INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Proteção de Dados, Gestão de Identidade e Deteção de Incidentes em Clusters Kubernetes

LUÍS MIGUEL SEIXAS CORREIA
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Proteção de Dados, Gestão de Identidade e Deteção de Incidentes em Clusters Kubernetes

Luís Correia

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Cibersegurança e Administração de Sistemas

Orientador: Jorge Pinto Leite
Supervisor: Duarte Petiz

Porto, 24 de junho de 2025

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 24 de junho de 2025

Resumo

O Kubernetes é uma tecnologia em adoção crescente, e como tal a necessidade de garantir a sua segurança é uma preocupação essencial para administradores destas infraestruturas. Considerando o caso específico de instalações em *bare-metal*, a preocupação de segurança é fulcral, dado não existirem as funcionalidades disponibilizadas por fornecedores *cloud*. Desse modo, identifica-se a importância da existência de uma solução de segurança integrada, que contemple a encriptação de dados, controlo de acessos, e deteção de incidentes.

Foi realizada uma revisão sistemática de literatura, onde se investigaram técnicas de encriptação para a base de dados etcd, bem como práticas de autenticação e controlo de acessos. No âmbito da deteção e resposta a incidentes, foi investigado um conjunto de ferramentas com diferentes funções, aferindo o seu potencial papel numa solução de segurança holística.

Com base neste conhecimento, foi implementada uma solução que endereça a deteção de incidentes através da conjugação de diversas ferramentas, oferecendo uma perspetiva integrada da postura de segurança de um ambiente Kubernetes. Esta solução foi testada através da simulação de atividades de intrusão, avaliando-se se os eventos são detetados e reportados centralmente. Estes testes demonstraram a eficácia da solução proposta na deteção de incidentes.

Palavras-chave: Kubernetes, segurança, RBAC, MFA, deteção de ameaças

Abstract

Kubernetes is a technology with increasing adoption, and the need to guarantee it's security is of utmost concern for the administrators of these infrastructures. Considering the specific case of bare-metal environments, the security concern is mandatory, due to the absence of features usually present in cloud environments. It is therefore important to have an integrated security solution that includes data encryption, access control and incident detection.

A systematic literature review was carried out, investigating encryption techniques for the etcd database, as well as authentication and access control practices. Within the scope of incident detection and response, a set of tools with different functions was investigated, studying their potential role in a holistic security solution. Systematising the knowledge acquired made it possible to identify limitations and opportunities for integration in order to build an integrated solution that meets the security requirements of these environments.

Following this process, an architecture for the solution was proposed, consisting in the usage of several tools investigated during the literature review, and detailing how the information generated by these can be integrated. Regarding incident detection, the architecture defines the usage of multiple tools for the detection of vulnerabilities - `kube-score`, Tetragon, Trivy and Kubernetes-native audit policies. These are then integrated via the collection of the metrics and logs they expose, which are collected by Prometheus - a metrics database - and `fluent-bit` a log collector which aggregates log entries and ships them to Elasticsearch, a database used for storing logs. This information can then be centrally consulted in Grafana, the chosen tool for visualization. To address practices such as access and admission control, Kubernetes RBAC policies and OPA Gatekeeper were the chosen technologies. Regarding authentication, this is architecturally delegated to an external component - Google SSO - which allows for centralized enterprise identity management. The encryption of data stored in `etcd` was also addressed, adopting the KMSv2 encryption system using Hashicorp Vault as an external KMS.

Having defined the architecture, the implementation of it was carried out, focusing heavily on the incident detection components and their integration. The defined tools were configured according to the proposed requirements and objectives, resulting in a proof-of-concept which was then tested. These tests generally consisted in simulating potential intrusions and evaluating whether or not these are detected. Considering the encryption of Secrets, the tests were completely positive, verifying that secret values are no longer displayed in clear-text in `etcd`, but rather encrypted using a set of keys managed by the external KMS. The static analysis tool - `kube-score` proved rather eficient in detecting misconfigurations in Kubernetes manifests, detecting most of the expected conditions, as well as additional ones. Tetragon also detected all of the expected events, but a large number of false-positives was detected. Finally, Trivy properly reported vulnerabilities in an example container image, as well as compliance checks of the cluster.

Keywords: Kubernetes, security, RBAC, MFA, threat detection

Agradecimentos

Primeiramente, gostaria de expressar o mais profundo agradecimento aos meus pais. Sem o seu inestimável apoio, esta jornada académica certamente não teria sido possível. Foi pela sua motivação que me inscrevi no Mestrado e que o levei a cabo até ao fim. Estendo esta nota de gratidão aos meus amigos, cujo apoio e camaradagem foram fundamentais.

Agradeço à Jscrambler por me providenciar com a oportunidade de desenvolver este projeto, bem como por todo o apoio prestado no decorrer do mesmo. Expresso também uma nota de agradecimento ao meu supervisor, Duarte Petiz, por tornar este projeto uma realidade e por me ter acompanhado durante todas as fases do seu desenvolvimento.

Agradeço ao Instituto Superior de Engenharia do Porto, por me equipar com as habilidades necessárias para desenvolver este projeto. O conhecimento que adquiri nesta instituição é altamente valioso, e algo que me será útil em todas as fases da minha carreira profissional.

Por fim, expresso numa nota de agradecimento particular ao meu orientador, Professor Jorge Pinto Leite, por todo o apoio no decorrer desta dissertação. Obrigado pelo *feedback* prestado de modo regular, pelo cuidado e atenção ao detalhe na revisão do trabalho, e pelas diversas sessões de discussão construtiva que conduzimos. Este processo contribui em grande medida para a qualidade do trabalho, pelo que estou extremamente agradecido.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Código

Lista de Acrónimos

ABAC	Attribute-Based Access Control.
CNI	Container Network Interface.
CRD	Custom Resource Definition.
IDR	Incident Dectection and Response.
KMS	Key Management Service.
MFA	Multi-Factor Authentication.
PSP	Pod Security Policy.
RBAC	Role-Based Access Control.
RQ	Research Questions.

Capítulo 1

Introdução

O presente capítulo visa introduzir o projeto desenvolvido, apresentando o seu contexto envolvente, a definição concreta do problema e a identificação dos objetivos que serão abordados. Serão também apresentadas considerações éticas relevantes, bem como a estrutura do documento.

1.1 Contexto e Problema

O desenvolvimento tecnológico permitiu às organizações a utilização de novas abordagens para a disponibilização do seu serviço aos seus clientes, num ambiente cuja competitividade é crescente. Uma abordagem possível consiste na utilização de *containers*, que acrescentam uma camada de abstração entre o *software* e o ambiente onde este é executado, permitindo um desenvolvimento mais agnóstico relativamente ao meio onde estas aplicações são implementadas, bem como possibilitando uma maior escalabilidade. Contudo, demonstrou-se a necessidade de uma tecnologia para realizar a orquestração destes *containers*, nomeadamente no que toca à sua escalabilidade de forma dinâmica, bem como a sua resiliência. O Kubernetes mostra-se como uma solução a este problema, e a sua dominância de mercado é cada vez mais evidente. Na pesquisa conduzida em (Cloud Native Computing Foundation 2023), verificou-se que 84% dos interrogados está ativamente a utilizar ou a avaliar o Kubernetes, observando-se também que 71% das organizações interrogadas adota esta tecnologia em ambientes de produção.

No entanto, a crescente adoção de uma tecnologia encontra-se sempre acompanhada de novas questões de segurança. Tratando-se de uma tecnologia complexa como o Kubernetes, estas preocupações são ainda mais proeminentes. No inquérito (Red Hat 2024), 67% dos inquiridos afirmaram ter atrasado ou abrandado implementações em ambientes produtivos devido a preocupações de segurança relativamente a *containers* e Kubernetes. Adicionalmente, 46% dos participantes identifica a perda de clientes como resultado de um incidente de segurança originado em *containers* ou Kubernetes. Realça-se também que 60% dos inquiridos estão preocupados com vulnerabilidades, configurações incorretas e exposições nestes ambientes.

Ainda que existam configurações e ferramentas cujo objetivo é melhorar a segurança de ambientes Kubernetes de acordo com as necessidades organizacionais, que serão distintas entre elas, não existe nenhuma solução que se foque simultaneamente nas principais faces da segurança neste tipo de ambiente. Como tal, a inexistência de uma solução integrada para proteção de dados, gestão de identidade e resposta a incidentes em *clusters* Kubernetes é o principal problema endereçado pelo atual projeto. Mais concretamente, pretende-se

endereçar este problema numa infraestrutura Kubernetes *bare-metal*, sendo esta uma necessidade de negócio expressada pelos *stakeholders*. A vantagem da integração de todos os componentes numa única solução é o facto de esta potenciar uma melhor abordagem e visão face a possíveis incidentes de segurança, auxiliando a resposta aos mesmos.

Para esse efeito, a investigação abordará a aplicação de técnicas avançadas de criptografia para a proteção de volumes persistentes e base de dados etcd, bem como a implementação de políticas de Role-Based Access Control (RBAC) e autenticação multi-fator (MFA) para garantir a segurança do acesso. Além disso, será desenvolvido um sistema de deteção e resposta a ameaças (IDR) para monitorizar continuamente o ambiente, identificar anomalias e responder rapidamente a incidentes de segurança. Será realizada a avaliação da solução de modo a aferir se esta corresponde aos requisitos colocados.

1.2 Objetivos

Tendo sida realizada uma clara definição do problema, bem como do contexto em que este se insere, foram delineados os seguintes objetivos:

1. Analisar ferramentas e práticas existentes.
2. Implementar criptografia robusta para volumes persistentes e etcd.
3. Desenvolver políticas eficazes de RBAC e MFA para gestão de identidade.
4. Criar e implementar um sistema de IDR para Kubernetes.
5. Integrar ferramentas de monitorização contínua e resposta a incidentes.
6. Avaliar a eficácia das práticas de segurança e resposta a incidentes simulando cenários reais.

No objetivo 2 é mencionado o conceito de criptografia robusta. Ora, a definição da robustez de uma dado método criptográfico surgirá da análise dos métodos suportados pelas tecnologias reveladas na secção de estado da arte. Determinar-se-ão os métodos cuja implementação é possível, e estes serão comparados entre si face aos requisitos organizacionais de modo a aferir qual ou quais constituem uma melhor opção para figurar da solução desenvolvida.

Relativamente ao objetivo 3, é fulcral realçar que a definição de políticas eficazes irá advir dos resultados da revisão de literatura realizada posteriormente, bem como da análise e desenho da solução, pois para aferir se uma política é eficaz, esta deve ser enquadrada no contexto organizacional em questão e os seus requisitos.

1.3 Considerações éticas

Previamente a proceder com a realização do projeto, há que conduzir uma análise dos aspetos éticos implicados pelo mesmo, sendo essa a função da presente secção.

Enquanto estudante do Mestrado em Engenharia Informática - Cibersegurança e Administração de Sistemas do Instituto Superior de Engenharia do Porto, subscrevo o código de conduta do Instituto Politécnico do Porto (Instituto Politécnico do Porto 2020). O artigo 8º do respetivo código encontra-se expressamente aderido pela Declaração de Integridade presente no documento, visando reforçar a consideração pela adoção da conduta apropriada no

desenvolvimento do projeto de dissertação. O artigo 6º, relatando os deveres dos estudantes do mencionado instituto, é tido em conta e obedecido, no qual se realça a alínea n) que expressa que ações constituem uma conduta ilícita no decorrente âmbito académico, mais concretamente os pontos 2.8 a 2.12, que expressam a ilicitude do plágio nas suas diversas formas. De modo a obedecer a este princípio fundamental, todo o trabalho de outrem será detidamente referido ou citado, dependendo do contexto em que o conteúdo desse mesmo trabalho se insere no documento atual.

Embora não seja membro da ACM, o "ACM Code of Ethics and Professional Conduct"(ACM 2024) é subscrito na realização de todas as fases que constituem o atual projeto. Nomeadamente, os princípios de honestidade e integridade são estritamente obedecidos no decorrer do projeto, relevando-se também o respeito pelo princípio de respeitar o trabalho alheio, utilizado para o desenvolvimento da nova solução que se visa produzir nesta dissertação.

Tratando-se de um projeto cuja área de atuação é a segurança informática, e acrescendo o facto de que serão realizados testes para avaliar a eficácia da solução concebida, é necessário refletir nas implicações éticas que este processo de avaliação despoleta. Como tal, os princípios de respeitar a privacidade e honrar a confidencialidade mencionados no código de ética da ACM serão, também, obedecidos.

1.4 Estrutura do documento

O presente documento encontra-se organizado em quatro capítulos, que serão agora descritos.

