Tocker: *framework* para a segurança de *containers* Docker

Felipe Balabanian, Marco A.A. Henriques

Faculdade de Engenharia Elétrica e de Computação - Universidade Estadual de Campinas (UNICAMP) CEP 13083-852 Campinas, SP.

felipebalabanian@gmail.com, marco@dca.fee.unicamp.br

Abstract. *Tocker is a framework that restricts the network's communication between Docker containers to the minimum necessary and blocks unnecessary ports with the introduction of firewalls, according to the user settings of services and their relationships in a text file or via a graphical interface. The framework also automatically adds containers containing security monitoring services (Snort) between connections, based on pre-established rules. Moreover, it performs static security analysis of the images used in the containers thanks to an integration with the Snyk scanner that searches in external databases which dependencies (software) of the image are outdated (and vulnerable) and alerts the system administrator.*

Resumo. *Tocker é um framework que restringe a comunicação entre containers Docker ao mínimo necessário e bloqueia as portas não necessárias dos containers com a introdução de firewalls, de acordo com as configurações de serviços e seus relacionamentos feitas pelo usuário em um arquivo texto ou interface gráfica. O framework também adiciona, de forma automática, containers contendo serviços de monitoramento de segurança (Snort) entre as conexões, baseando-se em regras pré-estabelecidas. Mais ainda, ele realiza análise de segurança estática das imagens utilizadas nos containers graças a uma integração com o scanner Snyk que busca em bases de dados externas quais dependências (softwares) da imagem estão desatualizadas (e vulneráveis) e alerta o administrador do sistema.*

1. Introdução

A tecnologia Docker permitiu que implantações de novos sistemas e serviços fossem portáteis, estáveis e rápidas, o que a tornou popular e amplamente utilizada por milhares de desenvolvedores e administradores de sistemas. Porém, muitos desses usuários a utilizam como uma caixa preta, ignorando que ela possui limitações e que demanda cuidados com os aspectos de segurança.

A arte de configurar os *firewalls* de uma rede exige conhecimento técnico das regras, visualização da rede e muita atenção, visto que a adição de uma regra errada pode inutilizar todo o *firewall* e expor toda a rede ao mundo [1]. E, para atacantes motivados, uma brecha de poucas horas no *firewall* é o suficiente para uma invasão [2].

Portanto, o presente trabalho buscou oferecer um framework que possibilite a geração automática de regras de *firewall* para a conformidade de segurança de um sistema implementado em Docker e mais duas proteções adicionais: avaliação estática (busca com ferramenta Snyk [3] de versões desatualizadas e potencialmente vulneráveis

dos módulos das imagens adotadas) e avaliação dinâmica (monitoramento do tráfego de rede com ferramenta Snort para detecção de intrusão).

2. Objetivos

O trabalhou visou construir um *framework* denominado Tocker que auxilie os administradores de serviço em *containers* Docker a lidar melhor com questões de segurança. O *framework* adiciona quatro proteções de segurança nas redes Docker:

1. restringe a comunicação entre *containers* ao mínimo necessário com *firewalls*;
2. filtra as portas não autorizadas dos *containers* com *firewalls*;
3. realiza análise estática nas imagens utilizadas nos *containers*;
4. adiciona monitoramento entre as conexões críticas.

3. Revisão de literatura

3.1. Pilha de rede do Docker

A plataforma de *containers* Docker é demasiadamente permissiva quanto a comunicação entre os seus *containers* na mesma rede, permitindo que eles se comuniquem através de uma conexão comum (*bridge*) livremente.

Suponha que se queira a topologia de rede apresentada na Figura 1.(a). O Docker entregará a topologia da Figura 1.(b), que é equivalente à da Figura 1.(c), mas inclui várias outras possibilidades de conexão que são desnecessárias (marcadas em vermelho). Estas conexões desnecessárias aumentam a superfície de ataque caso um *container* seja invadido, permitindo, a partir dele, a invasão de outros *containers* na rede, com os quais ele não precisaria se comunicar para exercer suas atividades normais. Nota-se, porém, que esta abordagem é a mesma usada em muitas redes tradicionais com topologia estrela, sem restrições entre os nós internos.

