileConfigMap(ctx, wafCM)
    if err != nil {
        return reconcile.Result{}, err
    }

    mainCM := newMainCM(instance, rendered, wafCM)
    err = r.reconcileConfigMap(ctx, mainCM)
    if err != nil {
        return reconcile.Result{}, err
    }

    nginx := newNginx(instance, plan, mainCM)
    if err = r.reconcileNginx(ctx, instance, nginx); err != nil {
        return ctrl.Result{}, err
    }

    if err = r.refreshStatus(ctx, instance, nginx); err != nil {
        return ctrl.Result{}, err
    }

    return ctrl.Result{}, nil
}
```

O código dentro da função Reconcile é executado ininterruptamente dentro do cluster, e possui o seguinte comportamento: sempre é criado um objeto novo dentro do código nas funções: newWafConfig(), newMainCM() e newNginx(). Essas funções sempre criam novos objetos independentemente do fato deles terem sido alterados ou não. Quando chamamos as funções reconcileConfigMap() e reconcileNginx() esses objetos são enviados ao *client* do Kubernetes contido na estrutura WafReconciler e o *client* por sua vez realiza as atualizações aos objetos caso sejam necessárias.

Um último, porém importante, ponto sobre a implementação do WAF Operator é que ele faz uso da funcionalidade de outros *Operators* que precisam estar presentes no cluster para manter o estado de seus objetos equivalentes ao desejado. O objeto WAF é composto por diversos objetos nativos do Kubernetes e todos esses objetos que o compõem já possuem seus *Operators* criados por padrão no momento de criação de qualquer cluster Kubernetes. Porém, existe uma exceção que é o objeto Nginx. Como exposto anteriormente, a WAF do projeto consiste em um servidor de proxy reverso Nginx em conjunto com o módulo ModSecurity. O *Operator* desenvolvido no trabalho tem como responsabilidade realizar o controle das regras de segurança do servidor. Para realizar a declaração e o controle do servidor Nginx faz-se o uso de um outro *Operator* desenvolvido pelo grupo Tsuru, o Nginx Operator,

que define e mantém os estados desejados desses objetos no cluster. Pode-se interpretar o *Operator* desenvolvido neste projeto como uma expansão das funcionalidades do Nginx Operator. Nosso WAF Operator irá configurar o objeto Nginx definido pelo Nginx Operator para que se comporte como um proxy reverso com lógicas de segurança internas. A figura 8 visa mostrar essa relação entre os objetos e como eles são declarados por seus respectivos *Operators* dentro do cluster.

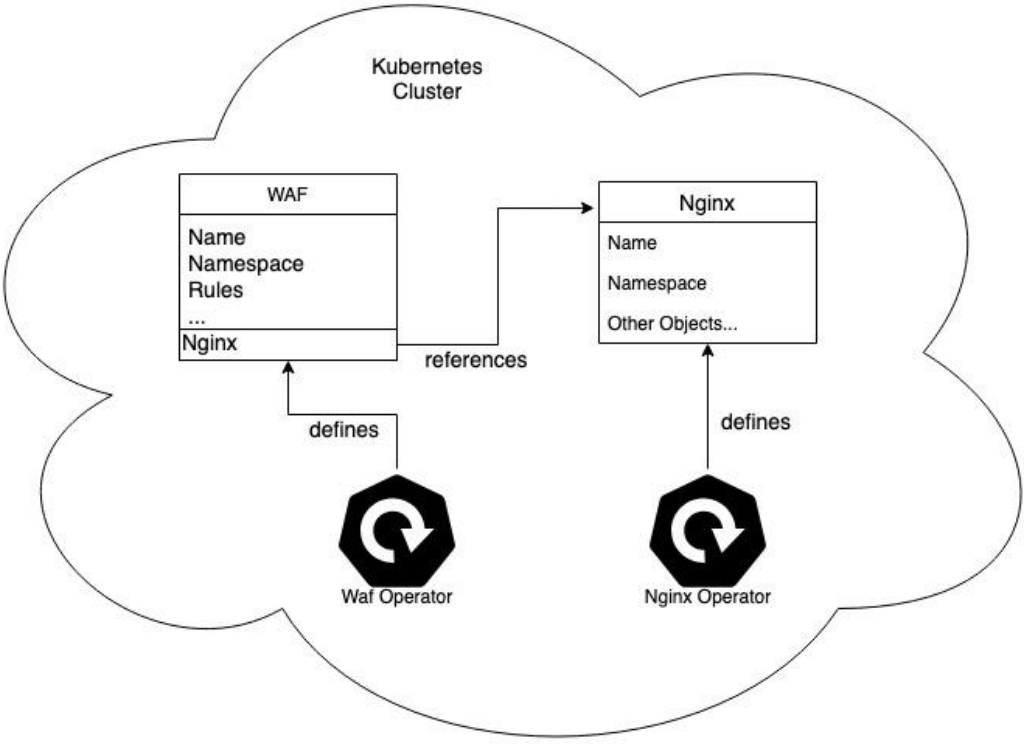


Figura 8: Relacionamento entre Waf Operator e Nginx Operator

A decisão de fazer o uso de dois *Operators* é embasada no princípio de decomposição de problemas presentes no pensamento computacional. É preciso realizar o controle de instâncias de Nginx assim como das regras de WAF e realizar todo esse trabalho em um componente só acabaria criando um monólito (software de extrema complexidade e difícil manutenibilidade) sem um propósito bem definido. Baseando-se nesse conceito, tomou-se a decisão de fazer o uso de dois *Operators* no cluster para que o papel de cada módulo ficasse bem definido: um *Operator* é encarregado de administrar todas as instâncias de Nginx presentes no cluster, o outro, desenvolvido neste projeto, é encarregado de administrar toda a configuração e regras presentes no módulo de ModSecurity. Isso faz com que se separe a

responsabilidade de cada software e que se possa tratá-los de formas diferentes e isoladas. Essa separação também permitiu isolar completamente os módulos desenvolvidos neste projeto de módulos já existentes.

Para disponibilizar o WAF Operator dentro do cluster foi criado uma receita em um arquivo *Makefile*. Essa receita automatiza os comandos e processos que devem ser realizados: ela cria as definições de WAF e WafPlan declaradas na pasta *api,* instancia as permissões necessárias para o *Operator* criar, editar e deletar objetos do tipo WAF e todos seus subcomponentes, realiza o *build* do código fonte, e, finalmente realiza o *deploy* (evento de criação de objetos no Kubernetes) do *Operator* no cluster. O trecho do *Makefile* que contém essas instruções pode ser visualizado abaixo:

```
all: manager

test: generate fmt vet manifests
	go test ./... -coverprofile cover.out

manager: generate fmt vet
	go build -o bin/manager main.go

run: manager manifests
	./bin/manager

install: manifests
	kustomize build config/crd | kubectl apply -f -

uninstall: manifests
	kustomize build config/crd | kubectl delete -f -

deploy: manifests
	cd config/manager && kustomize edit set image controller=${IMG}
	kustomize build config/default | kubectl apply -f -

manifests: controller-gen
	$(CONTROLLER_GEN) $(CRD_OPTIONS) rbac:roleName=manager-role webhook
paths="./..." output:crd:artifacts:config=config/crd/bases

fmt:
	go fmt ./...

vet:
	go vet ./...

generate: controller-gen
	$(CONTROLLER_GEN) object:headerFile="hack/boilerplate.go.txt" paths="./..."
```

```
controller-gen:
ifeq (, $(shell which controller-gen))
```

```
	@{ \
	set -e ;\
	CONTROLLER_GEN_TMP_DIR=$$(mktemp -d) ;\
	cd $$CONTROLLER_GEN_TMP_DIR ;\
	go mod init tmp ;\
	go get sigs.k8s.io/controller-tools/cmd/controller-gen@v0.2.5 ;\
	rm -rf $$CONTROLLER_GEN_TMP_DIR ;\
	}
CONTROLLER_GEN=$(GOBIN)/controller-gen
else
CONTROLLER_GEN=$(shell which controller-gen)
endif
```

A receita de *Makefile* é gerada através do framework de criação de *Operators* mencionado anteriormente, o *kubebuilder.*

Depois de criar o novo *Operator,* podemos visualizar seu funcionamento dentro do cluster ao tentar criar um objeto do tipo WAF, declarando suas especificações em um arquivo *yaml.*

```
apiVersion: waf.arthurcgc.waf-operator/v1
kind: Waf
metadata:
 name: tcc-sample
 namespace: frontend
spec:
 replicas: 1
 planName: default
 bind:
   name: unsafe-application
   hostname: http://unsafe-application-service.backend.svc.cluster.local
 service:
   type: NodePort
```

Executando o comando via *kubectl:*