O capítulo de Introdução apresenta o tema da dissertação, descrevendo o problema que se pretende resolver e o contexto no qual ele se insere. Com base nesse problema, são definidos objetivos para o projeto. Adicionalmente, também se descrevem algumas considerações éticas tidas em conta no desenvolvimento da dissertação.

De seguida, o capítulo de Estado da Arte apresenta a metodologia de revisão de literatura adotada, bem como os resultados obtidos. Estes são analisados, considerando-se a sua contribuição para o projeto. É também apresentada uma secção introdutória, que estabelece os conceitos-chave da dissertação.

Segue-se o capítulo de Arquitetura e Desenho de Solução, apresentando-se as decisões arquiteturais tomadas para cumprir com os objetivos definidos. Exploram-se também as decisões tecnológicas do projeto, definindo que tecnologias são utilizadas para cumprir que propósitos.

Proposta a arquitetura, foi realizado o desenvolvimento da solução, sendo este descrita no capítulo de Implementação. Neste, também são descritos os testes conduzidos para avaliar a eficácia da prova de conceito desenvolvida.

Finalmente, são apresentadas as conclusões obtidas pela realização do presente projeto, descrevendo as contribuições do mesmo, as limitações, e delineando o trabalho futuro.

Capítulo 2

Estado da Arte

Tendo extraído a definição do problema e dos seus objetivos, torna-se necessário realizar um estudo relativamente ao estado da arte, que contém o conhecimento fundamental necessário para a conceção de uma solução ao problema apresentado. Como tal, serão definidas as Research Questions (RQ) que permitirão conduzir a investigação necessária. A partir destas RQs, serão formuladas *queries* para a pesquisa nas bases de dados de literaturas selecionadas. Obtida esta informação, será feita uma filtragem posterior dos documentos encontrados, de modo a agregar aqueles que sejam relevantes para a resposta às questões colocadas.

Realizada esta fase de pesquisa, serão apresentados os trabalhos relacionados - obtidos através do processo de revisão de literatura - bem como algumas tecnologias relevantes para o projeto, explicando em que consistem e como podem auxiliar na resposta ao problema vigente.

Contudo, revela-se a necessidade de introduzir algum contexto relativamente às tecnologias sob as quais assenta o problema apresentado, lacuna essa que vem ser preenchida pela seguinte secção.

2.1 Preâmbulo

Originalmente desenvolvido pela Google em 2014, o Kubernetes é uma ferramenta open-source para orquestração de *containers*, automatizando o seu *deployment*, escalabilidade e gestão (Huang e Jumde 2020). No seu cerne reside o contexto de *cluster*, que nada mais é do que um conjunto de nós computacionais que disponibilizam recursos de memória, armazenamento e rede. Estes nós - *nodes* em Kubernetes - são máquinas singulares, podendo ser físicas ou virtuais, cuja responsabilidade é a execução de Pods. O Pod é "a unidade de trabalho em Kubernetes"(Huang e Jumde 2020), e pode ser constituído por um ou mais *containers*.

A gestão do estado do *cluster* é realizada pelo Control Plane, sendo este um conjunto de componentes utilizados para esse efeito (The Linux Foundation 2024b). Desses componentes, vale realçar a importância de dois deles. O API Server é o componente do Control Plane que expõe a API do Kubernetes, sendo esta "o *front end* para o Control Plane"(The Linux Foundation 2024a). Este componente é de elevada importância, pois é através dele que utilizadores, ferramentas de automatização e até mesmo componentes aplicacionais do sistema interagem com o Control Plane. Já o etcd trata-se de uma base de dados utilizada pelo Kubernetes para o armazenamento de todos os dados do *cluster*. A arquitetura deste sistema é apresentada na figura 2.1.

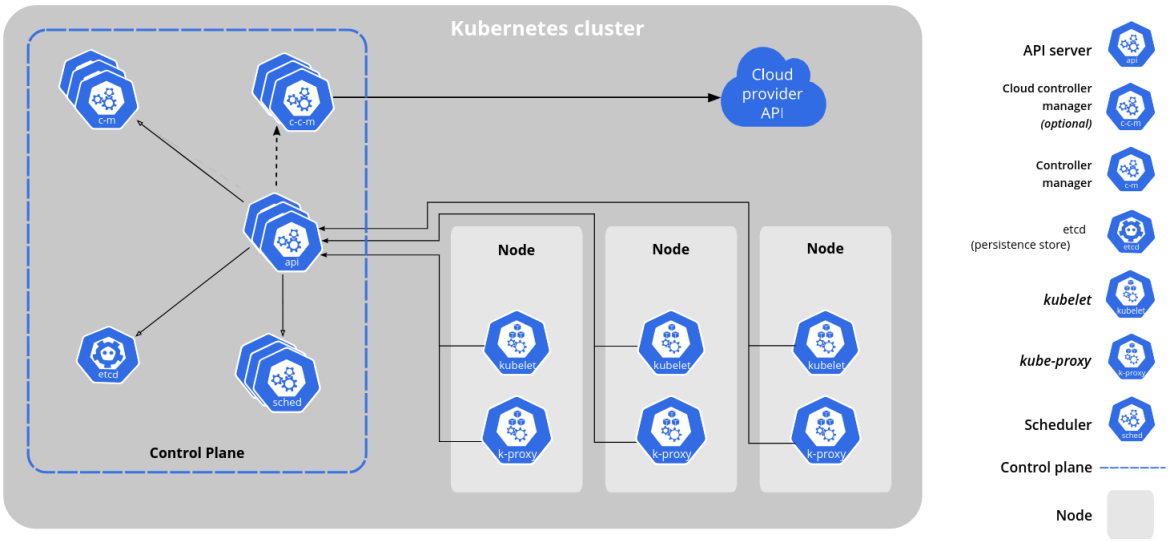

Figura 2.1: Arquitetura do Kubernetes (The Linux Foundation 2024a)

Um dos tipos de objetos do Kubernetes que são armazenados no etcd são os Secrets. Estes contém dados sensíveis como palavras-passe, *tokens* ou chaves de encriptação (The Linux Foundation 2024c). A sua principal função é evitar a inclusão de segredos no código aplicacional, oferecendo para isso uma forma de os externalizar deste.

Apresentado este contexto, estão definidos os conceitos-chave sob os quais assenta o tema da dissertação atual. Como tal, prosseguir-se-à para uma descrição da metodologia de pesquisa adotado. Não obstante, recomenda-se a leitura da documentação do Kubernetes para um melhor entendimento dos seus conceitos técnicos e respetivo funcionamento [1] [2].

2.2 Metodologia de Pesquisa

No âmbito da atual revisão de literatura, foi adotada uma metodologia de pesquisa sistemática, mais concretamente do tipo *mapping study*. Como tal, o problema foi analisado de modo a ser possível definir um conjunto de Research Questions, que guiarão a recolha de informação de modo a obter referências relevantes para o tema em análise. De modo a reduzir o volume de dados obtidos, foram definidos um conjunto de critérios de inclusão e exclusão - expostos nas tabelas 2.2 e 2.3, respetivamente. Para algumas das referências encontradas, foi também adotada uma metodologia de *snowballing* (Wohlin 2014), em que as referências indicadas em alguns dos artigos obtidos foram analisadas. Esta análise foi feita para casos em que uma dada referência demonstra ser interessante para o projeto em questão, nomeadamente pelo conteúdo em que esta se encontra citada, como também pelo seu título e, numa fase posterior, pela leitura do resumo dessa referência.

O volume de dados obtido na fase de pesquisa figura mais de 500 artigos, sendo seguro concluir que uma análise detalhada de todos estes inviabiliza a conclusão do presente projeto em tempo útil. Torna-se, então, crucial definir que volume de dados constitui uma amostra significativa cujo conteúdo é relevante para o processo de revisão de literatura. Desse modo, definiu-se o requisito de apresentar não menos do que 10 referências, nem mais do que

[1] Arquitetura do Kubernetes: https://kubernetes.io/docs/concepts/architecture/
[2] Principais componentes: https://kubernetes.io/docs/concepts/overview/components/

40. Esta gama de valores permite a agregação de conhecimento essencial para o posterior desenvolvimento da dissertação, mantendo o âmbito desta como algo exequível.

2.3 Revisão de Literatura

Para o presente projeto, foi realizada uma revisão sistemática da literatura, para a qual foram definidas as seguintes RQs:

- **RQ1:** Existem técnicas e ferramentas que otimizem a segurança dos dados armazenados em volumes persistentes e na base de dados (etcd) de clusters Kubernetes, operando em infraestruturas bare-metal?
  - **Explicação:** Dado que um dos objetivos assenta na proteção de volumes persistentes e da base de dados etcd, é necessário investigar que técnicas e ferramentas existem para a otimização da segurança dos dados guardados nestes meios. Adicionalmente, visto tratar-se de uma infraestrutura bare-metal, é relevante investigar a aplicabilidade destas técnicas e ferramentas neste tipo de infraestrutura.
- **RQ2:** Quais as práticas mais eficazes para implementar Role-Based Access Control (RBAC) e Multi-Factor Authentication (MFA) de modo a melhorar a segurança e gestão de identidade em Kubernetes?
  - **Explicação:** Procura-se investigar que práticas e mecanismos existem para a implementação de RBAC e MFA em ambientes Kubernetes, bem como estudar que impacto estes terão na postura de segurança e gestão de identidade. Devem ser estudadas as práticas sob as quais se podem restringir as operações realizadas por desenvolvedores sob este tipo de ambientes, de modo a obedecer ao princípio do privilégio mínimo.
- **RQ3:** Quais ferramentas de deteção e resposta a ameaças (IDR) estão disponíveis para ambientes Kubernetes e como podem ser utilizadas para identificar e mitigar incidentes de segurança?
  - **Explicação:** A deteção e resposta a ameaças é um componente fulcral para a solução desenvolvida, constituindo um componente significativo da solução integrada que se pretende produzir, pelo que é imperativo investigar que ferramentas existem cuja atuação incida nesta vertente. Com esta questão, o objetivo é recolher um conjunto de ferramentas que possam, posteriormente, ser conectadas de modo a construir uma solução integrada, recorrendo às informações de cada uma e oferecendo uma perspetiva mais enriquecedora sobre potenciais incidentes de segurança que ocorram no ambiente.

Face a esta definição, foram testadas várias *queries* de pesquisa nas bases de dados científicas escolhidas, sendo estas: B-On, Google Scholar, ACM Digital Library. Estes termos de pesquisa foram incrementalmente refinados de modo a apresentar resultados mais relevantes ao assunto atualmente em estudo. No final, foram utilizadas 3 *queries* de pesquisa, que são apresentadas na Tabela 2.1.

Tabela 2.1: Queries utilizadas

ID	Expressão utilizada
Q1	"kubernetes"AND ("persistent volume"OR "etcd") AND "encryption"AND ("techniques"OR "practices"OR "technologies"OR "configurations")
Q2	"kubernetes"AND ("Role-Based Access Control"OR "RBAC") AND ("user permissions"OR "user access"OR "team access"OR "user-specific configuration") AND ("Multi-Factor Authentication"OR "MFA") AND ("identity management"OR "security practices"OR "kubeconfig permissions")
Q3	"kubernetes"AND ("threat detection"OR "incident response"OR "intrusion detection") AND ("IDR tools"OR "security tools"OR "monitoring tools") AND ("identify security incidents"OR "mitigate security incidents"OR "threat mitigation")

Devido à quantidade de dados retornados pelas *queries* desenvolvidas, foram aplicados filtros para restringir os resultados aos mais relevantes para esta pesquisa na fase de triagem. Estes filtros consistem em:

1. **Data de publicação:** Dado o caráter recente do Kubernetes enquanto tecnologia, bem como dos seus desenvolvimentos relevantes para a área de segurança e proteção de dados, os resultados foram restringidos para que apenas sejam apresentados artigos lançados em anos posteriores a 2020, inclusive;
2. **Linguagem:** Para auxiliar a compreensão do material recolhido, foram apenas considerados resultados em Português e em Inglês;
3. **Formato:** Os resultados foram limitados a artigos, livros, publicações para conferências e teses;
4. **Relevância:** O conteúdo do material recolhido na pesquisa deve ser relevante para o tema em análise, isto é, deve mencionar pelo menos uma tecnologia, prática ou abordagem geral relativa às múltiplas vertentes abordadas na definição do problema, tais como encriptação de dados, RBAC e deteção e resposta a incidentes.