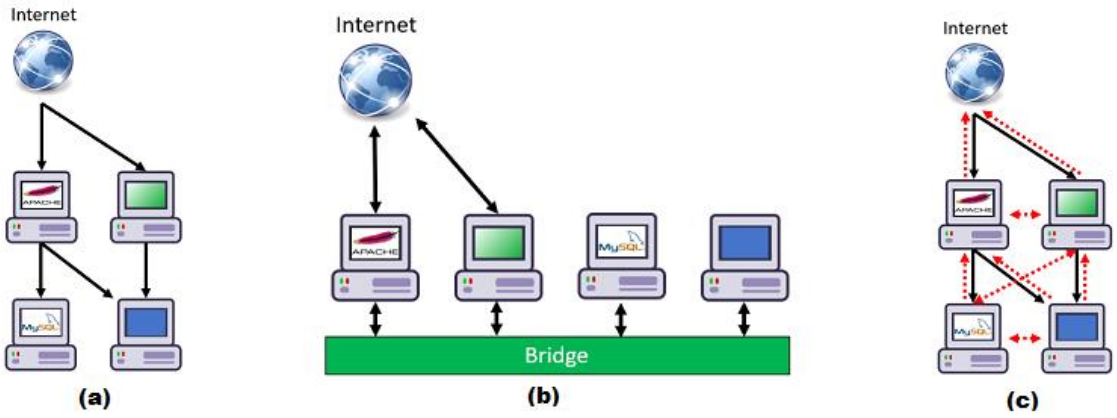

Figura 1: (a) Topologia desejada. (b) Topologia que o Docker entrega quando se deseja a topologia da Figura 1.(a). (c) Releitura da Figura 1.(b), explicitando as conexões entre os *containers* permitidas pela *bridge* e ressaltando, em vermelho, as conexões desnecessárias, mas possíveis.

3.2. Segurança dos *containers* em redes diferentes

No Docker, por padrão, os *containers* criados dentro da mesma rede só se comunicam com outros *containers* da mesma rede. Logo, em teoria, seria possível criar redes seguras e restritas utilizando somente os recursos do Docker. Há duas estratégias para a comunicação de *containers* de redes segregadas, apresentadas a seguir e novamente tomando a rede da Figura 1.(a) como exemplo.

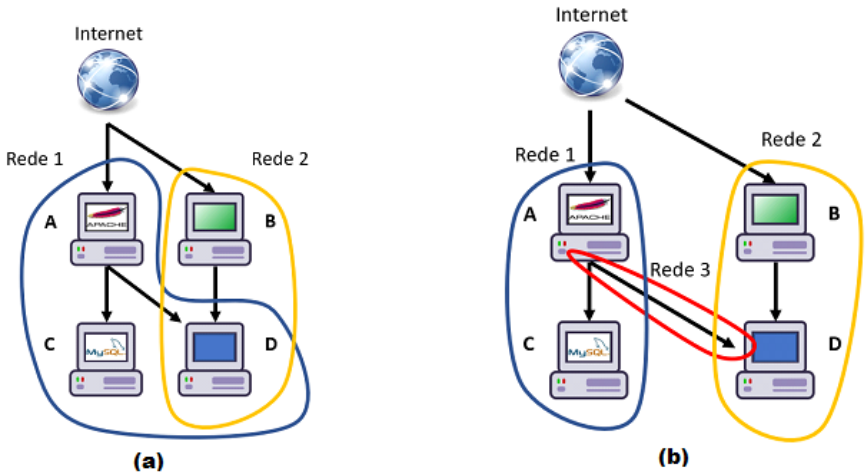

Figura 2: Estratégia para comunicação de *containers* em redes diferentes: (a) adiciona-se os *containers* de interesse nas duas redes; (b) adiciona-se uma rede exclusiva para os *containers* que precisam se comunicar.

3.2.1 Primeira estratégia

Incorporar o *container* de interesse nas duas redes: nesta abordagem o *container* pertence às duas redes, recebe duas interfaces (uma para cada rede) e possui dois endereços IP, o que lhe permite se comunicar com outros *containers* em ambas as redes. Com esta abordagem, a rede da Figura 1.(a) se transforma na rede da Figura 2.(a). A Rede 1 pode crescer à medida que precisar incorporar mais *containers* e, considerando que os sistemas não são iguais em importância, é provável que haja uma grande rede principal conectada a várias outras redes satélites menores. Ou seja, se houver múltiplas redes com interseção e uma delas for muito maior que as outras, então o problema dos *containers* estarem na mesma rede continuará existindo em relação à esta rede maior, pois a maioria dos *containers* estará incluída nela.