```
> kubectl apply -f config/samples/sample-tcc.yaml
waf.waf.arthurcgc.waf-operator/waf-sample created
```

Podemos observar que a instância de WAF foi criada pelo *output* do comando e observando os logs do *Operator:*

```
> kubectl logs waf-operator-controller-manager-7b68cbb9b-kvp8l \
-n waf-operator-system -c manager
2021-06-11T18:16:00.740Z	DEBUG controller	Successfully
Reconciled	{"reconcilerGroup": "waf.arthurcgc.waf-operator",
"reconcilerKind": "Waf", "controller": "waf", "name": "tcc-sample",
"namespace": "frontend"}
2021-06-11T18:16:02.353Z	DEBUG controller	Successfully
```

```
Reconciled  {"reconcilerGroup": "waf.arthurcgc.waf-operator",
"reconcilerKind": "Waf", "controller": "waf", "name": "tcc-sample",
"namespace": "frontend"}
2021-06-11T18:16:04.446Z      DEBUG controller  Successfully
Reconciled  {"reconcilerGroup": "waf.arthurcgc.waf-operator",
"reconcilerKind": "Waf", "controller": "waf", "name": "tcc-sample",
"namespace": "frontend"}
```

4.2- WAF API

Uma vez que foi desenvolvido o *Operator* apresentado na seção anterior, seguimos para a implementação da API proposta no projeto. A WAF API tem como objetivo se comunicar diretamente com o Kubernetes através de sua própria API. Isso ressalta o fato de que a WAF API em nenhum momento se comunica diretamente com o WAF Operator. O papel da WAF API consiste em disponibilizar uma interface própria de comunicação do usuário (mesmo que indiretamente) com o Kubernetes, para que ele não precise declarar arquivos de configuração e utilizar ferramentas como o *kubectl* para criar suas WAFs. Podemos nos remeter à figura 6 para relembrar seu fluxo de comunicação com o sistema.

A API foi desenvolvida em Go pelo fato de existirem bibliotecas em Go criadas pela própria equipe de desenvolvimento do Kubernetes para facilitar a comunicação com sua API.

Seu desenvolvimento se revela mais simples quando comparado com todo o trabalho realizado no WAF Operator, algo natural pois a lógica do sistema está presente nele. Como já mencionado, o Kubernetes possui uma API que nos permite realizar operações que manipulam os objetos do cluster. A implementação da WAF API consiste em definir requisições que abstraem a manipulação direta dos objetos do tipo WAF na API do Kubernetes. Em seu código definimos essas requisições e expomos o mínimo possível de especificações intrínsecas ao Kubernetes, com o objetivo de manter o foco apenas na configuração da WAF. Mais precisamente, implementamos requisições de criação, edição e deleção de objetos WAF e dentro do código nos comunicamos com a API do Kubernetes para realizar o objetivo dessas requisições.

O progresso do desenvolvimento assim como trechos de código pertinentes ao comportamento do componente estão presentes no Apêndice A.

4.3- WAF Frontend

Para simplificar a criação de WAFs introduzida pela WAF API, foi desenvolvida uma interface web para que o usuário possa realizar as requisições de criação, edição e deleção de forma mais simples e rápida. A aplicação de frontend foi escrita em Python, em conjunto com o framework Flask e utilizando recursos da biblioteca em Javascript Bootstrap. Ela consiste em três páginas de formulários que representam as operações disponíveis na WAF API: criar, atualizar e deletar uma WAF.

Sua interface de criação já foi apresentada na figura 5. Apresentamos nas figuras 9 e 10 a interface gráfica das operações remanescentes: *update* e *delete*. O processo de desenvolvimento e criação da WAF Frontend no cluster está presente no Apêndice B do documento.

Figura 9: WAF Frontend - operação de update

Figura 10: WAF Frontend - operação de delete

5- Resultados Obtidos

Com a demonstração de todos os componentes e a explicitação de seu desenvolvimento, resta testar se o produto desenvolvido de fato cumpre seu propósito; proteger uma aplicação vulnerável de ataques cibernéticos.

Nossa metodologia de testes para confirmar a proteção disponibilizada pela WAF criada consiste em: criar uma aplicação originalmente vulnerável, comprovar sua vulnerabilidade tentando atacá-la e obtendo dados sigilosos, conectar a aplicação à WAF utilizando o fluxo do sistema, atacar novamente a aplicação e checar se a vulnerabilidade persiste. O sistema se comprova bem sucedido se o segundo ataque, cujo fluxo de dados passa pela WAF, é identificado e interceptado antes de atingir a aplicação.

Para a representação de uma aplicação verdadeiramente insegura, utilizamos o projeto DVWA[1] (Damn Vulnerable Web Application), uma aplicação web em PHP que utiliza um banco de dados MySQL, desenvolvida com o propósito de ser extremamente vulnerável aos mais variados tipos de ataques. Ela é disponibilizada como uma imagem Docker que podemos utilizar para instalá-la em nosso cluster e realizar nossos testes.

Para acessarmos a aplicação presente em nosso cluster local, utilizamos a técnica de *port forward* para expor o endereço da aplicação à rede local do computador de teste com acesso ao cluster. O Kubernetes provê uma forma fácil de se realizar o *port forward* de *services* utilizando o *kubectl*:

```
kubectl port-forward svc/dvwa-service 80:7777 -n backend
```

Uma vez realizado o *port forward* do *service* com a interface de rede do computador, podemos acessar a aplicação no endereço especificado: *http://localhost:7777*.

Primeiramente faremos um teste manual de um ataque de SQL injection na aplicação insegura, comprovando que conseguimos acesso aos dados sigilosos.

[1] Disponível em: <https://dvwa.co.uk/>
Acesso em: 16 de junho de 2021

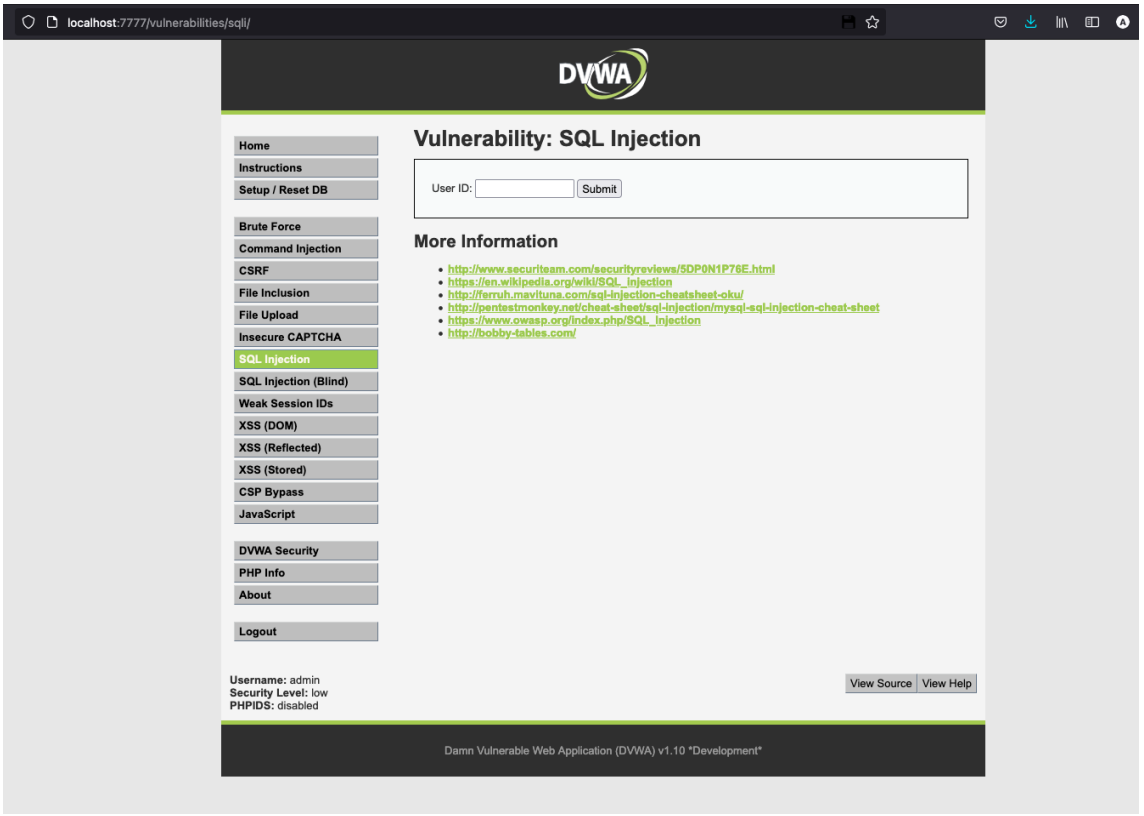


Figura 11: DVWA SQL Injection

Passamos a seguinte query ao campo de User ID exposto: "%' and 1=0 union select null, concat(user,':',password) from users #", obtendo o nome dos usuários no banco e suas respectivas senhas como podemos observar na figura 12.

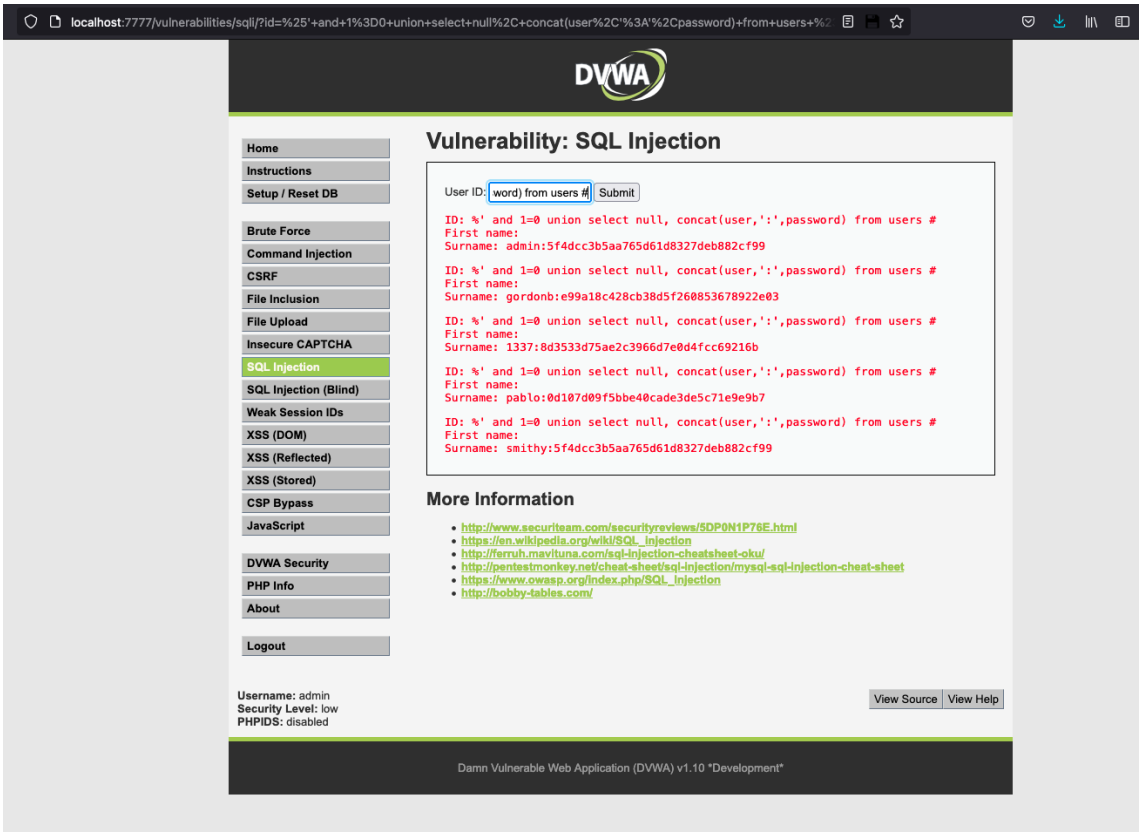


Figura 12: DVWA SQL Injection resultado

Podemos criar uma WAF através do sistema desenvolvido para proteger a aplicação e evitar esse tipo de ataque. Criamos a WAF a partir da WAF Frontend. A figura 13 ilustra essa criação.

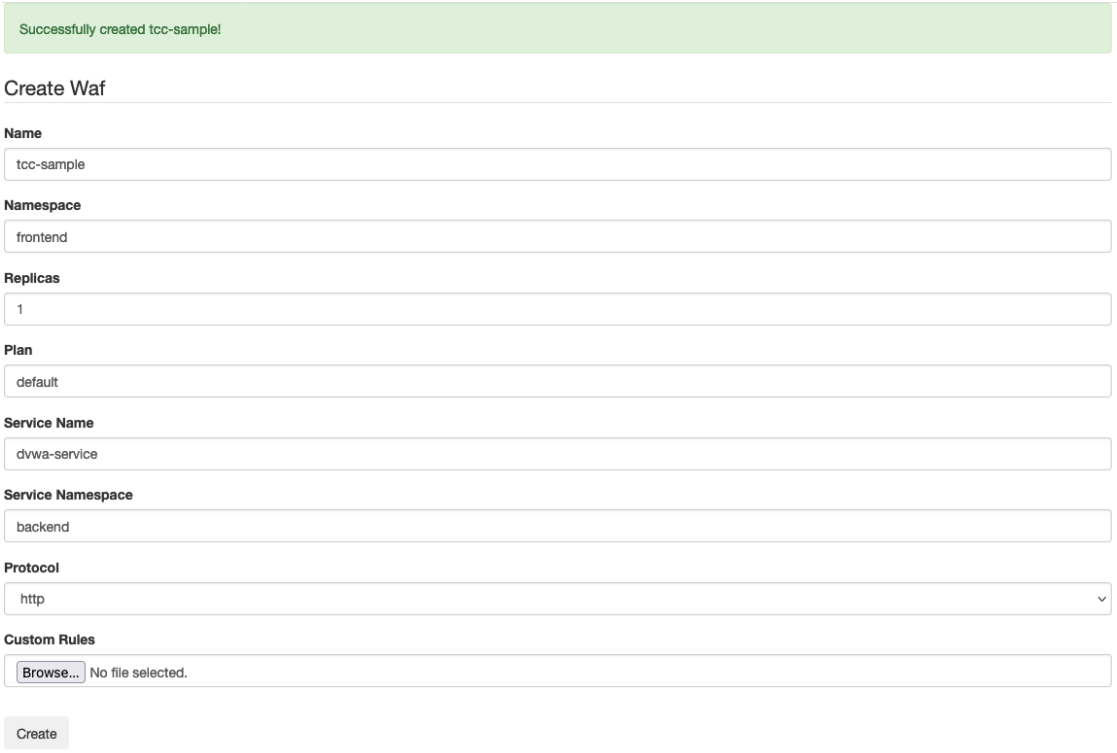


Figura 13: Criação da WAF de teste

É importante observar que criamos nossa aplicação insegura (DVWA) no cluster com um *service* de nome: *dvwa-service* e no namespace: *backend*. A receita denominada *dvwa.yaml* de configuração da aplicação e o comando de criação são apresentados a seguir.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: backend
  name: dvwa
spec:
  selector:
      matchLabels:
      app: dvwa
  template:
      metadata:
      labels:
      app: dvwa
      spec:
      containers:
      - name: dvwa
      image: vulnerables/web-dvwa
      resources:
```

```
            limits:
            memory: "128Mi"
            cpu: "500m"
      ports:
      - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  namespace: backend
  name: dvwa-service
spec:
  selector:
      app: dvwa
  ports:
  - port: 80
      targetPort: 80
```

Comando de criação:

```
kubectl apply -f dvwa.yaml
```

Caso a aplicação não tivesse sido criada a operação de criação da WAF retornaria um erro.

Uma vez criada a WAF podemos testar sua efetividade realizando a mesma técnica de *port forward* explicitada anteriormente, dessa vez iremos expor o *service* do nosso objeto WAF que irá repassar todas as requisições à aplicação insegura. Podemos e devemos remover o *port forward* feito anteriormente para o *service* da aplicação, fazendo com que a única forma de se acessá-la seja através de nossa WAF, simulando um ambiente de produção. Escolhemos por expor a WAF no endereço: *http://localhost:9999* e observamos que ela redireciona o tráfego com sucesso à aplicação.