No caso da data de publicação, foi escolhido o ano de 2020 como limite inferior pois este reflete a consolidação do Kubernetes como uma tecnologia amplamente adotada e nas necessidades regulamentares crescentes relacionadas à segurança de dados. Este critério permitiu a inclusão de referências desenvolvidas com base num conhecimento solidificado do Kubernetes e as suas complexidades, bem como também contempla artigos mais recentes com desenvolvimentos ainda pouco divulgados, porém relevantes para o tema em análise. Realça-se a ausência do número de citações como filtro para a inclusão ou exclusão de referências, pois isso poderia excluir publicações recentes que, devido à sua publicação recente, ainda não alcançaram ampla disseminação na comunidade académica.

Relativamente à aplicação destes critérios, esta foi realizada maioritariamente através da observação direta das referências. Ao visualizar os dados de cada referência, é possível rapidamente avaliar o formato da mesma, a linguagem em que se encontra escrita, e a sua data de publicação. Adicionalmente, a relevância foi avaliada numa fase inicial quer pelo título, quer pela leitura do resumo.

Com base nesta informação, foram definidos os critérios de inclusão e exclusão de literatura, que são aplicados na fase de filtragem da literatura obtida. Estes critérios de inclusão e exclusão são apresentados nas Tabelas 2.2 e 2.3, respetivamente.

Tabela 2.2: Critérios de Inclusão

ID	**Critério**
CI1	A fonte é posterior a 2020
CI2	A fonte encontra-se escrita em português ou inglês
CI3	A fonte constitui um artigo, livro, publicação para conferência, ou tese
CI4	A fonte menciona desenvolvimentos, tecnologias ou práticas que enderecem a encriptação de dados, RBAC, deteção e resposta a incidentes

Tabela 2.3: Critérios de Exclusão

ID	**Critério**
CE1	A fonte é anterior a 2020
CE2	A fonte não se encontra escrita em português ou inglês
CE3	A fonte não se trata de um artigo, livro, publicação para conferência, ou tese
CE4	A fonte não menciona desenvolvimentos, tecnologias ou práticas de encriptação de dados, RBAC, deteção e resposta a incidentes

Através destes critérios, os resultados obtidos foram filtrados de modo a obter apenas aqueles que são relevantes para o atual estudo. Através deste processo de recolha de dados e a sua posterior filtragem filtragem, foi obtido um conjunto de 17 referências. Este resultado encontra-se encapsulado na definição de amostra significativa apresentada na secção 2.2. Como tal, esses serão explorados na secção seguinte, avaliando-se detalhadamente a sua relevância, o potencial contributo para a resolução do problema e as lacunas que se propõem preencher no contexto desta dissertação.

2.4 Trabalhos Relacionados

Como ponto de partida para a restante investigação, os autores Islam Shamim, Ahamed Bhuiyan e Rahman 2020 apresentam uma sistematização de conhecimento relativamente a práticas de segurança em ambientes Kubernetes. Inicialmente, é estabelecido o contexto da adoção crescente do Kubernetes por grandes organizações como a IBM, Capital One e Adidas, transitando-se para uma exploração da preocupação demonstrada pelos administradores relativamente à segurança em ambientes Kubernetes (Islam Shamim, Ahamed Bhuiyan e Rahman 2020). Estes factos encontram-se alinhados com o contexto estabelecido em 1.1, solidificando a necessidade da solução que a atual dissertação visa desenvolver. De modo a consolidar as principais práticas de segurança reportadas pelos administradores de ambientes Kubernetes, os autores conduziram uma revisão da *grey literature*, analisando um conjunto de 104 artefactos obtidos da Internet. Através desta análise, foi possível catalogar as onze práticas mais reportadas pelos administradores, sendo estas (Islam Shamim, Ahamed Bhuiyan e Rahman 2020):

1. Autenticação e Autorização;
2. Implementação de políticas específicas ao Kubernetes;

3. Análise de vulnerabilidades;
4. *Logging*;
5. Separação de *namespaces*;
6. Encriptação do etcd e restrição ao seu acesso;
7. Atualização contínua;
8. Limitação de cotas de CPU e memória;
9. Configuração de suporte para SSL/TLS;
10. Separação de *workload* sensível;
11. Segurança do acesso aos metadados.

É notável que várias destas práticas são diretamente mencionadas na definição do problema que a presente dissertação visa resolver. Para cada uma destas atividades, os autores listam um conjunto de medidas frequentemente reportadas pelos administradores para endereçar potenciais falhas de segurança. Torna-se, então, relevante sistematizar as que visam áreas da segurança em Kubernetes contempladas pelo projeto atual. Note-se que nem todas constam no âmbito deste, visto que esta inclusão impossibilitaria a conclusão do projeto em tempo útil. Não obstante, são considerações relevantes que serão tidas em conta no decorrer da análise e desenvolvimento.

2.4.1 Autenticação e Autorização

Em Kubernetes, o procedimento para comunicação das entidades com a sua API transita entre três fases: autenticação, autorização e admissão. Este processo encontra-se evidenciado na Figura 2.2.

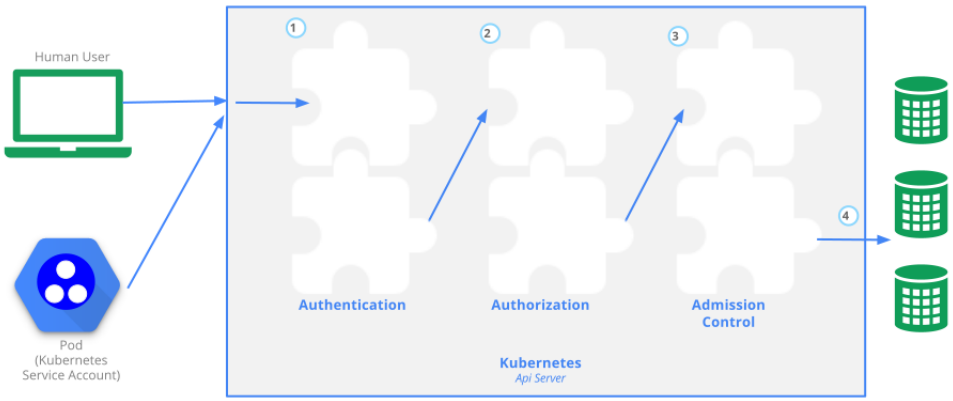

Figura 2.2: Controlo de acessos em Kubernetes (The Linux Foundation 2023)

No caso da autenticação, é recomendada a desabilitação de acesso anónimo à API do Kubernetes, obrigando a que todas as entidades que pretendam interagir com esta tenham que ser

previamente autenticadas e autorizadas (Islam Shamim, Ahamed Bhuiyan e Rahman 2020). Conclui-se com uma menção ao protocolo OpenID, recomendado pelos administradores para a autenticação perante a API do Kubernetes. Trata-se de uma configuração nativamente suportada por esta, ainda que necessite de um provedor de identidade externo, como referido por Creane e Gupta 2021. É relevante, no âmbito do projeto atual, realizar esta integração para garantir uma gestão de identidades mais restrita, proveniente de um serviço de identidade central, e como tal diminuir a superfície de ataque à API do Kubernetes. Não só é possível a configuração com um provedor como a Google, como alternativamente também poderá ser utilizado um provedor de identidade privado, hospedado pela organização (Creane e Gupta 2021). Como tal, esta decisão deve ser ponderada na fase de *design*.

Após a autenticação, identificando-se a entidade que pretende comunicar com a API do Kubernetes, é imperativo que ocorra um processo de autorização, avaliando se a entidade está permitida, de acordo com as políticas organizacionais, a executar a ação que pretende. Este processo é suportado pela configuração de controlo de acessos ao nível da API do Kubernetes, como apresentado por Sayfan 2020. Notavelmente, a API suporta a configuração de `authorization-mode` para Attribute-Based Access Control (ABAC) ou RBAC, entre outros métodos. O sistema ABAC recorre a um ficheiro local onde as políticas estão definidas, enquanto que para o RBAC as políticas de autorização encontram-se armazenadas e geridas pela API (Sayfan 2020).

Em Huang e Jumde 2020 é realizada uma apresentação teórica e técnica de como o RBAC do Kubernetes pode ser configurado e operado. Cada política de RBAC deve identificar: o *subject* a quem será aplicada a política; os recursos aos quais a política diz respeito; e os *verbs* que identificam as ações a serem realizadas sob esses recursos (Huang e Jumde 2020). Note-se que o *subject* pode ser um utilizador - cuja identidade é obtida de uma fonte externa, como explorado anteriormente - como também pode ser uma *Service Account*, cuja utilização é realizada por Pods para que estes se possam autenticar perante a API e conduzir as ações necessárias, de acordo com a função do Pod (Huang e Jumde 2020).

No âmbito do projeto, é essencial analisar que *Roles* devem ser definidas e com que permissões, algo que será contemplado na fase de *design*. Deve também ser analisada a possibilidade de integrar esta atividade de definição de políticas para RBAC com os restantes componentes da solução, oferecendo uma solução completa de gestão de identidade.

2.4.2 Implementação de políticas

Relativamente à implementação de políticas, estas são categorizadas por Islam Shamim, Ahamed Bhuiyan e Rahman 2020 em três grupos: políticas de rede, de Pods e genéricas. Para políticas de rede, recomenda-se a restrição de tráfego entre Pods e para com o API Server do Kubernetes, de modo a reduzir a superfície de ataque na rede (Islam Shamim, Ahamed Bhuiyan e Rahman 2020). Enfatiza-se também a importância de reduzir a exposição do *cluster* à rede, limitando a sua superfície de ataque. Creane e Gupta 2021 exploram extensivamente o funcionamento de `NetworkPolicies`, que definem como sendo a "principal ferramenta para garantir a segurança de uma rede em Kubernetes"(Creane e Gupta 2021). Por defeito, todos os Pods podem livremente comunicar entre si, comportamento esse que deve ser repensado num contexto organizacional de modo a segregar *workloads* distintos, que não necessitem desta comunicação. Como tal, estas políticas oferecem uma solução de restrição e segregação de tráfego que acompanha a natureza dinâmica e escalável do Kubernetes, fazendo-o sem necessitar que os programadores e administradores possuam conhecimento de detalhes de *network* (Creane e Gupta 2021).

O estudo conduzido por Budigiri et al. 2021 acresce validade ao uso de políticas de rede num contexto empresarial. Neste, os autores compararam a *performance* de vários *plugins* para Container Network Interface (CNI) - um componente necessário para implementar *networking* em Kubernetes, sendo estes Calico e Cilium. Comparando os CNI num ambiente de aplicações 5G, com necessidade de baixa latência, concluíram que a utilização de recursos `NetworkPolicy` não introduz um *overhead* significante no desempenho do *cluster* (Budigiri et al. 2021). Aferiram ainda que comparando o Calico com o Cilium, ambos utilizando eBPF, o primeiro oferece uma resposta mais rápida aos pedidos, celeridade essa mais notada na comunicação entre nós computacionais.

Note-se que a definição de políticas de rede em Kubernetes é, ainda assim, um processo complexo no qual existem vulnerabilidades e ameaças. O design destas deve ser ponderado e realizado recorrendo ao princípio do privilégio mínimo (Budigiri et al. 2021). De modo a reduzir o risco de configuração indevida, existem ferramentas como o OPA Gatekeeper que auxiliam nesta tarefa, ainda que não garantam por si só o isolamento de rede, mas sim validem a sintaxe e presença destas políticas no *cluster* (Budigiri et al. 2021).

No âmbito da presente dissertação, a aplicação de políticas de rede mostra-se como sendo um requisito, pelo que esta informação deve ser ponderada. Nomeadamente, qual CNI utilizar e como definir as referidas políticas de modo integrado com as restantes valências da segurança em Kubernetes.

Para as políticas de Pods, recomenda-se a definição de um contexto de segurança para as mesmas, impedindo que estas sejam executadas com privilégio de `root` e com permissão de escrita para o sistema de ficheiros sob o qual executam. Finalmente, as políticas genéricas visam a proteção dos componentes do Kubernetes perante utilizadores externos maliciosos. Realça-se a necessidade de autenticação perante estes componentes; bem como a aplicação do princípio do privilégio mínimo para os utilizadores que tenham acesso a estes ambientes (Islam Shamim, Ahamed Bhuiyan e Rahman 2020). Apresenta-se também o conceito de *audit policy*, configurável ao nível da API do Kubernetes que deve ser analisado na fase de *design*, pois este pode produzir informação valiosa que deva constar da solução integrada a desenvolver.