3.2.2 Segunda estratégia

Adicionar uma rede exclusiva para cada comunicação inter-redes: nesta abordagem uma nova rede é criada para cada conexão que existe entre os *containers* de duas redes diferentes. Isso implica que, para cada par de *containers* que se comunicam, é preciso criar um novo par de interfaces de rede com seus respectivos IPs. Esta é a estratégia mais onerosa, pois para cada conexão adiciona-se duas interfaces de rede, um network-namespace e uma *bridge*, o que obviamente dificulta o gerenciamento dos componentes. Com todos esses componentes adicionais, há um crescimento da complexidade das informações que descrevem o sistema e uma maior dificuldade de gerenciamento. Portanto, esta é uma abordagem evitada em redes grandes e com muitas conexões. Com esta abordagem, a rede da Figura 1.(a) se transforma na rede da Figura 2.(b). Uma

solução alternativa seria criar uma rede exclusiva para todas as conexões inter-redes, o que viola o princípio de um-para-um e não ajuda no requisito de segurança, pois esta nova rede é muito extensa e possui acesso a todas as outras redes, sendo tão grande quanto o número de conexões desejadas. Além disso, estaremos novamente voltando ao problema de uma única rede.

Vale ressaltar que ao iniciar um *container* sem especificar a rede, ele é conectado na rede padrão do Docker (nomeada "bridge") e o problema se resume novamente a uma única rede. Esta é a abordagem mais simples, menos preocupada com a segurança e a mais adotada, pois gerenciar e cuidar da segregação de redes é trabalhoso e não oferece nenhum benefício muito palpável. O *framework* Tocker proposto visa reverter este quadro, fornecendo um ambiente que implementa o conceito da segunda solução apresentada de forma muito mais eficiente e transparente para o Docker. Para isso, ele utiliza para segregar *containers* os mesmos conceitos que o Docker usa para segregar redes.

4. Detalhamento dos problemas e respectivas soluções

A seguir são explicitadas as soluções para cada um dos problemas encontrados quando os *containers* estão na mesma rede. No fim é apresentada a arquitetura geral do framework Tocker.

4.1. Livre comunicação de todos os *containers* pela bridge

Na Figura 3 há uma representação de como é o arranjo padrão do Docker quanto às conexões e aos *firewalls* [4]. Todos os *containers* passam por um *firewall* para acessarem a *internet*; porém possuem acesso livre para se comunicarem pela *bridge*. Isso permite que, caso um *container* seja invadido, ele possa alcançar e atacar qualquer outro *container* que pertença à mesma rede.

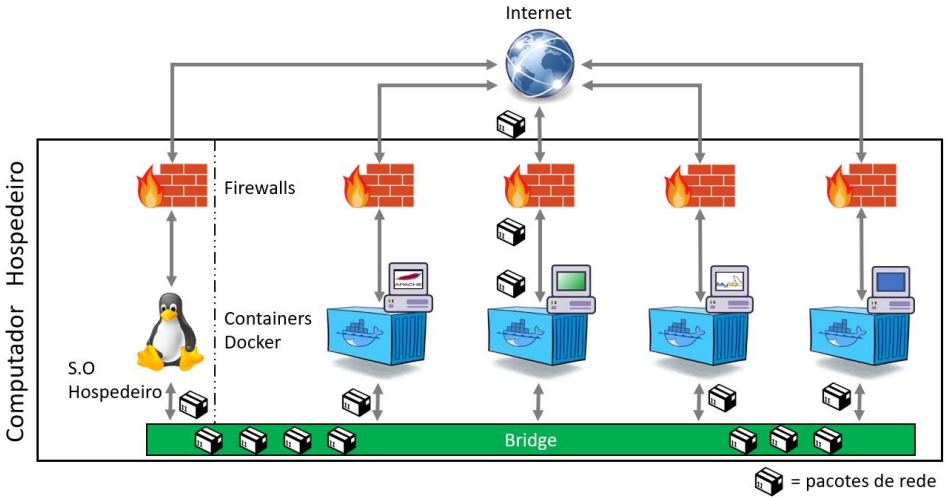

Figura 3: Arranjo padrão do Docker quanto às conexões e aos *firewalls*.

O framework proposto adiciona *firewalls* entre os *containers* e a *bridge*, como mostrado na Figura 4. Isso garante que seja possível conseguir uma topologia exatamente como da Figura 1.(a) respeitando o sentido das conexões, ou seja, que só seja possível iniciar novas conexões no sentido da seta. A configuração da rede Docker

que originalmente era a da Figura 3, se transforma na da Figura 4 e consegue prevenir o ataque como representado nas Figuras 5.(a) e (b).