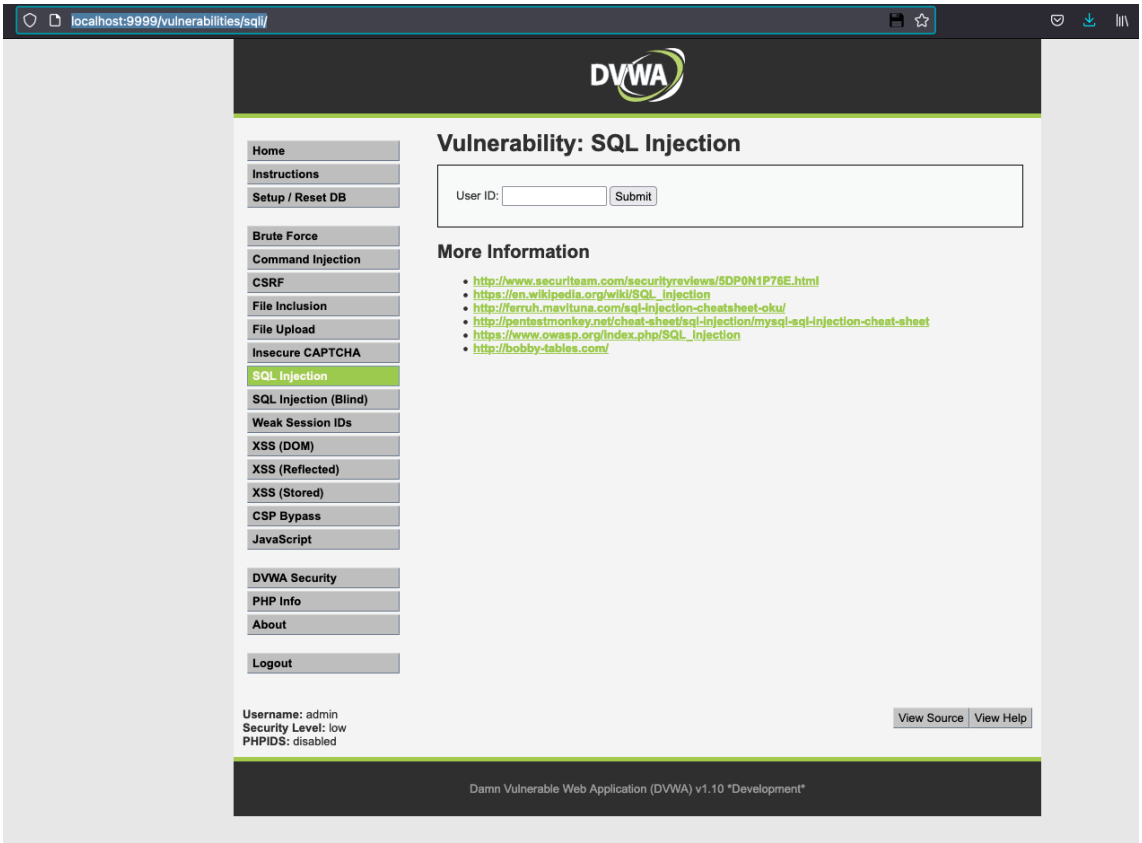


Figura 14: Acessando a aplicação através da WAF

O próximo passo é reproduzir o mesmo ataque de SQL Injection exposto anteriormente e observar seu resultado, explicitado na figura 15.

Figura 15: WAF retorna 403 Forbidden quando tentamos realizar um SQL Injection

Ao tentar realizar o ataque, somos retornados uma página com o código http 403 *Forbidden*, a WAF criada identificou o ataque e interrompeu o tráfego da requisição até a aplicação.

Para melhor exemplificar a capacidade de customização e expor a facilidade que o sistema provê na criação de WAFs, podemos remover a regra de SQL injection através da interface web e realizar a narrativa de ataque novamente. Observando que, a partir da remoção das regras relacionadas a SQL injection, o sistema deve retornar a ser vulnerável a esse tipo de ataque. Pode ser interessante para o desenvolvedor manter apenas regras de mitigação relacionadas a tecnologias utilizadas em sua aplicação, com o intuito de manter a simplicidade e desempenho de seu sistema.

Realizamos um *update* na WAF criada anteriormente, removendo a regra de SQL Injection presente no menu de regras da interface web, explicitado na figura 16.

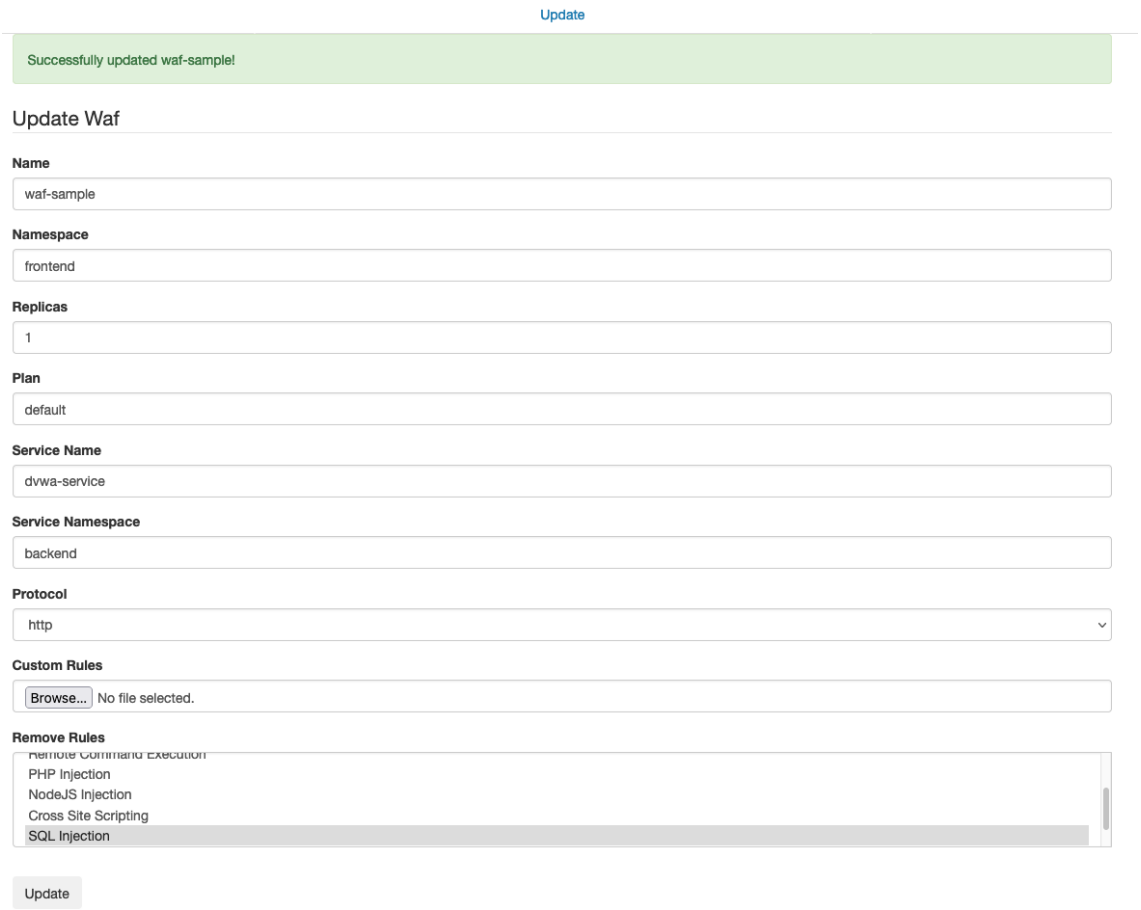


Figura 16: Update da WAF, retirando a proteção a ataques de SQL Injection

Agora devemos realizar a narrativa de ataque de SQL injection novamente e comprovar que a WAF atualizada já não possui mais proteção à esse tipo de ataque. O resultado é exibido na figura 17:

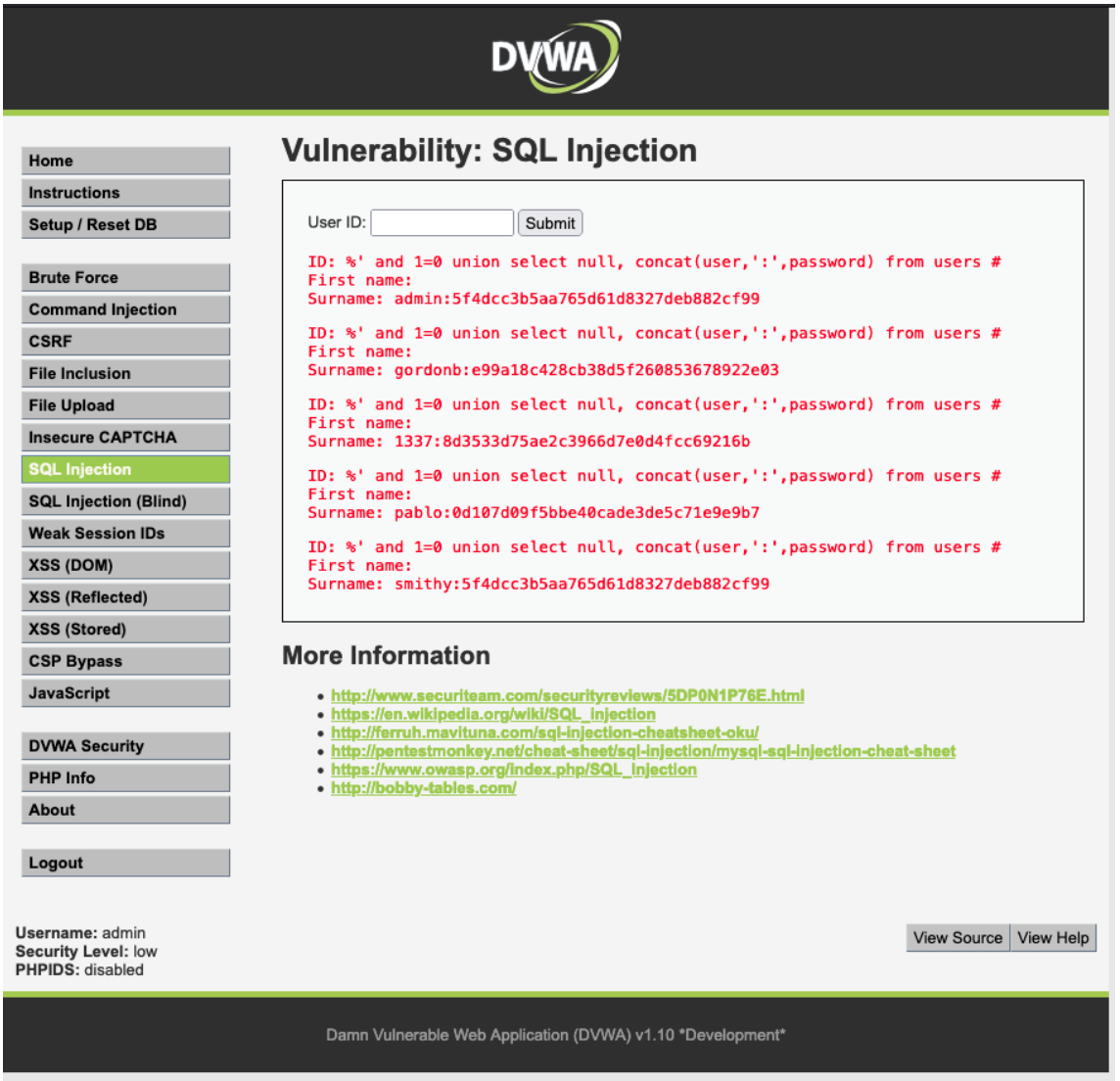


Figura 17: Resultado da remoção de regras de SQL Injection através da interface web

Uma vez testada a efetividade da WAF criada manualmente para a regra de SQL Injection, utilizamos a ferramenta automatizada de análise de segurança *nikto*[2], para que possamos obter um resultado de teste mais contundente. Essa ferramenta irá gerar uma série de testes do tipo caixa preta na *URL* especificada, assim como um relatório de segurança dos resultados obtidos.

Comparamos os resultados da análise utilizando a aplicação sem a proteção da WAF, e, em seguida, com a proteção da WAF à frente das requisições. Para isso, realizamos a técnica de *port forward* do serviço da aplicação mapeando-a sem WAF para: http://localhost:7777 e para a WAF mantemos o *port forward* de

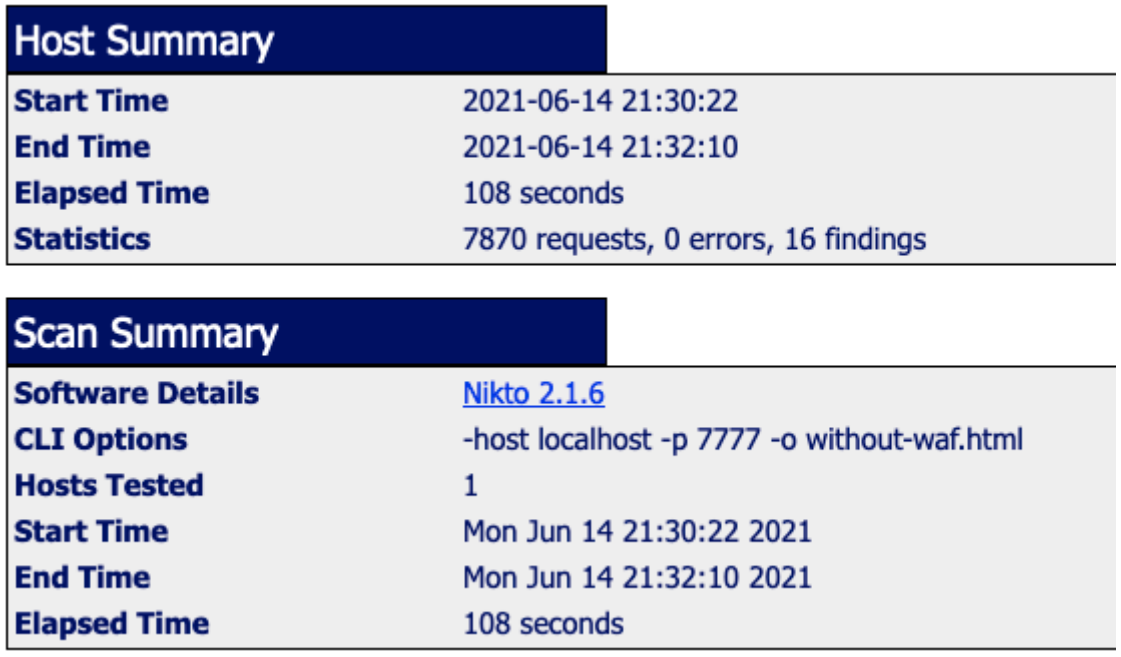
[2] Disponível em: <https://github.com/sullo/nikto>
Acesso em: 18 de Junho, 2021

http://localhost:9999. Os comandos utilizados para realização dos testes são expostos a seguir:

```
nikto -host localhost -p 7777 -o without-waf.html
```

```
nikto -host localhost -p 9999 -o with-waf.html
```

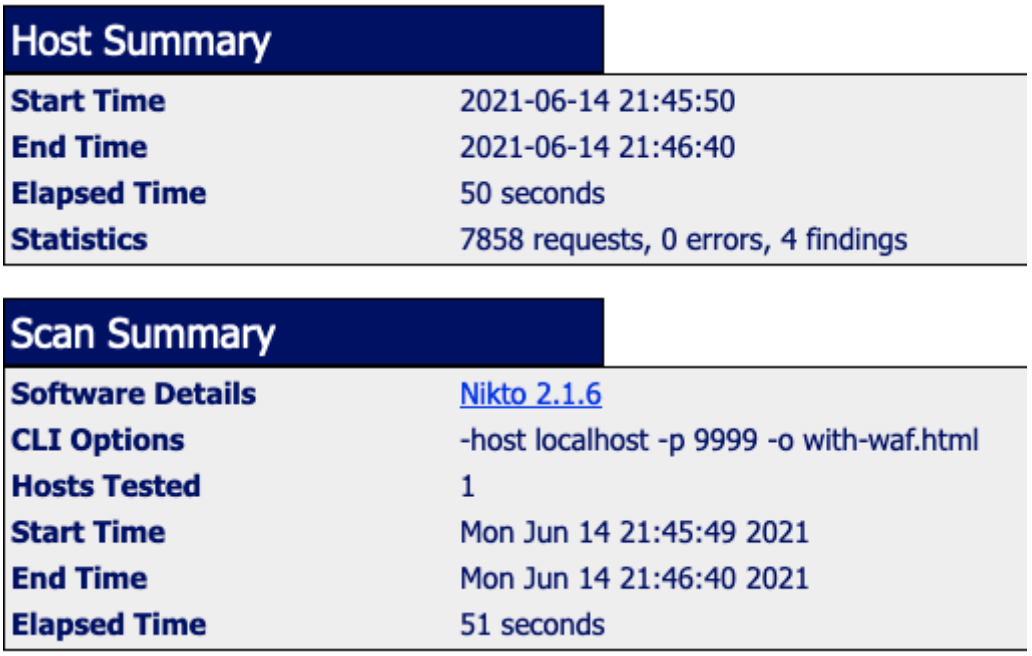
Seus resultados da varredura de segurança sem a proteção da WAF e com a proteção da WAF estão presentes nas figuras 18 e 19 respectivamente:



Host Summary	
Start Time	2021-06-14 21:30:22
End Time	2021-06-14 21:32:10
Elapsed Time	108 seconds
Statistics	7870 requests, 0 errors, 16 findings

Scan Summary	
Software Details	Nikto 2.1.6
CLI Options	-host localhost -p 7777 -o without-waf.html
Hosts Tested	1
Start Time	Mon Jun 14 21:30:22 2021
End Time	Mon Jun 14 21:32:10 2021
Elapsed Time	108 seconds

Figura 18: Resultados da análise da aplicação sem proteção da WAF



Host Summary	
Start Time	2021-06-14 21:45:50
End Time	2021-06-14 21:46:40
Elapsed Time	50 seconds
Statistics	7858 requests, 0 errors, 4 findings

Scan Summary	
Software Details	Nikto 2.1.6
CLI Options	-host localhost -p 9999 -o with-waf.html
Hosts Tested	1
Start Time	Mon Jun 14 21:45:49 2021
End Time	Mon Jun 14 21:46:40 2021
Elapsed Time	51 seconds

Figura 19: Resultados da análise da aplicação com proteção da WAF

Como esperado, foram encontradas mais falhas de segurança na aplicação sem a proteção da WAF. Além disso, as 4 vulnerabilidades encontradas na varredura com a proteção da WAF são falhas que, caso fossem exploradas de fato, a instância de WAF as interceptaria, tornando as 4 vulnerabilidades mencionadas pelo software casos de falsos positivos.