Em Huang e Jumde 2020, afirma-se que uma *audit policy* permite aos administradores a definição de regras sobre "que tipos de eventos devem ser gravados e quanto detalhe dos mesmos deve ser incluído"(Huang e Jumde 2020). De modo a auxiliar esta implementação, são apresentados exemplos de recursos do tipo `Policy`, bem como a configuração necessária da API para que esta interprete as políticas e as aplique, registando os eventos no *backend* desejado(Huang e Jumde 2020). Como tal, a implementação destas políticas deve ser estudada no âmbito do presente projeto, aferindo se estas trarão valor à solução integrada que se pretende desenvolver. Note-se que, como referido por Martin e Hausenblas 2021, que existem alternativas que expandem a funcionalidade destas políticas, acrescentando outro tipo de funcionalidades e *backends* para o envio da informação.

Abordando o tema das políticas em Kubernetes como um todo, deve considerar-se o trabalho de Vugt e Malik 2023, em que os autores fazem uma análise crítica das funcionalidades de segurança existentes de forma nativa no Kubernetes, tais como Admission Controllers, que validam e modificam pedidos enviados à API de controlo, bem como Pod Security Standards que visam garantir a configurabilidade de segurança por Pod. Estas funcionalidades, ainda que valiosas, consideram apenas questões de segurança ao nível dos Pods, não endereçando a segurança do cluster como um todo nem a segurança das comunicações de rede (Vugt e

Malik 2023). Desse modo, os autores apresentam um conjunto de ferramentas de observabilidade de segurança, com o objetivo de estudar o seu impacto na *performance* do sistema como um todo. As ferramentas analisadas foram: Falco (quer usando o driver de kernel, quer usando eBPF); Tetragon; KubeArmor e Tracee. No ambiente de testes, o Prometheus e Grafana foram usados com o objetivo de recolha de métricas, bem como a sua apresentação gráfica. Foi possível concluir que algumas das ferramentas (Falco e Tetragon) apresentaram um desvio de consumo de recursos menor em comparação ao nível base utilizado nos testes, sugerindo que "[realizam uma] gestão de recursos mais eficiente ou focam-se em medidas de segurança mais leves que exercem menos esforço nos recursos do sistema"(Vugt e Malik 2023). Esta análise é concluída com a afirmação de que estas ferramentas, individualmente, são limitadas no seu foco, pelo que uma abordagem com múltiplas camadas deve ser adotada. Esse pensamento é congruente com o problema que originou o atual projeto, pelo que deve ser considerada não só a utilidade destas ferramentas no contexto do problema, mas também o modo como seriam integradas como parte de uma abordagem de segurança holística.

Relativamente às ferramentas apresentadas anteriormente, o trabalho de German e Ponomareva 2023 oferece um contributo relevante. Os autores realizaram uma comparação de várias ferramentas para segurança de *containers*, das quais consta o Falco, bem como novas alternativas: Prisma Cloud, Aqua Security, NeuVector, e Sysdig Secure. Destas, as únicas *open-source* são o Falco e NeuVector, pelo que as restantes não poderão ser contempladas no projeto. Contudo, a análise realizada revela que o NeuVector supera o Falco no que toca ao número de funcionalidades, pois também contempla configurações para monitorização de *compliance*, análise de vulnerabilidades e proteção em tempo de execução (German e Ponomareva 2023). O contributo de German e Ponomareva 2023 é, então, importante para a atual dissertação, pois apresenta uma ferramenta com potencial para endereçar várias das vertentes especificadas no problema e objetivos.

Partindo da menção de Admission Controllers em Vugt e Malik 2023, é relevante analisar o trabalho de Slik e Wiersma 2021, onde os autores analisam a possibilidade da substituição do sistema de Pod Security Policy (PSP) - cuja utilização foi deprecada em 2021 (Sable 2021) - por Admission Controllers externos. Inicialmente, argumentam que o sistema de PSP é considerado confuso pela comunidade, bem como carece de uma funcionalidade de *audit*, que auxiliaria os administradores na definição de políticas de controlo para a admissão de Pods (Slik e Wiersma 2021). Como tal, foram escolhidos três Admission Controllers desenvolvidos pela comunidade para o estudo: OPA Gatekeeper; Kyverno; e k-rail. Neste projeto de investigação, os autores concluíram que o Gatekeeper e Kyverno são capazes de contemplar a grande maioria das funcionalidades presentes no PSP, enquanto que o k-rail não é capaz de o fazer, à data de escrita do artigo. Acrescenta-se também que estas soluções alternativas oferecem um modo de *audit*, constituindo imediatamente uma vantagem face ao PSP. A diferença mais notável entre os Admission Controllers analisados e o PSP é o facto de estes ainda não suportarem operações de mutação, encontrando-se esta funcionalidade numa fase experimental e de desenvolvimento (Slik e Wiersma 2021). Contudo, os autores alegam que isto não figura uma preocupação no que toca à segurança, visto que a possibilidade de validar a admissão de recursos bloqueando configurações vulneráveis minimiza a superfície de ataque de um *cluster*. Adicionam ainda que os Admission Controllers alternativos, contrariamente ao PSP, oferecem funcionalidades de validação de outros recursos para além de Pods, algo que permitirá uma diminuição ainda maior da superfície de ataque. No âmbito do presente projeto, a análise destas ferramentas é de elevado interesse,

contemplando também a sua integração com os restantes componentes da solução desenvolvida. Adicionalmente, será relevante realizar uma investigação mais exaustiva do referido modo de *audit* disponibilizado por estas ferramentas. Com este modo, pode ser possível oferecer um modo mais ágil aos administradores de avaliarem a eficácia das políticas que definem, incorporando os *logs* emitidos por este modo de *audit* com o restante da solução, criando-se então um fluxo de eventos cuja interpretação habilitará ao administrador um melhor aprimoramento destas políticas. Por último, também se pretende oferecer um método de visualização destes eventos como parte de uma solução de deteção de incidentes.

No artigo de Ali et al. 2024, os autores apresentam as medidas que adotaram para melhorar a postura de segurança do seu *cluster* Kubernetes num contexto prático e organizacional. Para esse efeito, adotaram um conjunto de novas tecnologias e práticas: implementação de Network Policies; a incorporação do OPA Gatekeeper; e a integração do Hashicorp Vault com o *cluster* (Ali et al. 2024). No caso das Network Policies, estas permitem aos administradores realizar uma filtragem detalhada de entidades de rede com as quais um Pod pode comunicar, sendo possível restringir quer o tráfico de *input*, quer o de *output*. Para esse efeito, os autores utilizaram o Calico como CNI, que permite a definição de políticas de rede mais granulares, neste caso sendo realizada uma filtragem por Pod e por namespace. Assim, apenas os serviços de *frontend* podem comunicar com os de *backend*, bloqueando-se comunicações quer com outros namespaces, quer com Pods não permitidos, mesmo que estes se encontrem em namespaces autorizados. No que toca ao Gatekeeper, os autores utilizaram este Admission Controller para restringir as imagens de *containers* que podem ser executadas no *cluster*. Mais concretamente, apenas imagens provenientes do registo do CERN podem ser executadas no ambiente Kubernetes em questão. Adicionalmente, esta tecnologia foi utilizada para assegurar que todos os Pods executados no cluster têm definidas *readiness* e *liveness probes*, forçando os administradores a garantir a confiabilidade do serviço que pretendem implementar no *cluster* (Ali et al. 2024). Finalmente, o Vault foi utilizado para injetar credenciais em Pods autorizados para as aceder. Os autores relatam as vantagens desta abordagem, consistindo no desacoplamento de segredos e serviços, melhorando a escalabilidade, bem como a existência de um ponto centralizado para gestão destes segredos. Estas práticas respondem a lacunas apresentadas no artigo de Vugt e Malik 2023, porém, por si só também constituem uma solução incompleta de segurança. É notória a melhoria que estas práticas trazem a um ambiente Kubernetes, mas a observabilidade não é endereçada por elas. Como tal, é do interesse da presente dissertação ponderar como estas práticas de Network Policies, Admission Controllers e gestão centralizada de segredos podem ser integradas de forma coesa com as tecnologias de observabilidade apresentadas por Vugt e Malik 2023.

2.4.3 Análise de Vulnerabilidades

A análise de vulnerabilidades também é relevante para o projeto atual, visto que se trata de uma prática essencial para assumir uma postura de segurança robusta. Se um *cluster* Kubernetes executar componentes com vulnerabilidades, então "todo o sistema de orquestração, e as aplicações aprovisionadas, tornam-se suscetíveis a ataques"(Islam Shamim, Ahamed Bhuiyan e Rahman 2020). Como tal, os autores recomendam a utilização de ferramentas para análise de *containers* tais como o Dockscan e CoreOS Clair, existindo, no entanto, outras alternativas. Adicionalmente, aconselha-se a utilização de um registo de imagens privado a partir do qual as imagens executadas no *cluster* sejam obtidas. Indo de encontro a esta recomendação, Russell e Dev 2024 desenvolveram um sistema baseado em ferramentas *open-source* que deteta configurações incorretas, bem como vulnerabilidades em *containers*

que estejam a ser executados em Kubernetes, armazenando esta informação num sistema centralizado de *logging*. É de particular interesse para este projeto a integração, por parte dos autores, de valências distintas da segurança em Kubernetes - agregando informação de fontes distintas no mesmo ponto central. Para a deteção de vulnerabilidades, o Trivy foi utilizado, enquanto que para a deteção de configurações incorretas ou não conformes com as boas práticas foram utilizados o Kubesec, Kube-score, Kubeaudit e KubeLinter. Note-se, no entanto, a ausência nesta solução dos outros pontos referidos previamente, sendo essa expansão um dos fatores motivadores desta dissertação.

2.4.4 Logging

Em Islam Shamim, Ahamed Bhuiyan e Rahman 2020, este componente é definido como sendo a prática de "ativar e monitorizar *logs* do *cluster* Kubernetes"(Islam Shamim, Ahamed Bhuiyan e Rahman 2020). Estes logs devem contemplar as aplicações, os containers dentro de cada Pod e o *cluster* como um todo. Para esse efeito, os administradores recomendam a monitorização regular dos *logs*, bem como a implementação de mecanismos de alarmística caso ocorra uma mudança significativa nos padrões de *logs* previamente identificados (Islam Shamim, Ahamed Bhuiyan e Rahman 2020).

A importância desta prática também é realçada em Kampa 2024, que recomenda a configuração de sistemas de monitorização "que capturem e analisem uma vasta gama de métricas, eventos, e *logs* em todo o *cluster*"(Kampa 2024). Com este sistema, os administradores podem obter informação detalhada que permite aferir a postura de segurança de um *cluster*, bem como ter evidências necessárias na eventualidade de uma auditoria ou resposta forense a um incidente (Kampa 2024).

Uma solução de *logging* que cumpre as especificações apresentadas foi desenvolvida pelos autores de Sukhija et al. 2020. Nesta, um conjunto de ferramentas *open-source* - Prometheus, Grafana, Kafka e ServiceNow - foram utilizadas para expandir a solução existente no ambiente dos autores. Essa solução inclui uma implementação escalável de Elasticsearch para providenciar tendências históricas, bem como visualizações cujo objetivo é oferecer conhecimentos organizacionais e auxiliar a tomada de decisão (Sukhija et al. 2020).

Como tal, é crucial analisar esta prática no *design* da solução para o problema apresentado. Todos os eventos relacionados com segurança, sejam gerados pelo RBAC, políticas de rede, controlo de admissão ou qualquer outro componente do seu âmbito, devem ser registados. O *logging* será um tema central da solução, e terá um papel fulcral na integração dos diferentes componentes da solução. As tecnologias apresentadas por Sukhija et al. 2020 devem ser estudadas, bem como potenciais alternativas a estas. Há também que considerar a apresentação dos dados provenientes desta recolha de *logs* aos administradores que usem a solução, fazendo-o de modo a que os gráficos e históricos gerados sejam úteis na deteção e resposta a incidentes.

2.4.5 Encriptação do etcd e restrição ao seu acesso

No livro Gkatziouras et al. 2024, são apresentadas várias considerações relativamente à gestão de Secrets em Kubernetes. Trata-se de um tema de importância crucial para o presente projeto, pois estes são frequentemente utilizados para o armazenamento de credenciais, que serão posteriormente utilizadas pelas aplicações que são executadas num ambiente Kubernetes. Os Secrets são escritos na base de dados etcd, e por defeito não são persistidos de forma criptografada, mas sim codificados em Base64 (Gkatziouras et al. 2024), algo

bastante fácil de descodificar. Como alternativa a esta prática, o Kubernetes oferece de modo nativo vários métodos de encriptação dos dados persistidos no etcd, configuráveis através de recursos do tipo `EncryptionConfiguration`. Estes métodos podem ser categorizados em dois grupos: encriptação sem componentes externos; e encriptação com recurso a componentes externos. No primeiro caso, os recursos de `EncryptionConfiguration` são definidos incluindo a chave de encriptação utilizada, e existem vários algoritmos que podem ser utilizados para o efeito. Ora, analisando este método, é possível aferir que tem uma grande desvantagem: a presença da chave de encriptação - ainda que codificada em Base64 - num ficheiro em formato YAML, presente no sistema de ficheiros local de cada *control plane node* (Gkatziouras et al. 2024). Na eventualidade de um comprometimento do sistema de ficheiros de um destes nós, o atacante teria acesso à chave de encriptação, por consequência conseguindo extrair todos os segredos do etcd e revelar o seu texto claro correspondente. No âmbito do presente projeto, esta preocupação torna-se ainda mais relevante, tratando-se de um ambiente Kubernetes hospedado numa infraestrutura *bare-metal*, na qual os administradores têm frequentemente acesso direto aos nós computacionais que disponibilizam o *cluster*. Para colmatar esta falha, existem então os métodos que recorrem a componentes externos, notavelmente o `kmsv2`.