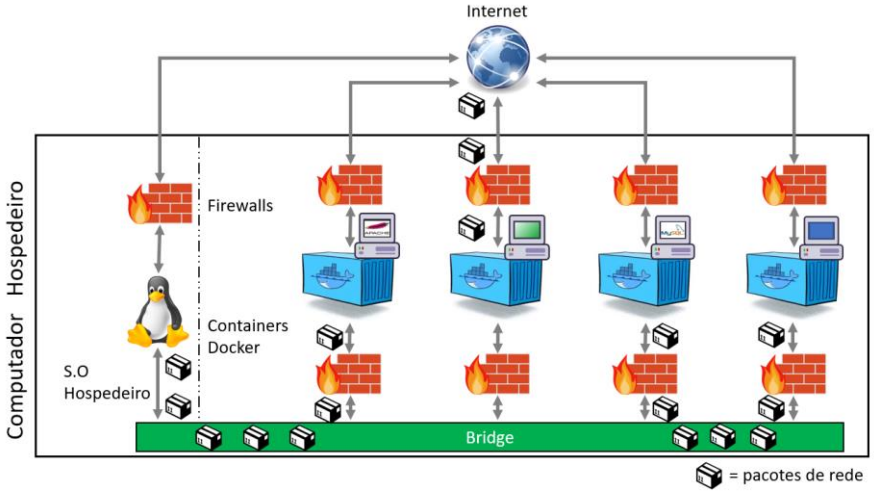

Figura 4: Novo arranjo da Fig. 3 após o framework Tocker ser utilizado.

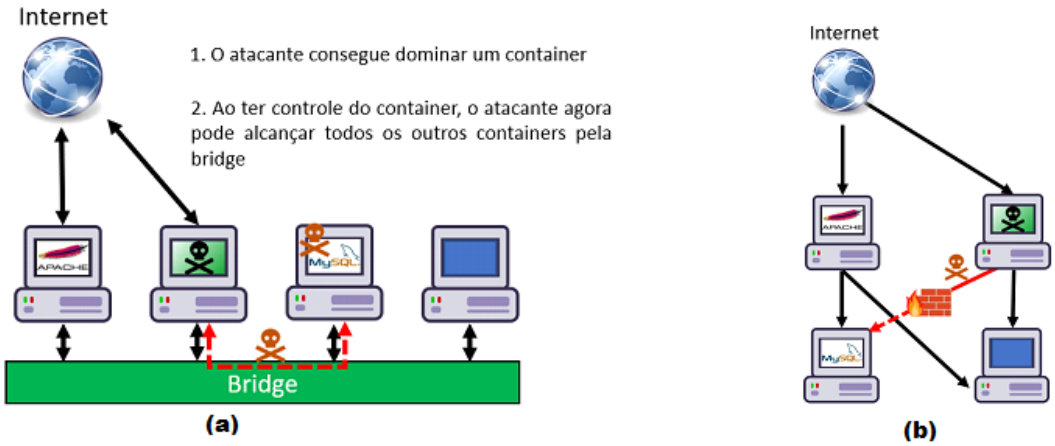

Figura 5: (a) Ataque de um *container* adjacente se utilizando da livre comunicação pela bridge. (b) Tocker bloqueia conexões de ataque de *containers* adjacentes.

4.2. Conexão reversa (shell reverso)

A implementação padrão dos *firewalls* do Docker não distingue o sentido de início da comunicação, ou seja, quem inicia a conexão. Isso faz sentido em ambiente de desenvolvimento, onde o *container* precisa buscar pacotes na *internet* e fazer *downloads*; porém é mais raro quando o serviço já foi colocado em produção, pois o comportamento padrão de um servidor é atender requisições da *internet* e não iniciá-las. Esta falta de distinção de origem torna o servidor mais propício a ataques de conexão reversa. O atacante explora uma vulnerabilidade, faz execução de código arbitrário para o *container* iniciar uma conexão com a máquina do atacante na *internet* e expor o seu *shell* (Figura 6.(a)) e, assim, obter controle do *container*.