5.1- Desempenho

Apesar dos resultados mostrados até o momento refletirem as vantagens da utilização de WAFs, devemos levar em conta o inevitável impacto computacional que elas adicionam às aplicações. Por ser necessária a inspeção de todos os pacotes *http* que trafegam pela aplicação, a WAF acaba adicionando um *overhead* de latência considerável. Realizamos testes de carga na aplicação para exemplificar uma amostra do impacto gerado, que pode ser crucial para aplicações que necessitam manter um tempo de resposta muito baixo.

Os testes de carga foram realizados utilizando a ferramenta *k6* que provê uma sdk completa para testes de cargas de aplicações. Especificamos os testes para realizar a requisição da página principal da aplicação, passando um parâmetro denominado *testparam* com o valor de "*thisisaloongstringthatneedstobechecked*" apenas para quando utilizarmos a WAF ela tenha que inspecionar este parâmetro além dos já existentes no request da página, adicionando mais trabalho

computacional propositalmente. O teste de carga realizado na aplicação sem passar pela waf tem seu resultado exposto na figura 20.

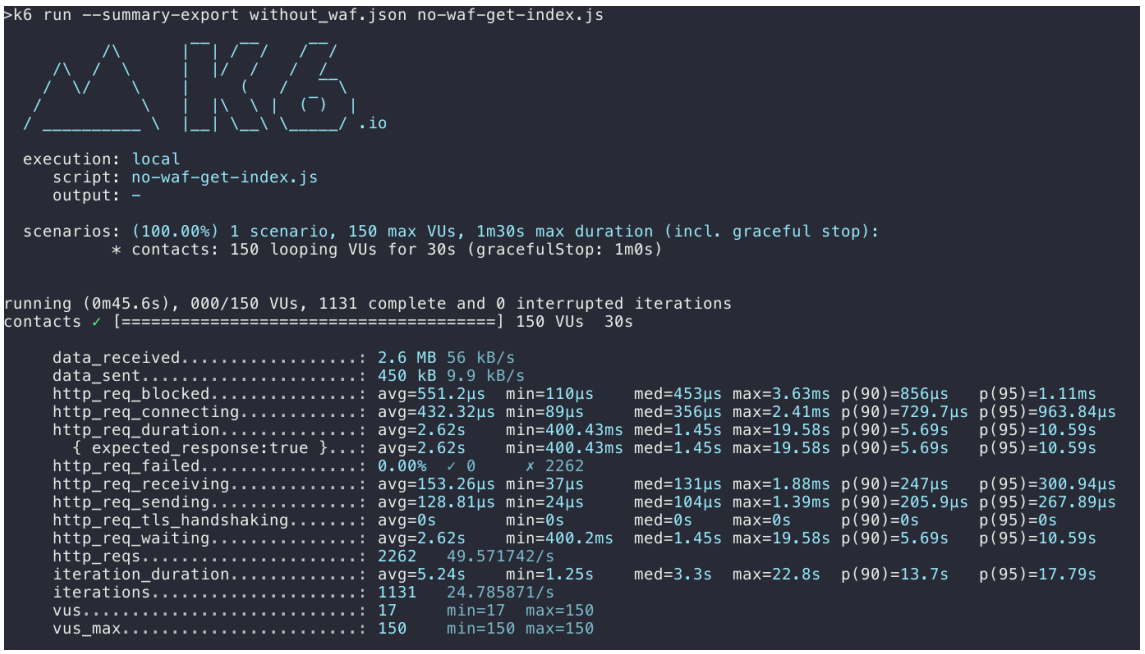


Figura 20: Resultados do teste de carga da aplicação sem a WAF

Simulamos o acesso simultâneo de 150 usuários por 30 segundos. O dado mais importante para essa demonstração, no momento, é o http_req_duration, que detém os dados relacionados à duração dos requests *http* feitos. Podemos observar que obtemos uma média de 2.62s de tempo de resposta da aplicação. Comparando com os resultados obtidos da figura 21:

Figura 21: Resultados do teste de carga da aplicação passando pela WAF

Como esperado, a latência medida por http_req_duration se revela maior quando utilizamos a WAF. Tivemos um acréscimo de quase 50% de *latência* se comparado com o teste anterior. É necessário considerar os *requests* que estão sendo feitos não possuem muita complexidade em seu corpo e o conteúdo sendo servido pela aplicação não representa um conteúdo de dados dinâmico, que requer a comunicação com outros serviços para serem retornados. Fazendo com que o impacto da WAF, ainda que notável, não represente um limite de impacto de latência único. O impacto vai depender das funcionalidades que cada aplicação implementa, exigindo um estudo de desempenho da WAF nas aplicações que pretendem manter um tempo de resposta reduzido para seus usuários ou serviços que a utilizem.

6- Considerações Finais

Neste trabalho, produzimos três componentes que juntos, representam um sistema de criação de WAFs seguindo a arquitetura de microsserviços, utilizando o Kubernetes como plataforma de disponibilização dos microsserviços desenvolvidos. A escolha de tal arquitetura se justifica pela forma como foi planejado que o sistema deveria se comportar, disponibilizando um software como serviço (SaaS) a seus usuários. O projeto faz uso de tecnologias exclusivamente *open source* assim como seu próprio código e tem como objetivo contribuir para tal comunidade disponibilizando uma alternativa gratuita e facilmente acoplável a qualquer ambiente

de desenvolvimento que faça uso do Kubernetes. Os repositórios dos componentes desenvolvidos estão disponíveis em: https://github.com/arthurcgc/waf-operator, https://github.com/arthurcgc/waf, https://github.com/arthurcgc/waf-frontend.

Os resultados dos testes realizados foram satisfatórios. De forma que conseguimos criar, editar e deletar WAFs, e principalmente, comprovar sua eficácia em proteger a aplicação originalmente insegura. Desenvolvemos um sistema completamente expansível com o qual o usuário é capaz de expandir as funcionalidades introduzidas, como: a criação de regras customizadas de segurança, planos de segurança customizados e automatizar possíveis processos através da API desenvolvida. Comprovamos o comportamento esperado de acréscimo de latência nas requisições utilizando WAFs. No entanto, a própria arquitetura do sistema nos permite escalar as instâncias de WAF até o limite de sua arquitetura física, uma solução que pode mitigar esse acréscimo de latência obtido.

Trabalhos futuros envolvem um aprofundamento na arquitetura de proxies reversos. Seria interessante o sistema disponibilizar uma forma de se configurar o servidor de proxy reverso da forma com a qual ele desejar. Atualmente todas as requisições são interpretadas e repassadas para a aplicação, mas poderiam existir opções de configuração do servidor como: estratégia de *keep-alive* das conexões, configurações de *caching* de conteúdo e redirecionamento de páginas de erro. O sistema provê apenas uma configuração de proxy reverso e isso poderia ser expandido. Além disso, é possível automatizar a escalabilidade de objetos no Kubernetes, de tal forma que se o objeto sofre degradação, novas instâncias deste objeto podem ser criadas a fim de reduzir a carga das instâncias existentes. Poderíamos utilizar dessa técnica para automatizar a criação de novas instâncias de WAF quando essas estiverem adicionando demasiada latência nas requisições devido ao alto consumo de CPU.

Referências Bibliográficas

[1] Pałka, Dariusz & Zachara, Marek. (2011). Learning Web Application Firewall - Benefits and Caveats. 295-308. 10.1007/978-3-642-23300-5_23.

[2] Burns, Brendan & Grant, Brian & Oppenheimer, David & Brewer, Eric & Wilkes, John. (2016). Borg, Omega, and Kubernetes. Communications of the ACM. 59. 50-57. 10.1145/2890784.

[3] Vayghan, Leila & Saied, Mohamed & Toeroe, Maria & Khendek, Ferhat. (2019). Kubernetes as an Availability Manager for Microservice Applications.

[4] Silva, Vitor & Kirikova, Marite & Alksnis, Gundars. (2018). Containers for Virtualization: An Overview. Applied Computer Systems. 23. 21-27. 10.2478/acss-2018-0003.

[5] Cito, Jürgen & Ferme, Vincenzo & Gall, Harald. (2016). Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research. 609-612. 10.1007/978-3-319-38791-8_58.

[6] Sommerlad, Peter. (2003). Reverse Proxy Patterns.. 431-458.