Trata-se de um sistema de encriptação mais complexo, porém com mais garantias de segurança, pois a chave de desencriptação não se encontra armazenada nos nós de *control plane*. Adicionalmente, o `kmsv2` introduziu mecanismos de *caching* de modo a realizar a desencriptação dos objetos em memória, reduzindo o número de consultas ao Key Management Service (KMS) externo (Gkatziouras et al. 2024). Contudo, vale realçar que a resiliência do KMS externo é uma consideração fulcral na implementação deste mecanismo, pelo que se deve garantir a sua disponibilidade, bem como a confidencialidade e integridade dos dados armazenados no âmbito deste processo.

Ainda em Gkatziouras et al. 2024, são mencionadas várias opções de implementação de um KMS externo, sendo maior parte delas ferramentas de plataformas *cloud*, algo cuja implementação num ambiente *bare-metal* é menos apropriada. Contudo, o Hashicorp Vault apresenta-se como solução para administradores deste tipo de infraestrutura, podendo ele próprio ser disponibilizado nesta. Como tal, é do interesse da presente dissertação a implementação desta encriptação recorrendo a um KMS externo, de modo a melhorar substancialmente a segurança dos segredos armazenados no *cluster*.

2.5 Tecnologias Relacionadas

Na secção anterior foram identificadas as tecnologias relevantes utilizadas nas referências obtidas. Desse modo, é importante analisá-las e identificar o seu potencial papel na arquitetura da solução.

Para além das identificadas pela revisão de literatura, existem outras tecnologias que, ainda que não tenham surgido neste processo, são conhecidas pelo autor. Este reconhecimento surge quer através de experiência laboral na área de conhecimento, quer através da presença em conferências sobre Kubernetes. Como tal, estas serão apresentadas de seguida, oferecendo-se uma visão geral sobre estas, bem como uma análise da sua importância no contexto do problema.

O permission-manager (SIGHUP 2024) é uma aplicação para Kubernetes que possibilita a gestão de utilizadores em RBAC de modo acessível e simples. Na interface gráfica, os

administradores podem criar novos utilizadores para o *cluster*, definir as suas permissões, e exportar os certificados de autenticação. Estes certificados são, consequentemente, distribuídos para os respetivos utilizadores. As permissões são atribuídas por *namespace*, para os quais existem os privilégios de leitura e de escrita. É uma ferramenta relevante como alternativa à federação de utilizadores a partir de um ponto externo, algo explorado nas referências obtidas. Contudo, nota-se a falta de granularidade no que toca à definição de permissões. Por exemplo, não existe a possibilidade de filtrar permissões por Pod, ou relativas a acesso a outros recursos do *cluster*, tais como Secrets ou volumes persistentes.

Para o componente de análise de vulnerabilidades, o kubebench (Aqua Security 2024a) é uma ferramenta complementar às restantes identificadas para análise de vulnerabilidades. O seu fator diferenciador é o facto de este validar a segurança face ao CIS Kubernetes Benchmark, que se trata de um conjunto de "guidelines para a configuração segura"(Center for Internet Security 2024) de um ambiente Kubernetes. Vale realçar, no entanto, que as suas funcionalidades estão incluídas no Trivy - tecnologia retornada pelas referências. Não obstante, a sua menção é importante, pois caso seja utilizada uma alternativa ao Trivy, que não inclua este componente, é possível implementá-lo como um recurso separado, preservando a funcionalidade desejada.

O conceito de *logging* foi explorado na secção anterior, tal como algumas ferramentas para recolha de *logs* e observabilidade. Contudo, nota-se a ausência de uma ferramenta que faça a recolha dos logs do *cluster* e os envie para um ponto central - como é o caso do Elasticsearch. Para esse efeito, o fluent-bit (Fluent 2024b) é extremamente útil. Trata-se de um processador de *logs* e métricas para sistemas Linux, que permite a recolha destes artefactos através de diferentes fontes (Fluent 2024b). É uma ferramenta bastante extensível, pois permite a escrita de filtros a serem aplicados aos dados, bem como vários *outputs* para armazenamento destes (Fluent 2024b). No contexto do projeto, esta ferramenta pode ser utilizada para recolher *logs* que ocorram no ambiente Kubernetes e os armazenar num ponto central. Estes *logs* podem originar dos próprios servidores que servem como infraestrutura para o ambiente, dos Pods executados, de eventos do Kubernetes, entre outras fontes. Sendo possível recolher e centralizar dados das diversas ferramentas de monitorização, é também exequível a exibição destes dados e métricas ao utilizador final. Assim, os administradores que usem a solução desenvolvida recorrem a uma visão unificada dos eventos e estatísticas de segurança processados por esta.

Visto que o Grafana foi retornado na revisão de literatura, é importante mencionar a tecnologia Grafana Loki no âmbito do *logging*. Esta trata-se de uma ferramenta para agregação de *logs* (Grafana 2024c), tal como o fluent-bit. O fator distintivo que leva à sua menção é a integração direta que tem com o Grafana, algo que é importante para a solução a desenvolver. Adicionalmente, Grafana 2024c menciona que é uma ferramenta "especialmente adequada para armazenar *logs* de Pods em Kubernetes". A importância desta afirmação é notável, visto tratar-se de um componente importante para a solução. Como tal, deve ser feita a comparação entre esta ferramenta e as restantes do respetivo âmbito, algo que será apresentado posteriormente.

Para a gestão de segredos, a única tecnologia retornada no âmbito da revisão de literatura foi o Hashicorp Vault, algo que se deve a vários factos. É tecnologia bastante utilizada - com mais de mil contribuidores e 31 mil membros da comunidade (Hashicorp 2024a). Adicionalmente, é uma solução implementável em infraestruturas *bare-metal*, por exemplo através de uma instalação em Docker (Hashicorp 2024b). O Vault é "uma ferramenta para

acessar segredos de forma segura"(Hashicorp 2024a), fazendo-o com controlo de acessos restrito, *audit logs* detalhados e armazenamento seguro dos segredos.

Uma vertente de segurança revelada na revisão de literatura é o *runtime enforcement*. Esta consiste em aplicar decisões com base nos resultados obtidos, prevenindo ameaças em tempo real (Cilium 2024b). Para esse efeito, foram retornadas apenas duas tecnologias: Tetragon e NeuVector. Estas apresentam, então, uma vantagem face às restantes tecnologias de observabilidade. Enquanto que uma ferramenta como o Falco, por exemplo, apenas consegue reportar ocorrências no *cluster*, uma aplicação com esta capacidade consegue detetar e bloquear ocorrências, mediante a configuração dos administradores. Note-se, no entanto, a complexidade acrescida deste tipo de configuração, pois algumas políticas de resposta a incidentes podem bloquear ações que são necessárias no contexto do ambiente produtivo.

No decorrer da secção anterior, não foi feita nenhuma referência ao método de instalação das tecnologias retornadas, pelo que é relevante apresentar uma alternativa para tal. O Helm é, então, uma tecnologia importante, visto ser "o gestor de pacotes para Kubernetes"(Helm 2024b). No seu contexto, os pacotes são intitulados de Charts, que são "o conjunto de informação necessária para criar uma instância de uma aplicação"(Helm 2024c). Cada Chart tem um ficheiro `values.yaml`, onde a configuração dos parâmetros da aplicação pode ser realizada (Helm 2024a). Muitas das tecnologias disponibilizam Helm Charts próprios, aos quais os administradores podem recorrer para instalar e configurar tecnologias no *cluster*. Como tal, a existência de um Chart beneficia as tecnologias retornadas, algo que deve ser ponderado aquando da sua análise.

Recolhidas as tecnologias, é necessário fazer uma avaliação das mesmas, de modo a aferir a sua aplicabilidade no contexto da solução. Para esse efeito, foram definidos alguns critérios que permitam realizar uma comparação entre as soluções:

- Dimensão da comunidade: número de pessoas que seguem a aplicação no GitHub - determinado pelo número de "*stars*". Quantos mais o tiverem feito, mais atrativa e utilizada é a tecnologia;
- Data de lançamento da versão mais recente: Quanto mais recente for esta data, mais ativo é o desenvolvimento da tecnologia. É um fator importante, pois atualizações e lançamentos regulares indicam um compromisso dos desenvolvedores para com a melhoria contínua do produto. A sua avaliação é feita vendo a data da *release* mais recente no GitHub;
- Impacto no desempenho: Quão elevado é o impacto no desempenho usando a tecnologia em análise, comparativamente a não a utilizar;
- Simplicidade da implementação: Quão simples é a instalação e configuração da tecnologia num ambiente Kubernetes.

Quanto a estes critérios, note-se que os dois primeiros são possíveis devido ao facto de as ferramentas estudadas serem *open-source*. Como tal, o seu repositório de desenvolvimento é público, e nele é possível extrair a informação necessária.

Estabelecidos estes critérios, é realizada uma avaliação das tecnologias. Dado o volume elevado de resultados retornados, a comparação é apresentada sob a estrutura de um conjunto de tabelas. Estas encontram-se separadas pela componente da segurança que abordam, estando esta devidamente identificada.

É importante realçar que algumas práticas identificadas estão excluídas desta representação, sendo estas: RBAC em Kubernetes; `NetworkPolicies`; *audit policies* e `kmsv2`. Esta exclusão deve-se ao facto de estas serem práticas nativamente suportadas no Kubernetes. A sua configuração é crucial, conforme relevado na secção anterior, porém são componentes *built-in*. Não se tratando de ferramentas externas ao Kubernetes, a sua inclusão nas referidas tabelas produziria um resultado pouco coerente.

A comparação entre as tecnologias, divididas de acordo com a sua função, apresenta-se nas tabelas 2.4, 2.5, 2.6 e 2.7.

Tabela 2.4: Ferramentas estudadas para análise de vulnerabilidades

Ferramenta	**Dimensão da comunidade**	**Data da versão mais recente**	**Impacto no desempenho**	**Simplicidade de implementação**
NeuVector (SUSE 2024a)	1122	16/11/2024	Sem impacto no modo Monitor, com impacto no modo Protect (SUSE 2024b)	Instalação simples, configuração complexa devido ao número de funcionalidades.
Dockscan (kost 2024)	220	27/09/2016	N/D	Simples, pois apenas consiste na execução de um script (kost 2024).
CoreOS Clair (Quay 2024a)	10410	09/10/2024	Significativo, pois consulta informação de vulnerabilidades de várias fontes (Red Hat 2019)	Quay 2024b oferece pouca informação para implementação em Kubernetes.
Trivy (Aqua Security 2024c)	24044	03/12/2024	Reduzido, pois as consultas estão otimizadas (Murillo 2024)	Implementação existente para análise contínua em Kubernetes (Aqua Security 2022).
Kubesec (ControlPlane 2024a)	1255	22/11/2024	N/D	Pela análise de ControlPlane 2024b, conclui-se que se trata de uma ferramenta para análise estática. A sua implementação é simples mas não diretamente suportada em Kubernetes.

Ferramenta	**Dimensão da comunidade**	**Data da versão mais recente**	**Impacto no desempenho**	**Simplicidade de implementação**
Kube-score (Westling 2024)	2814	01/10/2024	N/D	Sendo também uma ferramenta de análise estática, não tem um suporte direto em Kubernetes, servindo apenas como uma interface de linha de comandos para analisar definições de objetos (Westling 2024).
Kubeaudit (Shopify 2024)	1906	21/09/2024	N/D	Realiza análise estática, portanto não tem uma instalação direta em Kubernetes. Note-se, contudo, que esta tecnologia se encontra deprecada (Shopify 2024).
Kubelinter (StackRox 2024)	2999	07/11/2024	N/D	Realiza análise estática, portanto não tem uma instalação direta em Kubernetes. Adicionalmente, também permite analisar Helm charts (StackRox 2024).
Kubebench (Aqua Security 2024a)	7135	16/12/2024	N/D	Suporte nativo e simples em Kubernetes, basta apenas criar o objeto respetivo e os resultados são apresentados nos *logs* da aplicação (Aqua Security 2024a).