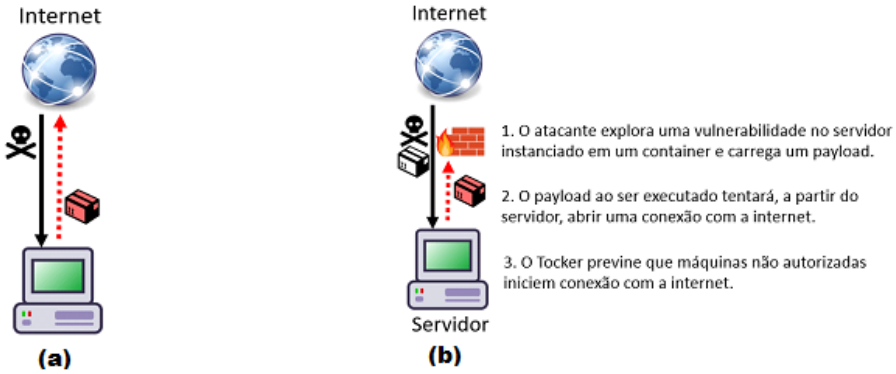

Figura 6: (a) Ataque de shell reverso. (b) Como um ataque de shell reverso é prevenido pelo Tocker.

Como o servidor não abre novas conexões com a *internet* em situações normais de operação, ao definir uma topologia como da Figura 1.(a) no *framework* Tocker, este permitirá que o servidor (no *container*) receba requisições da *internet* e as responda. Entretanto, o Tocker restringirá o servidor de iniciar novas conexões para a *internet*, pois endurecerá as permissões dos *firewalls* que estão entre os *containers* e a *internet* na Figura 4. Caso uma vulnerabilidade seja explorada no servidor e um *payload* de *shell* reverso seja executado, este não conseguirá acessar a *internet* ao tentar iniciar uma nova conexão. Como os servidores no Docker são expostos para a *internet* através de um mapeamento NAT no computador hospedeiro, acessá-los diretamente se torna difícil. Mesmo que este acesso ocorra, ainda haverá o *firewall* para filtrar as portas que não são do serviço padrão do servidor. A Figura 6.(b) ilustra como o Tocker protege contra conexões reversas.

4.3. Erros inadvertidos em Iptables

O docker utiliza Iptables [5] e é sensível à sua modificação, principalmente na tabela NAT e na cadeia FORWARD da tabela FILTER. Caso um administrador modifique inadvertidamente a Iptable, toda a rede Docker pode ficar exposta para computadores adjacentes, ou seja, na rede local do computador hospedeiro. O framework Tocker dificulta isto inserindo suas regras de uma maneira que sejam verificadas primeiro e exigindo que todos os pacotes originários de, ou com destino a, um IP da rede Docker casem com uma regra ou sejam descartados (whitelist). Isso remove a possibilidade de avaliação de regras de terceiros para IPs da rede criada com o Tocker. Além do mais, as regras do Tocker são baseadas no IP de cada *container* e são muito mais granulares que as regras de segregação de redes do Docker que são baseadas nos bridges (interfaces de rede) de origem e destino do pacote, o que torna esta proteção mais eficaz.

5. Recursos extras do framework

5.1. Análise Estática

A fim de oferecer mais recursos de segurança, o scanner Snyk [3] foi integrado no framework Tocker, permitindo que todas as imagens utilizadas nos *container*s sejam analisadas. O que o Snyk faz é buscar no Dockerfile todas as dependências que serão instaladas na imagem e todas as dependências que existem na imagem base. Uma vez com a lista de todos os pacotes que serão instalados na imagem e suas versões, ele busca

(em bases de dados específicas) por CVEs (vulnerabilidades) relacionados à versão daquela dependência. No final é produzido um relatório por imagem, informando as vulnerabilidades que ela possui, e alertando o administrador da rede sobre quais dependências da imagem precisam ser atualizadas. Todos os relatórios são salvos numa pasta do Tocker para consultas futuras. A Figura 7 mostra o que o Tocker apresenta quando está iniciando uma nova rede e executa o Snyk.

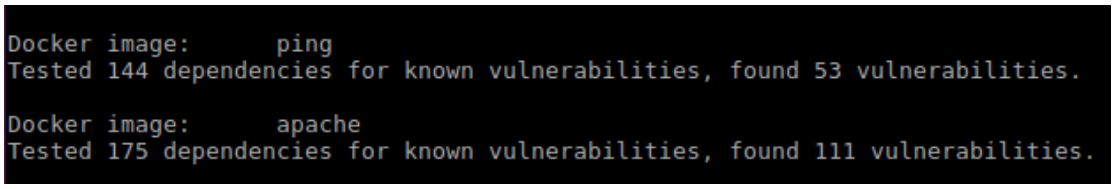

Figura 7: Exemplo de alerta do Tocker ao escanear as imagens com o Snyk.