[7] T. D. Sobola, P. Zavarsky and S. Butakov, "Experimental Study of ModSecurity Web Application Firewalls," 2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS), 2020, pp. 209-213, doi: 10.1109/BigDataSecurity-HPSC-IDS49724.2020.00045.

[8] Dobies, J., & Wood, J. (2020). Kubernetes Operators: Automating the

Container Orchestration Platform (1st ed.). O'Reilly Media.

[9] Ibryam, B., & Huß, R. (2019). Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications (1st ed.). O'Reilly Media.

[10] Fredj, Ouissem & Cheikhrouhou, Omar & Krichen, Moez & Hamam, Habib & Derhab, Abdelouahid. (2021). An OWASP Top Ten Driven Survey on Web Application Protection Methods. 10.1007/978-3-030-68887-5_14.

[11] Dalai, Asish & Jena, Sanjay. (2011). Evaluation of web application security risks and secure design patterns. 565-568. 10.1145/1947940.1948057.

Apêndices

APÊNDICE A - DESENVOLVIMENTO DA WAF API

Instanciamos uma nova estrutura do tipo Api e chamamos sua função de Start(), que fará com que o servidor seja inicializado na porta correspondente à variável de ambiente WAF_PORT.

```
func main() {
    api, err := api.New()
    if err != nil {
        panic(err)
    }

    api.Start()
}
```

A estrutura Api assim como seu método construtor são definidos a seguir:

```
type Api struct {
    logger  *logrus.Logger
    server  *echo.Echo
    manager manager.Manager
}

func New() (*Api, error) {
    api := &Api{
        logger: logrus.New(),
        server: buildServer(),
    }
    var mgr manager.Manager
    var err error
    if viper.GetBool("outside_cluster") {
        mgr, err = manager.NewOutsideCluster()
        if err != nil {
            return nil, err
        }

    } else {
        mgr, err = manager.NewInCluster()
        if err != nil {
            return nil, err
        }
    }
    api.manager = mgr
    api.setRoutes()
```

```
    return api, nil
}
```

É definido um logger padrão durante sua construção que irá imprimir ao *stdout* todas as requisições recebidas pela API e seus resultados (sucesso ou erro), a variável *server* define o framework utilizado como *router* para as requisições, foi feita a escolha de utilizar o framework Echo, uma implementação de roteador de requisições http performática e com uma grande comunidade de suporte e desenvolvimento por trás. A função buildServer() inicializa o objeto *router* da framework Echo e suas rotas são definidas pelo método setRoutes():

```
func buildServer() *echo.Echo {
    server := echo.New()
    server.Use(middleware.Logger())
    server.Use(middleware.Recover())
    return server
}

func (a *Api) setRoutes() {
    a.server.POST("/", a.createInstance)
    a.server.PUT("/", a.updateInstance)
    a.server.DELETE("/", a.deleteInstance)
    a.server.GET("/", a.healthcheck)
}
```

A última composição da estrutura Api é o *manager* que é a interface que se comunica com a API do Kubernetes, pode-se observar que o *manager* possui duas formas de ser instanciado durante a execução do construtor New(), utilizando o método NewOutsideCluster() ou NewInCluster(). Isso se dá pelo fato de que a API foi implementada permitindo a comunicação com a API do Kubernetes dentro e fora do cluster, ou seja, seu funcionamento é idêntico mesmo que ela esteja sendo disponibilizada fora de um cluster Kubernetes. Durante o desenvolvimento do projeto essa funcionalidade se revelou extremamente útil para se testar e realizar a depuração do código, pois não era necessário criar a todo tempo um novo objeto no Kubernetes para testar novas funcionalidades. Além disso, introduz maior liberdade de arquitetura ao sistema, o único componente que necessariamente precisa estar disponibilizado dentro do cluster é, de fato, o WAF Operator. A definição para se instanciar a forma de comunicação dentro ou fora do cluster é feita por meio da variável de ambiente WAF_OUTSIDE_CLUSTER, quando instanciada com o valor booleano *true* é indicado que a mesma está sendo executada fora do cluster.

O último componente de código pertinente a ser exibido para descrever o funcionamento da API é a estrutura que implementa a interface *manager*.

```
type Manager interface {
   CreateInstance(context.Context, CreateArgs) error
   UpdateInstance(context.Context, UpdateArgs) error
   DeleteInstance(context.Context, DeleteArgs) error
}
```

Declaramos três métodos necessários para a implementação da interface *Manager* CreateInstance(), UpdateInstance() e DeleteInstance(). Suas funcionalidades são auto explicativas por seus nomes e possuem fluxos similares de execução: recebem os parâmetros via requisição http onde esperam em seu corpo uma definição em JSON do objeto WAF, transformam essa definição em um objeto Kubernetes dentro do código e utilizam uma biblioteca de cliente http do Kubernetes para criar, atualizar ou deletar o objeto. A fim de não tornar a exposição dos métodos repetitiva será exposta apenas o fluxo de criação dos objetos WAF pelo *Manager.*

A estrutura que implementa os métodos do *Manager* se chama k8s (apelido para Kubernetes) e possui apenas um campo que a compõe que é o cliente que se comunica com a API do Kubernetes.

```
type k8s struct {
   dynamicClient dynamic.Interface
}
```

O método CreateInstance() é apenas um *wrapper* para o método interno createWafInstance():

```
func (k *k8s) CreateInstance(ctx context.Context, args CreateArgs) error {
   if err := k.createWafInstance(ctx, args); err != nil {
       return err
   }

   return nil
}
```

```
func (k *k8s) createWafInstance(ctx context.Context, args CreateArgs) error {
   protoPrefix := "http://"
   if strings.Compare("HTTPS", args.Bind.Protocol) == 0 {
       protoPrefix = "https://"
   }
   waf := &unstructured.Unstructured{
       Object: map[string]interface{}{
           "apiVersion": "waf.arthurcgc.waf-operator/v1",
           "kind":       "Waf",
           "metadata": map[string]interface{}{
```

```
                "name": args.Name,
            },
            "spec": map[string]interface{}{
                "replicas": args.Replicas,
                "planName": args.PlanName,
                "bind": wafv1.Bind{
                    Name:     args.Bind.ServiceName,
                    Hostname: fmt.Sprintf("%s%s.%s.svc.cluster.local", protoPrefix,
args.Bind.ServiceName, args.Bind.Namespace),
                },
                "rules": wafv1.Rules{
                    CustomRules:           args.Rules.CustomRules,
                    EnableDefaultHoneyPot: args.Rules.EnableDefaultHoneyPot,
                },
                "service": map[string]interface{}{
                    "type": "NodePort",
                },
            },
        },
    }

    _, err := k.dynamicClient.Resource(wafGVR).Namespace(args.Namespace).Create(ctx,
waf, metav1.CreateOptions{})
    if err != nil {
        return err
    }

    return nil
}
```

Vemos que a partir dos argumentos passados criamos o objeto com a ajuda das bibliotecas em Go do Kubernetes previamente mencionadas e chamamos o método de criação do cliente ao final, retornando um erro caso tenha acontecido algo inesperado durante a criação do objeto.

Assim como o WAF Operator a WAF API possui um *Makefile* com as receitas de *build,* execução e *deploy* de seu código. Porém, não foi utilizado nenhum framework de automação para sua criação.

```
.PHONY: build
build:
    go build -o bin/api cmd/main.go

.PHONY: run
run: build
    ./bin/api

.PHONY: image
image:
    docker build . -t arthurcgc/waf:latest
    docker push arthurcgc/waf:latest
```

```
.PHONY: rbac
rbac:
   kubectl apply -f k8s/rbac/

.PHONY: deploy
deploy:
   kubectl apply -f k8s/deploy.yaml

.PHONY: deploy-vulnerable
deploy-vulnerable:
   kubectl apply -f k8s/vulnerable-web-app/dvwa.yaml

.PHONY: all
all: build image rbac
   kubectl apply -f k8s/deploy.yaml
```

Podemos perceber que também fazemos o uso de Dockerfiles para publicar a imagem da API no DockerHub de modo que podemos utilizá-la durante o deploy da mesma no cluster. Além disso, existe a receita de publicação da aplicação no cluster, presente no diretório denominado *k8s* do projeto.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 namespace: waf-operator-system
 name: waf-api
spec:
 selector:
   matchLabels:
     app: waf-api
 replicas: 1
 template:
   metadata:
     labels:
       app: waf-api
   spec:
     serviceAccountName: waf-api-service-account
     containers:
     - name: waf-api
       image: arthurcgc/waf:latest
       ports:
       - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
 namespace: waf-operator-system
 name: waf-api
spec:
 type: NodePort
 selector:
```

```
    app: waf-api
  ports:
    - name: http
      port: 80
      targetPort: 8080
```

É declarado um objeto *Deployment* que possui seu controlador (*Controller)* próprio que irá manter as especificações consistentes caso qualquer mudança de estado aconteça no cluster. Além disso, como já mencionado anteriormente, para expor o objeto e fazer com que se seja capaz de se comunicar com outros componentes dentro e fora do cluster, precisamos expô-lo via um *Service* do Kubernetes, também definido no fragmento de código anterior.

Para testar a API e sua integração com o WAF Operator, podemos utilizar o *curl,* um programa de linha de comando que realiza requisições http.