Tabela 2.5: Ferramentas estudadas para observabilidade de segurança

Ferramenta	**Dimensão da comunidade**	**Data da versão mais recente**	**Impacto no desempenho**	**Simplicidade de implementação**
Falco (Falco 2024)	7477	21/11/2024	Reduzido, usando o *driver* eBPF (Vugt e Malik 2023)	Instalação nativa em Kubernetes, possível através de Helm Chart, e simplicidade na criação de regras (Sysdig Inc. 2024).
Tetragon (Cilium 2024a)	3696	31/12/2024	Reduzido (Vugt e Malik 2023)	Instalação nativa em Kubernetes, possível através de Helm Chart. Configuração de fábrica permite a deteção de alguns ataques (Vugt e Malik 2023), e as políticas de observabilidade encontram-se detalhadamente documentadas em Cilium 2024c.
KubeArmor (KubeArmor 2024b)	1618	06/12/2024	Elevado, devido a capacidades extensas (Vugt e Malik 2023)	Suporte nativo em Kubernetes e instalação por Helm Chart. Configuração documentada para vários casos de uso, dos quais um deles é o *bare-metal* (KubeArmor 2024a).

Ferramenta	**Dimensão da comunidade**	**Data da versão mais recente**	**Impacto no desempenho**	**Simplicidade de implementação**
Tracee (Aqua Security 2024b)	3660	10/12/2024	Moderado (Vugt e Malik 2023)	Instalação simples com recurso a Helm Chart (Aqua Security 2024b). Pouca informação sobre configuração de políticas em Aqua Security 2024d
NeuVector (SUSE 2024a)	1122	16/11/2024	Sem impacto no modo Monitor, com impacto no modo Protect (SUSE 2024b)	Instalação simples, configuração complexa devido ao número de funcionalidades.

Tabela 2.6: Ferramentas estudadas para Implementação de políticas

Ferramenta	**Dimensão da comunidade**	**Data da versão mais recente**	**Impacto no desempenho**	**Simplicidade de implementação**
OPA Gatekeeper (Open Policy Agent 2024b)	3744	16/12/2024	Não especificado, mas Open Policy Agent 2024c documenta mecanismos para melhorar a *performance*	Instalação possível através de Helm Chart, configuração complexa visto recorrer a uma linguagem diferente, porém Open Policy Agent 2024a documenta vários exemplos de políticas.
Kyverno (Kyverno 2024a)	5861	10/12/2024	Reduzido (Kyverno 2024b)	Instalação possível através de Helm Chart, configuração numa sintaxe comum ao Kubernetes (YAML) e com mais de 300 exemplos documentados em Kyverno 2024c.
k-rail (Cruise 2024)	443	28/10/2021	N/D	Instalação possível através de Helm Chart, descrição extensa de políticas mas pouca informação sobre a sua configuração (Cruise 2024)

Tabela 2.7: Ferramentas estudadas para Logging, Metricas e Visibilidade

Ferramenta	**Dimensão da comunidade**	**Data da versão mais recente**	**Impacto no desempenho**	**Simplicidade de implementação**
Prometheus (Prometheus 2024)	56301	28/11/2024	Não especificado, mas Calvert 2023 apresenta alguns exemplos sobre como pode ser otimizada. Depende do número de métricas recolhidas, entre outros fatores.	Instalação possível através de Helm Chart, porém trata-se de uma ferramenta com uma arquitetura complexa (Prometheus 2024), bem como vários conceitos próprios, tornando a sua configuração complexa.
Grafana (Grafana 2024a)	65554	05/12/2024	Não especificado, mas dado tratar-se de uma ferramenta de visibilidade (Grafana 2024a), o seu impacto no desempenho pode ser reduzido a moderado.	Instalação possível através de Helm Chart (Grafana 2024b). A complexidade da configuração irá depender da complexidade e granularidade dos dados que se pretendem exibir.
Elasticsearch (Elastic 2024)	71063	12/12/2024	Dependerá do volume de dados recolhidos, entre outros fatores. Elastic 2020 e Gigasearch Engineering 2021 apresentam sugestões para melhoria do desempenho.	Pouca informação para instalações em Kubernetes.
Fluent-bit (Fluent 2024b)	5958	28/11/2024	Otimizado para alto desempenho e baixo consumo de recursos (Fluent 2024c)	Instalação por Helm Chart. Configuração granular, porém documentada em Fluent 2024a.

Ferramenta	Dimensão da comunidade	Data da versão mais recente	Impacto no desempenho	Simplicidade de implementação
Grafana Loki (Grafana 2024c)	24201	19/11/2024	Otimizado para alta disponibilidade através da indexação de metadados (Grafana 2024c)	Instalação possível através de Helm Chart (Grafana 2024b). Desenhado para operação simples e compatibilidade com Prometheus (Grafana 2024c).

2.6 Resumo

No presente capítulo, foi demonstrada a metodologia de revisão de literatura utilizada. Apresentadas as Research Questions, bem como os critérios de inclusão e exclusão a utilizar no processo de pesquisa, foi sintetizado o conhecimento obtido neste. Esta informação permitiu responder às questões colocadas, identificando-se as ferramentas e práticas adotadas para esse efeito. Dado o contributo significativo que o uso de ferramentas externas traz para a solução do problema, estas foram comparadas. Para essa comparação, foram definidos critérios, e conduziu-se a pesquisa de modo a aferir a posição de cada uma das tecnologias face a estes.

Realizada esta análise, existe conhecimento suficiente para a preconização de uma arquitetura para a solução, que servirá de base para o seu desenvolvimento. Contudo, devem antes ser abordadas as considerações de planeamento e gestão de competências tomadas no projeto. Como tal, esta análise será apresentada no próximo capítulo.

Capítulo 3

Arquitetura e Desenho de Solução

Tendo sido conduzida a investigação relativa ao estado da arte na área do conhecimento sobre qual o presente projeto se debruça, é necessário o desenho da solução para o mesmo. Primeiramente, será apresentada a arquitetura desenhada, bem como possíveis alternativas a esta, comparando-as entre si de modo a aferir as vantagens e desvantagens de cada uma. De seguida, e com base nos resultados obtidos em 2.5, escolher-se-ão as tecnologias que irão desempenhar cada uma das funções descritas.

3.1 Arquitetura da solução

Em 3.1, apresenta-se a arquitetura desenhada como solução para o problema descrito.

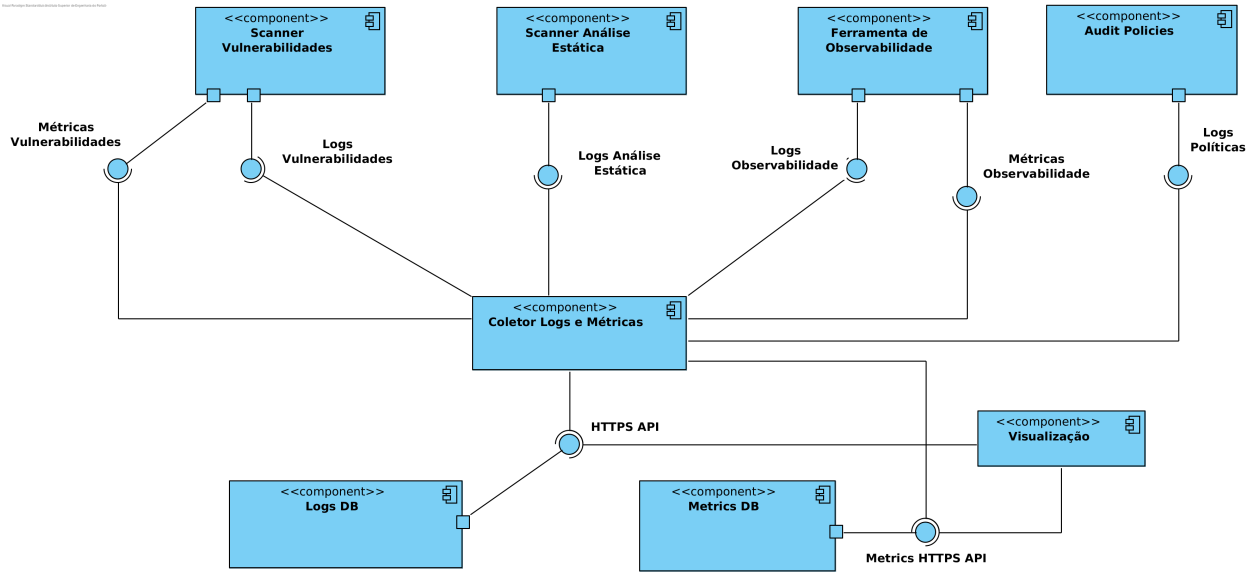

Figura 3.1: Arquitetura desenvolvida inicial

Como evidenciado, existe um conjunto de componentes que realiza a análise do *cluster*, em vários níveis, sendo estes: scanner de vulnerabilidades; scanner de análise estática; ferramenta de observabilidade e *audit policies*. Todas estas ferramentas exportam *logs*, isto é, registos das suas atividades, como por exemplo: resultados das análises conduzidas, ocorrências de eventos relevantes numa perspetiva de segurança, entre outros. No entanto, apenas duas delas exportam métricas - sendo estas distintas dos *logs*. O seu propósito é mais analítico, porém a sua informação é complementar àquela obtida nos *logs*, oferecendo

uma perspetiva mais completa na análise de eventos de segurança do *cluster*. As *audit policies*, por si, não exportam métricas da sua utilização, apenas emitem registos da eventual violação das condições definidas. No que toca ao scanner de análise estática, este não se trata, usualmente, de uma ferramenta em constante execução, mas sim de um procedimento único que conduz uma série de testes predefinidos face às configurações do *cluster*, emitindo *logs* dos seus resultados.

Estes *logs* são coletados por um componente que exerce essa função, armazenando-os subsequentemente numa base de dados destinada a este tipo de informação. Adicionalmente, este componente recolhe as métricas expostas pelos componentes, enviando-as para a base de dados respetiva. Tal solução apresenta uma vantagem no que toca à unificação da recolha destes dados, centralizando a configuração da mesma - ou seja, apenas é necessária a configuração de um componente para a recolha de dois tipos de dados.

Por fim, existe o componente de visualização, que consulta estas duas bases de dados para exibir a informação ao utilizador.

Ainda que a arquitetura apresentada corresponda às necessidades estipuladas, uma alternativa a esta foi desenvolvida, visando modificar a recolha de *logs* e métricas. Esta é apresentada em 3.2.

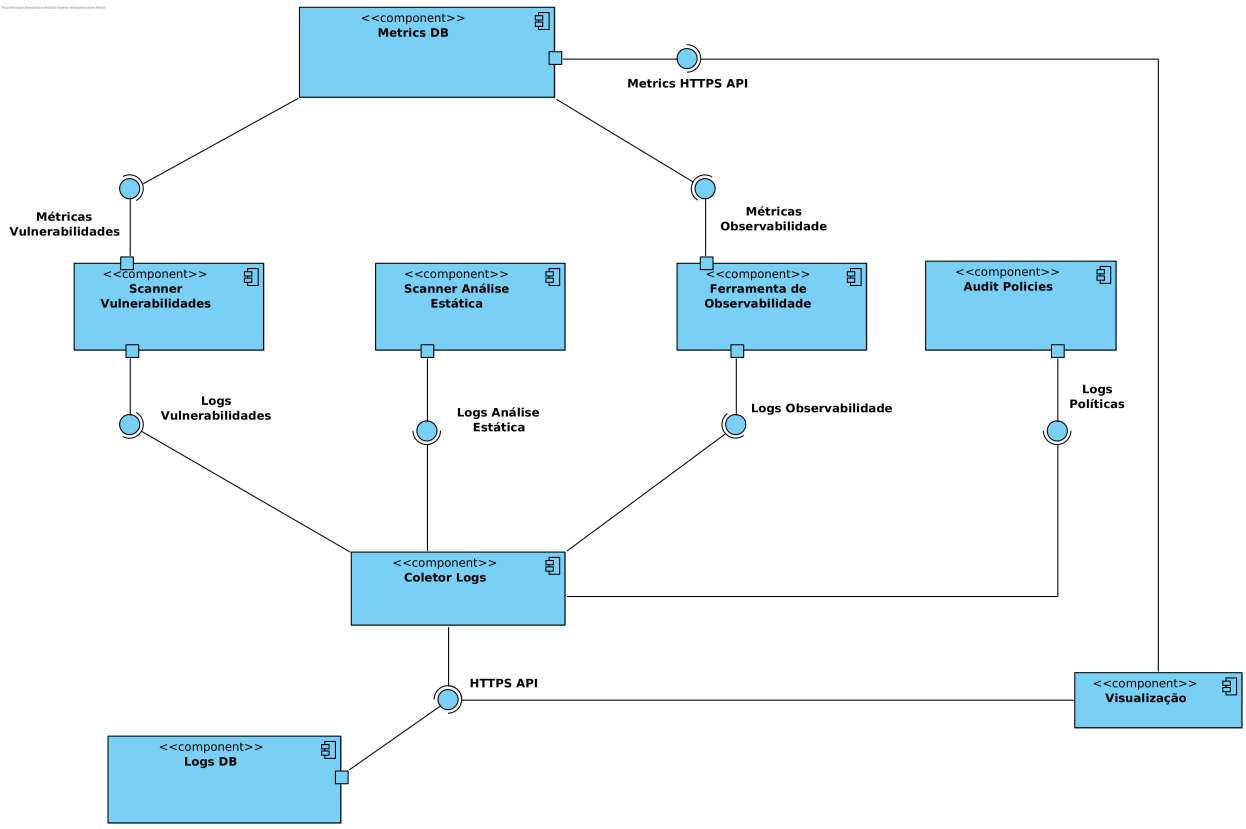

Figura 3.2: Arquitetura desenvolvida

A diferença apresentada na arquitetura alternativa reside na separação da recolha de *logs* e métricas. Enquanto que na primeira solução estas funções são realizadas pelo mesmo componente, na nova abordagem estes dados são recolhidos por componentes distintos. Como tal, dois componentes diferentes têm que ser configurados para que a recolha destes dados seja realizada, enquanto que na solução inicial apenas se configura um componente