5.2. Monitoramento de Segurança

Caso seja desejado, o Tocker permite adicionar *container*s de monitoramento (que precisam estar no meio de conexões) de forma automática e baseada em regras pré-estabelecidas como, por exemplo, adicionar um *container* para gravar logs de todos os *containers* que estiverem implementando um *webserver*. Vale notar que este é o único processo no Tocker que utiliza de métodos invasivos para os *containers*, pois altera as tabelas de roteamento (*route tables*) deles. Todos os outros processos descritos no trabalho utilizam os *containers* como caixas pretas e sem alterá-los.

Para isso o *framework* Tocker importa um arquivo de regras com todas as transformações que ele deve aplicar sobre a topologia. Cada regra especifica uma modificação que deverá ser feita sobre a topologia para inserir *containers* de monitoramento de segurança, ou seja, *containers* que necessitam estar entre as conexões. Por exemplo, a regra insertAfter(“Internet”, “Snort”) insere *containers* contendo o IDS Snort entre conexões da *internet* com qualquer *container*. Isso permite que, de forma rápida, o usuário defina quais medidas de monitoramento sejam mais adequadas para cada tipo de serviço, de forma que estas medidas sejam aplicadas a todos os serviços na rede. Um exemplo de situação que é resolvido com este framework seria: todo *webserver* da rede deve possuir um detector de DDoS [6]; para isso, será necessário apenas adicionar a regra insertBefore(“webserver”, “alguma imagem de DDoS monitor”) no arquivo de configuração.

6. Interfaces do framework

6.1. Interface Textual

Como pode ser visto na Figura 8.(a), a entrada textual possui duas seções: a primeira descreve os *containers* da rede, seu nome (HostName), imagem utilizada (HostImage), tipo de serviço que ele oferecerá (HostType), o nome do fabricante (HostVendor - opcional) e as portas as quais o serviço receberá conexões (Ports); A segunda parte, que inicia com a palavra reservada “Connections:”, define as conexões autorizadas entre os *containers* e deverá seguir a sintaxe Origem:Portas:Destino, sendo que o campo Portas pode ser preenchido por um valor numérico (80), uma sequência de valores numéricos (80,8080,8888), um asterisco (*) indicando todas as portas ou uma cerquilha (#)

indicando todas as portas nas quais o serviço receberá conexões (campo Ports do *container* Destino).

A Figura 8.(a) ilustra um exemplo da entrada textual que o Tocker deve receber para construir a rede Docker mais segura. Nela pode-se ver que há a especificação de um ponto de acesso para a *internet* e de três *containers* (um *webserver*, um banco de dados e um gerador de conteúdo para o banco de dados). No final do arquivo é possível ver as especificações das conexões que devem ser permitidas entre os *containers*: *internet* pode se conectar em todos os serviços públicos do *webserver* (ambas as portas 80 e 443, indicadas por #); o *webserver* W1 pode se conectar somente na porta 3306 do banco de dados DB (apesar de estarem disponíveis também as portas 33060, 33061 e 33062); o banco DB não pode se conectar a nenhum outro serviço, já que não há nenhuma especificação neste sentido; e o *container* H1 pode se conectar em qualquer porta do banco de dados (indicador *).

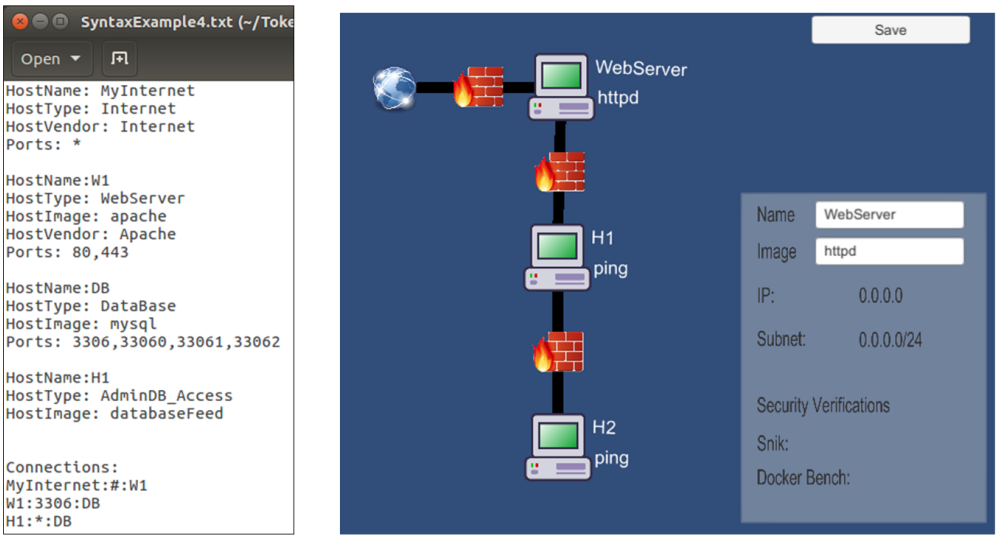

Figura 8: (a) Exemplo de entrada textual que define e descreve a rede para o Tocker. (b) Interface gráfica do Tocker, ainda em fase de protótipo.