```
> curl --location --request POST 'localhost:8080' --header 'Content-Type:
application/json' --data-raw '{
       "name": "tcc-sample",
       "namespace": "frontend",
       "plan": "default",
       "bind": {
       "serviceName": "dvwa-service",
       "namespace": "backend"
       },
       "replicas": 1
}'
Created waf resource!
```

APÊNDICE B - DESENVOLVIMENTO DA WAF FRONTEND

Assim como no caso da WAF API as rotas criadas na aplicação de frontend seguem o mesmo fluxo de declaração, e, portanto, será exibido, a fim de evitar repetição de códigos muito similares, apenas o código da rota de atualização da aplicação de frontend, visto que já foi apresentado na figura 5 a rota de criação.

```
@app.route('/update', methods=["GET", "POST"])
def update():
   form = UpdateWafForm()
   if form.validate_on_submit():
       payload = {
           "name": form.name.data,
           "namespace": form.namespace.data,
           "plan": form.planName.data,
           "bind": {
               "serviceName": form.bindSvcName.data,
               "namespace": form.bindNamespace.data,
           },
           "replicas": form.replicas.data
       }
       if form.customRules.data != None:
           filename = secure_filename(form.customRules.data.filename)
           path = 'uploads/' + filename
           form.customRules.data.save(path)
           customRulesList = parse(path)
           payload["rules"] = {
               "customRules": customRulesList
           }
           os.remove(path)
       else:
           payload["rules"] = {
               "customRules": []
           }

       r = requests.put(waf_api, json=payload)
       if r.status_code == 201:
           flash(f"Successfully updated {form.name.data}!", "success")
       else:
           flash("Error: " + r.text, "danger")
   return render_template('update.html', form=form)
```

Instanciamos um UpdateWafForm(), definimos o *payload* (os dados em JSON que serão enviados) e, para complementar o objeto de payload precisamos realizar o *parsing* do arquivo de regras de segurança customizadas caso ele tenha sido enviado pelo usuário. Realizamos o request http PUT para a WAF API e informamos ao usuário seu resultado. O fragmento de código que instancia o formulário de

atualização pode ser observado abaixo, assim como o código de parsing do arquivo de regras de segurança, respectivamente.

```
class UpdateWafForm(FlaskForm):
   name = StringField("Name", validators=[DataRequired(), Length(min=3, max=15)])

   namespace = StringField("Namespace", validators=[DataRequired(), Length(min=3,
max=15)])

   replicas = IntegerField("Replicas", validators=[DataRequired()])

   planName = StringField("Plan", validators=[DataRequired(), Length(min=3,
max=15)])

   bindSvcName = StringField("Service Name", validators=[DataRequired(),
Length(min=3, max=15)])

   bindNamespace = StringField("Service Namespace", validators=[DataRequired(),
Length(min=3, max=15)])

   bindProtocol = SelectField("Protocol",choices=["http", "https"],
validators=[DataRequired()])

   customRules = FileField("Custom Rules", validators=[FileAllowed(["conf"], "Only
.conf files allowed")])

   submit = SubmitField("Update")
```

```
def parse(path):
   customRules = []
   with open(path) as fp:
       for _, line in enumerate(fp):
           customRules.append(line)
   return customRules
```

Assim como os componentes anteriores, foi criado um Dockerfile que nos permite distribuir o código na forma de contêiner e realizar o *deploy* dele no Kubernetes. Para isso, também foi criado um *Makefile* abstraindo os passos necessários.

```
docker-build:
   docker build --tag arthurcgc/waf-frontend:latest .

docker-push:
   docker push arthurcgc/waf-frontend:latest
```

A declaração das especificações da aplicação que a instanciam no cluster fazem o uso da imagem publicada pelo *Makefile,* a configuração em *yaml* necessária para disponibilizar a aplicação no cluster segue a seguir.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 namespace: waf-operator-system
 name: waf-frontend
spec:
 selector:
   matchLabels:
     app: waf-frontend
 replicas: 2
 template:
   metadata:
     labels:
       app: waf-frontend
   spec:
     containers:
     - name: waf-frontend
       image: arthurcgc/waf-frontend:latest
       ports:
       - containerPort: 8000
       env:
         - name: SECRET_KEY
           value: "123"
         - name: WAF_API_URL
           value: "http://waf-api.waf-operator-system.svc.cluster.local"
---
apiVersion: v1
kind: Service
metadata:
 namespace: waf-operator-system
 name: waf-frontend
spec:
 type: NodePort
 selector:
   app: waf-frontend
 ports:
   - name: http
     port: 80
     targetPort: 8000
```

Analogamente à WAF API, declaramos um objeto *Deployment* e um *Service* para expormos a aplicação.

Para testarmos o fluxo de comunicação da aplicação, faremos uma atualização da WAF criada no tópico anterior, *tcc-sample*, adicionando um arquivo de *Custom Rules*, contendo a seguinte regra de segurança customizada:

```
SecRule ARGS:testparam "@contains test" "id:1234,deny,log,status:403"
```

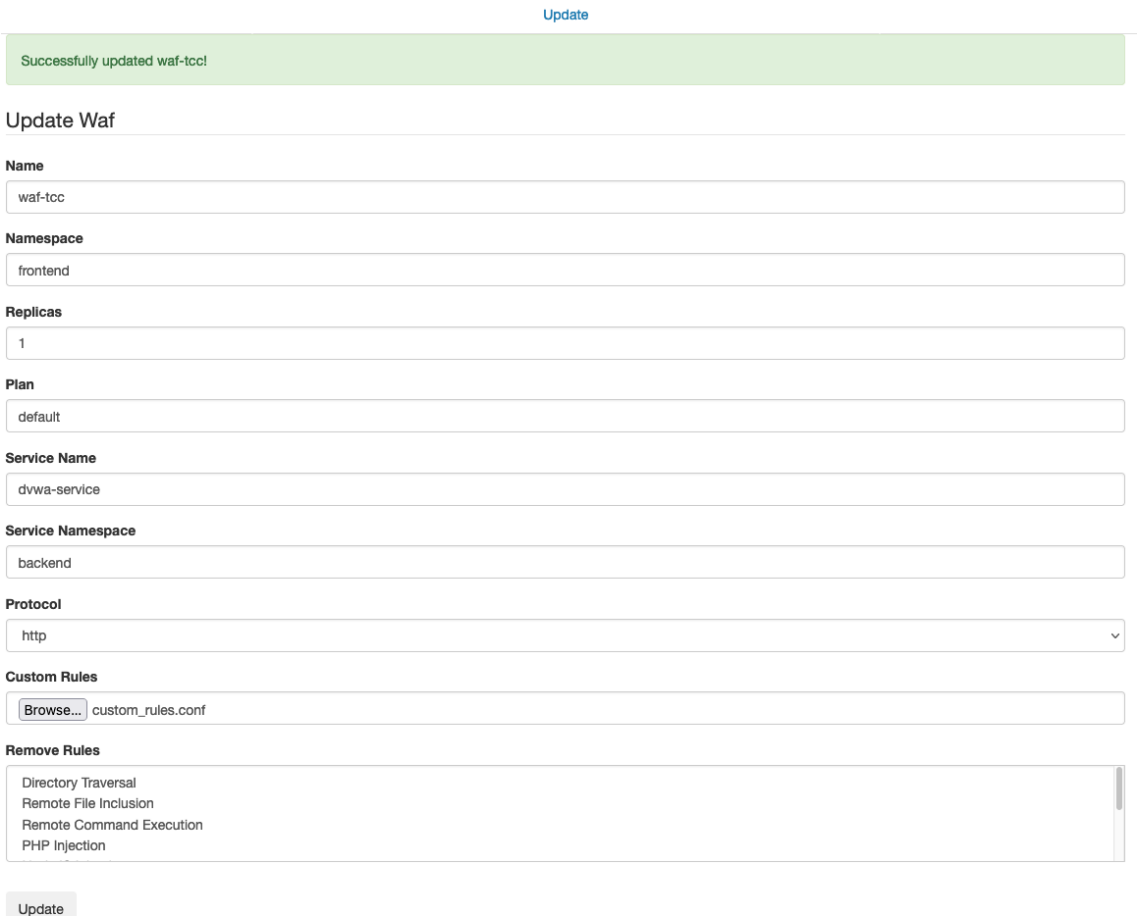


Figura 22: WAF Frontend - operação de update bem sucedida