para a recolha dos dois tipos de dados. Ainda que exista uma vantagem na unificação, esta não é significativa o suficiente para que a arquitetura 3.1 seja adotada. Tal deve-se ao facto de, tecnologicamente, a base de dados de métricas tipicamente ter uma configuração simples, até mesmo permitindo a auto-descoberta de métricas com base em certos critérios. Como tal, essa função não necessita de ser delegada a um componente que poderia tornar-se sobrecarregado, e cuja configuração seria de legibilidade inferior. Tratando-se de uma solução complexa, com diversos componentes, a configuração deve ser o mais simples e legível possível, para que o seu administrador consiga adaptá-la às necessidades do *cluster*.

Note-se que, de ambas as soluções apresentadas anteriormente, estão ausentes alguns elementos, nomeadamente: a encriptação do etcd; a autenticação e RBAC; e o *admission control*. Como tal, o diagrama em 3.3 foi desenvolvido, para evidenciar uma perspetiva que apresente os elementos externos ao *cluster*.

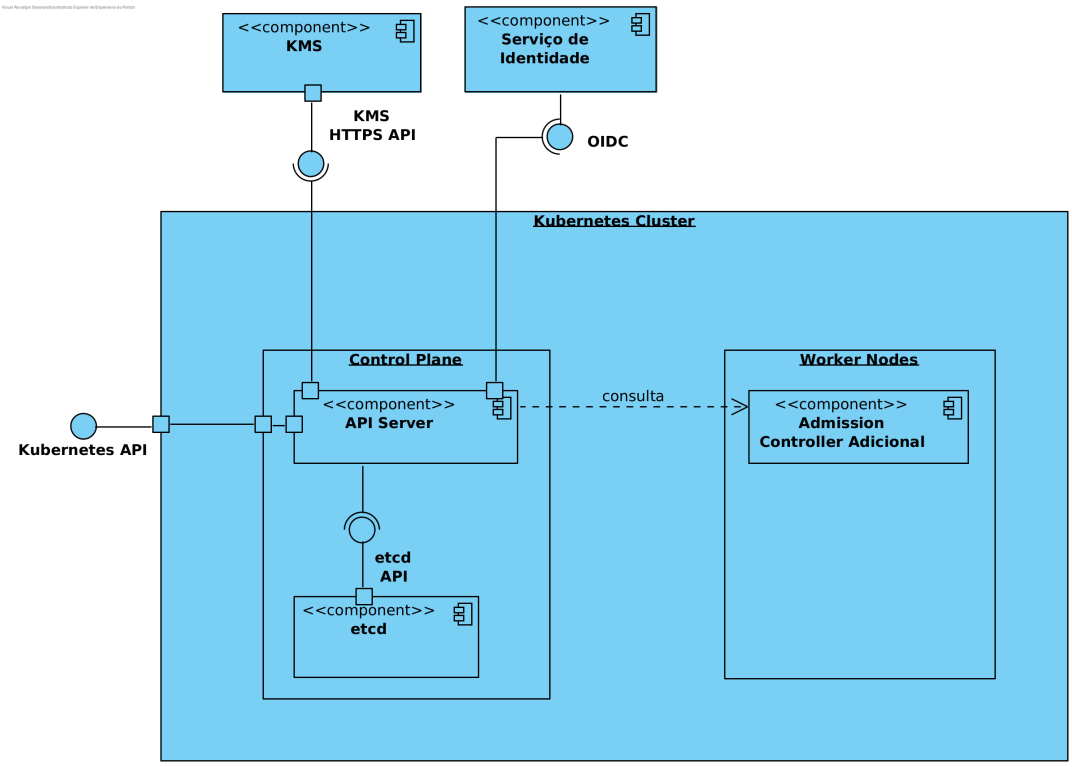

Figura 3.3: Arquitetura externa ao *cluster*

Como abordado em 2, a Kubernetes API é o principal ponto de comunicação externa para com o *cluster*, sendo esta exposta pelo Control Plane, que aloja o componente Kubernetes API Server. Esta API é responsável por várias funções, entre as quais: leitura/armazenamento de Secrets; e validação de políticas de RBAC.

Considerando a gestão de Secrets, estes são armazenados no etcd, pelo que quando se pretende escrever ou consultar um Secret, o API Server comunica com o etcd para esse efeito. Ora, adotando a abordagem de encriptação descrita em 2.4.5, existe também a comunicação entre a API e um KMS externo, responsável por gerir uma das chaves de encriptação utilizadas neste processo. O fluxo de comunicação neste sistema é descrito de modo mais compreensivo em 3.4.

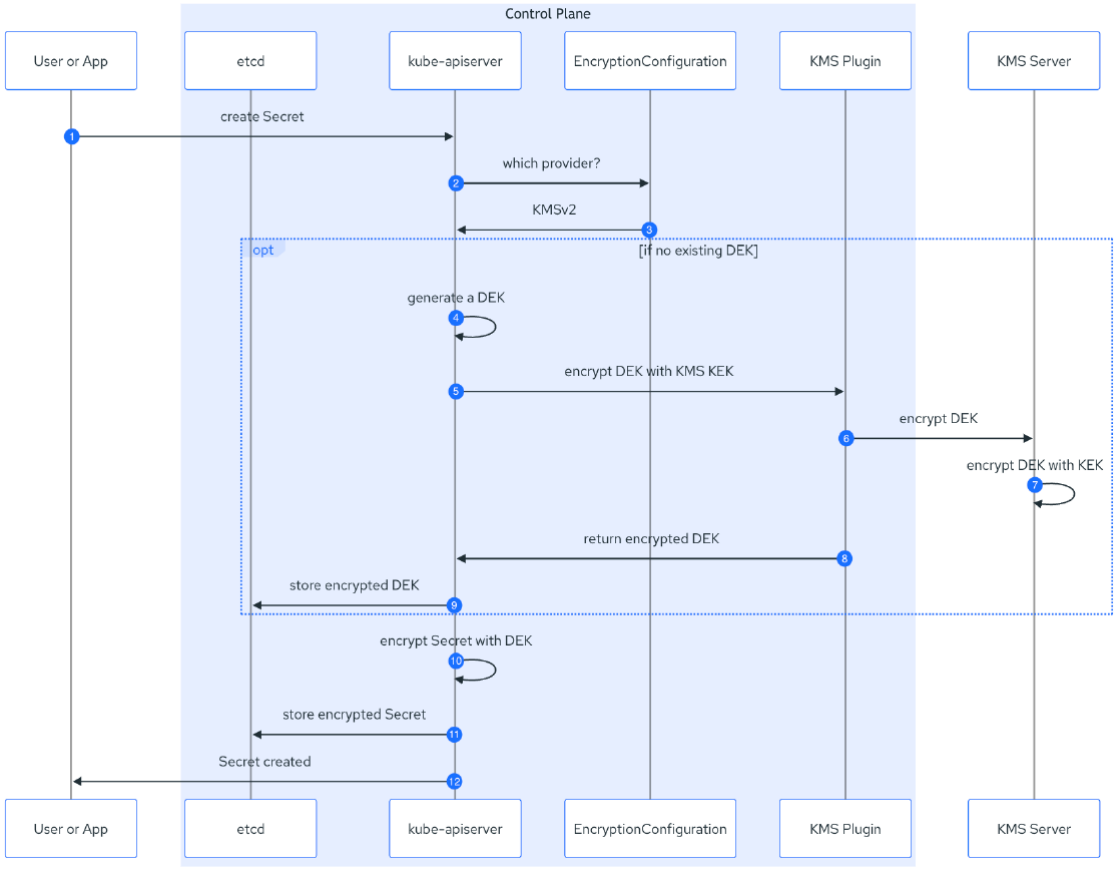

Figura 3.4: Funcionamento do sistema kmsv2 (Gkatziouras et al. 2024)

Utilizando esta tecnologia, a criação de um Secret segue o seguinte fluxo de alto nível: o utilizador envia um pedido HTTPS ao API Server, instruindo que pretende criar um Secret; a API gera uma chave de encriptação de dados (DEK), que por sua vez é encriptada com uma chave de encriptação armazenada num ponto externo (KEK); a chave DEK é utilizada para encriptar o Secret, que é posteriormente armazenado no etcd.

Relativamente à consulta de políticas de RBAC, estas também são armazenadas pelo etcd, pelo que o API Server comunica com este componente de modo a validar se a entidade que pretende executar uma determinada ação está autorizada a fazê-lo. Para algumas destas operações, será também consultado um sistema de *admission control* - conforme exemplificado em 2.4.2. Este sistema consiste num Deployment ao nível do Kubernetes, pelo que os seus Pods serão executados nos *worker nodes*.

Finalmente, vale realçar a vertente de gestão de identidade presente na solução. Adotando as recomendações obtidas durante a revisão de literatura, será utilizado um provedor de identidade externo, que será consultado pelo API Server na fase de autenticação de utilizadores. Assim, a gestão destes é delegada para um componente externo.

3.2 Escolhas tecnológicas

Tendo sido selecionada a arquitetura, e mantendo presente o conhecimento obtido no processo de revisão de literatura, é crucial a definição de quais tecnologias cumprirão os propósitos delineados para cada componente. Esta escolha é fundamentada na comparação realizada em 2.5, através da análise dos critérios definidos. As figuras 3.5 e 3.6 adaptam as apresentadas na secção anterior de modo a refletir estas escolhas.