6.2. Interface Gráfica

Foi desenvolvida uma interface gráfica (ainda em fase de protótipo), apresentada na Figura 8.(b), para auxiliar na especificação da topologia da rede de *containers*. A motivação para a mesma é permitir ao usuário desenhar a rede na interface gráfica e essa ser traduzida para a interface textual descrevendo a topologia, que é a entrada consumida pelo framework Tocker.

A interface gráfica, inicialmente, apresenta para o usuário uma área de trabalho com fundo azul, sem nenhum elemento na tela. Ao pressionar a tecla F1, um *container* (em formato de computador) é criado no lugar onde o mouse estiver, o qual pode ser arrastado pela tela ao pressionar o botão esquerdo do mouse e movimentá-lo. Ao clicar com o botão esquerdo sobre o *container*, um painel lateral é aberto com informações sobre o nome e imagem daquele *container*, tais informações podem ser modificadas no painel. Ao clicar com o botão direito do mouse sobre um *container* e depois sobre outro, cria-se uma conexão entre eles, representado por uma linha com um *firewall* no meio. Ao clicar com o botão esquerdo do mouse sobre o *firewall*, abre-se um painel lateral

para configurar as conexões autorizadas entre os *containers*. Finalmente, ao clicar sobre o botão *Save*, a topologia desenhada é convertida para a sua forma textual. Apesar de ainda estar na fase de prototipagem, esta interface já se mostra útil para a definição de redes de forma mais simples e intuitiva pelos usuários, diminuindo assim o risco de configurações incorretas.

7. Arquitetura do framework

Na Figura 9 está representada a arquitetura geral do framework Tocker, contemplando todos os estágios acima descritos. O código do projeto pode ser acessado publicamente no GitHub [7].

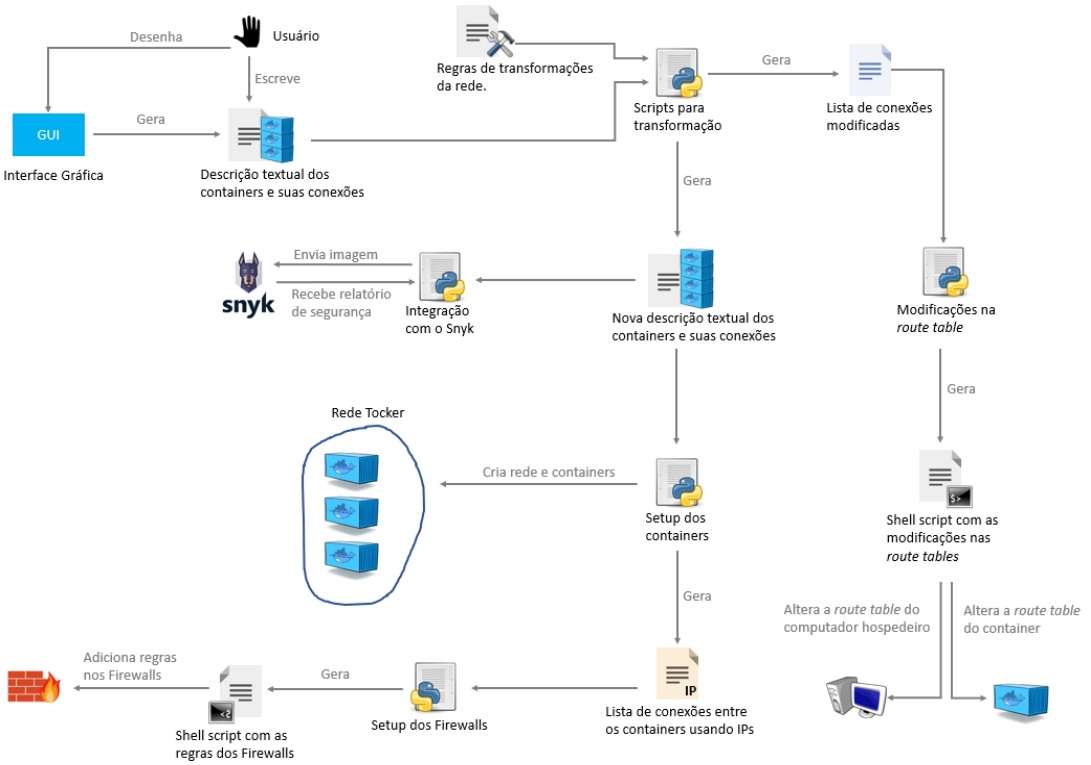

Figura 9: Arquitetura geral do Tocker.

8. Resultados e Discussões

O framework proposto consegue assumir a responsabilidade de configurar vários *firewalls* para redes Docker, diminuindo a complexidade do problema ao abstrair as configurações em um desenho na interface gráfica ou em uma descrição textual. Ao invés do desenvolvedor ser obrigado a alterar as regras de cada *firewall* uma a uma num terminal com centenas de linhas e regras lógicas, ele agora pode fazê-lo utilizando uma interface gráfica com elementos conceituais, aproximando-o do verdadeiro objetivo que é o de se preocupar com as relações entre os elementos. Por exemplo: na configuração de uma rede com cinco *containers* conectados em topologia de anel e considerando que todos possuam conexão com a *internet*, para atingir o mesmo nível de segurança proporcionado pelo framework Tocker (sem utilizar os *containers* de monitoramento), seria necessária a adição no *firewall* de 30 regras (5 x 2 = 10 para *internet* nos dois sentidos mais 5x2x2=20 para o anel, também nos dois sentidos). Neste caso pode-se ver que são maiores as chances de erro em alguma(s) destas configurações. Por outro lado,

para definir esta mesma rede no Tocker seria necessário apenas desenhar (ou inserir no arquivo texto de configuração) as 15 conexões (Internet:5 + Anel:10), deixando toda a complexidade por conta do próprio framework. Estima-se que quanto maior a rede de serviços e a complexidade entre suas conexões, mais vantajosa será a adoção de um framework como Tocker.

9. Conclusões e Trabalhos Futuros

Neste trabalho foi desenvolvido um framework (Tocker) que automatiza a manipulação de regras de *firewalls* a fim de tornar as topologias de rede Docker mais seguras e restritivas. Implementações de grandes redes serão particularmente favorecidas com o uso deste framework pela redução de trabalho manual devido a automatização e a consequente redução dos erros humanos. Além da segurança adicional provida pelas funcionalidades básicas do Tocker, uma interface gráfica e dois módulos extras foram incluídos: a primeira visa facilitar ainda mais o trabalho do administrador dos *containers* e os últimos buscam incrementar a segurança via análise estática das imagens com o Snyk e a adição de *containers* de monitoramento.

Dentre possíveis trabalhos futuros, destacamos: (i) desenvolver um framework que, a partir de uma rede Docker já em operação, consiga inferir o comportamento das conexões e criar regras de *firewalls* que restrinjam as comunicações na rede sem afetar a sua funcionalidade. E como é preciso gerenciar falsos negativos e positivos, temos aqui uma possível e nova aplicação para a área de inteligência artificial; (ii) aprimorar a interação do usuário com a interface gráfica, de forma a aproximar o desenvolvedor menos experiente dos conceitos de segurança e, assim, permitir o entendimento e a exploração de recursos mais avançados; (iii) estudar como *containers* de monitoramento ou de serviço podem ser inseridos, trocados ou removidos nas redes Docker de forma dinâmica, permitindo que *firewalls* e outras soluções de segurança possam ser implementados sem a supervisão humana.

Referencias

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey", in *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14-76, Jan. 2015.

[2] L.Whitney - Security firm Barracuda hit by cyberattack<https://www.cnet.com/news/security-firm-barracuda-hit-by-cyberattack/> último acesso: 30/06/2019.

[3] Snyk <https://snyk.io/> [Página do fabricante] último acesso: 30/06/2019.

[4] Runnable Team - Basic Docker networking <https://runnable.com/docker/basic-docker-networking> último acesso: 30/06/2019.

[5] Linode Group <https://www.linode.com/docs/security/firewalls /control-network-traffic-with-iptables/> último acesso: 30/06/2019.

[6] Chelladhurai, Jeeva, Pethuru Raj Chelliah, and Sathish Alampalayam Kumar. "Securing docker containers from denial of service (dos) attacks." 2016 IEEE International Conference on Services Computing (SCC). IEEE, 2016.

[7] F.Balabanian <https://github.com/FelipeBala/Tocker> [Código do projeto] último acesso: 30/06/2019.