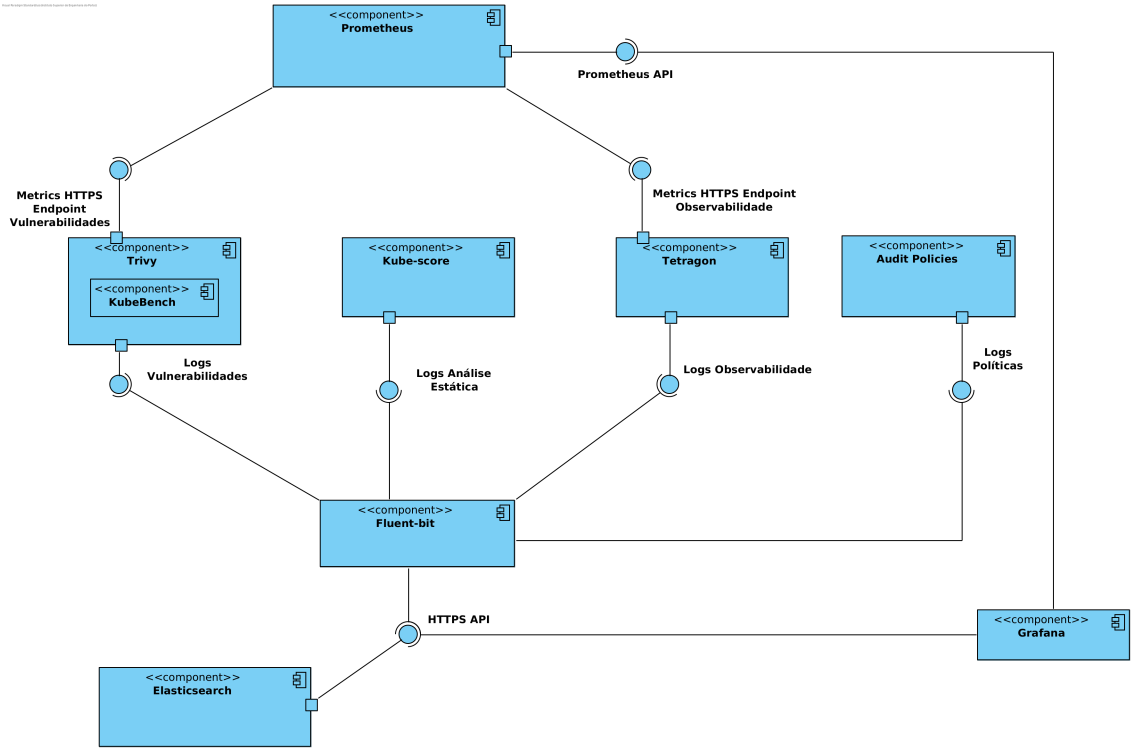

Figura 3.5: Escolhas tecnológicas

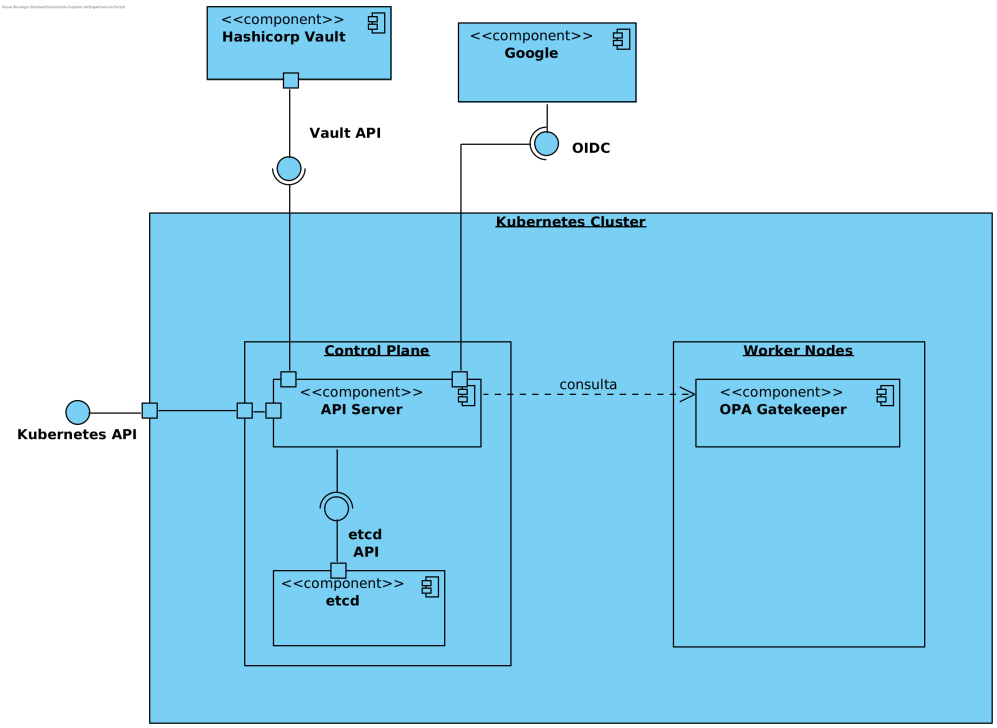

Figura 3.6: Escolhas tecnológicas para arquitetura externa

Analisando a perspetiva interna ao *cluster*, as ferramentas de análise de vulnerabilidades escolhidas foram o Trivy e o Kube-score. Quanto à escolha do Trivy, esta mostra-se vantajosa não só pelos factos apresentados na sua análise, mas também por esta já incluir o KubeBench, possibilitando então uma junção entre a análise de vulnerabilidades e a análise estática face a padrões documentados de boas-práticas. Complementando esta vertente de análise estática, o Kube-score foi escolhido para analisar os manifestos persistidos no *cluster*, verificando se estes estão totalmente corretos. O Tetragon foi escolhido para a observabilidade de eventos de segurança em Kubernetes, dado o seu potencial como tecnologia emergente, bem como a sua rápida *performance* e a possibilidade de ser utilizado também como ferramenta de *runtime enforcement*. As *audit policies* são uma funcionalidade já presente no Kubernetes, pelo que a sua adoção consistirá na configuração centralizada de políticas relevantes, bem como na recolha de eventos relativos a estas e a sua subsequente apresentação ao utilizador.

Para a recolha de *logs*, o Fluent-bit mostrou-se como sendo a melhor escolha devido ao seu baixo consumo de recursos, bem como extensibilidade de configuração. O Elasticsearch e Prometheus exercerão funções complementares - um recolherá *logs*, enquanto outro fará a coleção de métricas. Os dados de ambas estas bases de dados serão apresentados no Grafana, a ferramenta escolhida para visualização.

No que toca à perspetiva mais abrangente do *cluster*, realça-se a utilização do Hashicorp Vault como KMS, utilizado para a encriptação de Secrets. Para a autenticação e autorização de utilizadores, o Google será utilizado como provedor de identidade externa, em detrimento de uma solução como o Keycloak. Tal deve-se ao facto de a organização atualmente utilizar a *suite* da Google para autenticação em outras ferramentas do seu ecossistema, sendo então mais viável a sua implementação no presente contexto. Finalmente, o OPA Gatekeeper foi escolhido para a implementação de políticas adicionais no foro do *admission*

control, ainda que a sua linguagem de especificação de regras seja distinta do YAML. Esta escolha é motivada pela extensiva possibilidade de configuração de políticas oferecida pelo Gatekeeper, auxiliada pelo catálogo destas fornecido pelos desenvolvedores.

3.3 Resumo

No atual capítulo foi apresentada a arquitetura desenhada para a construção da solução. Inicialmente, uma possível abordagem foi demonstrada e explicada, seguindo-se da sua comparação com uma alternativa. Extraídas as devidas conclusões, foi também apresentada uma vista da solução arquitetural de mais alto nível, mostrando as interações entre o *cluster* e ferramentas externas a este. Finalmente, as escolhas tecnológicas para cada componente definido foram elencadas, oferecendo-se uma justificação para cada uma delas.

Capítulo 4

Implementação

O presente capítulo descreve o processo de implementação da solução desenvolvida, começando por uma descrição do ambiente de testes utilizado, elencando também os passos tomados para a sua preparação. De seguida, o processo de implementação da solução será explorado, apresentando os conceitos base, a configuração das ferramentas de modo a resolver o problema apresentado, e a integração das mesmas através de *logs* e métricas, culminando numa prova de conceito para a solução final. Finalmente, será apresentado processo de testes, descrevendo os que foram realizados e interpretando os seus resultados. Estes servem de fundamento para a análise da solução, aferindo se esta cumpre os objetivos pretendidos.

É importante realçar que o desenvolvimento realizado não corresponde à solução final, mas sim a uma prova de conceito. Desse modo, apenas alguns componentes da arquitetura preconizada foram implementados. A escolha de que ferramentas implementar foi concebida tendo em conta quer a sua relevância para os objetivos definidos, quer a sua adequação à validação da proposta pelo processo de testes. Assim, não se pretendeu abranger integralmente tudo o que foi inicialmente idealizado, mas sim implementar um subconjunto representativo, que permitisse a realização de testes e a avaliação da viabilidade da solução proposta.

4.1 Descrição do ambiente de desenvolvimento

Para o desenvolvimento e subsequente avaliação da prova de conceito apresentada no âmbito do projeto, foi configurado um ambiente de testes, visando replicar - em menor escala - um ambiente de produção. Na secção atual, o procedimento de preparação deste será descrito, apresentando-se também as especificações relevantes de cada componente.

O ambiente de testes foi configurado com recurso a máquinas virtuais, hospedadas num servidor interno da organização. Este utiliza Ubuntu 24.04 LTS como seu sistema operativo, tendo no seu *hardware* um CPU com 12 núcleos, 72Gb de memória RAM, e 1Tb de espaço em disco.

Neste sistema, foi criado um conjunto de 4 máquinas virtuais (VM), conectadas na mesma rede privada, com as especificações apresentadas na tabela 4.1. A função de cada uma será explicada posteriormente.

Tabela 4.1: Características das VMs do ambiente de testes

Hostname	CPUs	Memória (Gb)	Espaço em disco (Gb)
`thesis-m-01`	4	8Gb	100Gb
`thesis-w-01`	4	8Gb	100Gb
`projects-dbs`	4	16Gb	50Gb

O *cluster* de Kubernetes onde a solução foi testada e executada é consistido pelas máquinas `thesis-m-01` e `thesis-w-01`, sendo estas o Control Plane e o worker node, respetivamente. O aprovisionamento desta infraestrutura foi feito com recurso a uma Ansible Role, proprietária à empresa.

No Kubernetes, existe o conceito de distribuições. Estas são versões alteradas da plataforma base, que visam cumprir vários propósitos diferentes, nomeadamente otimização do consumo de recursos, simplificação da gestão do *cluster*, entre outros. Para o ambiente de testes, a distribuição utilizada foi o `k3s`[1]. Esta utilização foi um requisito por parte da empresa, visto ser esta a distribuição utilizada em todos os seus *clusters*, e não ser plausível no âmbito atual a troca para qualquer outra opção.

Quanto à máquina `projects-dbs`, esta serviu o propósito de hospedar alguns componentes externos necessários à solução. Nomeadamente: o Hashicorp Vault, que foi definido como um componente externo; e o Elasticsearch. Quanto ao segundo, a instalação deste foi preconizada, na secção anterior, como sendo interna ao *cluster*. Contudo, a sua instalação foi realizada neste servidor externo ao Kubernetes por dois fatores. Primeiramente, a organização já utiliza instâncias externas de Elasticsearch em outros cenários, pelo que a sua instalação deste modo proporciona uma integração com a restante *stack* tecnológica. Acresce-se o facto de que esta ferramenta assume uma utilização de recursos considerável, pelo que não seria viável nem para a empresa, nem no atual ambiente de testes, utilizá-la numa execução em Kubernetes. Deve mencionar-se que a instalação e configuração destes serviços - tal como com o `k3s` - foi realizada através de Ansible Roles e Playbooks criados para esse efeito, e proprietários da organização, pelo que detalhes sobre estes não podem ser divulgados no presente relatório.

Conclui-se, deste modo, a apresentação do ambiente de testes utilizado, pelo que na próxima secção se descreverá o processo de implementação da prova de conceito desenvolvida.

4.2 Descrição da implementação

Na presente secção, será descrito o processo de implementação da solução desenvolvida. Este será apresentado de modo incremental, isto é, começando pela descrição dos componentes individuais, explanando de seguida o modo sob o qual a sua integração foi realizada, resultando na solução final.

4.2.1 Encriptação de Secrets - KMSv2

O primeiro passo na implementação da solução proposta foi a configuração da encriptação de Secrets, recorrendo ao sistema KMSv2. Conforme explicitado, este recorre a um componente externo - o Hashicorp Vault, considerando a arquitetura proposta - para a geração e manutenção das chaves de encriptação.

[1] Mais informação sobre esta distribuição pode ser encontrada em: `https://k3s.io/`

Inicialmente, foi necessária alguma configuração no Vault para que o Kubernetes consiga interagir devidamente com ele [2]. Começou-se pela ativação do sistema de *Transit Secrets*, destinado ao desempenho de funções criptográficas para dados em trânsito (Hashicorp 2025). Ativada esta *Engine* ao nível do Vault, foi criado um *Transit Secret* intitulado `kms`, que será o utilizado pela API do Kubernetes para o sistema de encriptação de Secrets. Este processo foi realizado através do conjunto de comandos apresentado em 4.1 (o *token* utilizado para interação com o Vault não será apresentado, por motivos de segurança).

```
1 $ export VAULT_ADDR="https://projects-dbs.jscrambler.com:9443"
2 $ export VAULT_TOKEN="hvs.XYZW"
3 $ vault secrets enable transit
4 $ vault write -f transit/keys/kms
```

Excerto de código 4.1: Configuração de *Transit Secret* no Vault

De seguida, foi criada uma política ao nível do Vault, que posteriormente será associada ao método de autenticação usado pelo Kubernetes para contactar com este serviço. O intuito desta configuração é melhorar a segurança do sistema, garantindo que permissões excessivas não são atribuídas. A definição desta política apresenta-se em 4.2, e analisando-a é possível aferir que esta apenas permite a manipulação do *Transit Secret* configurado anteriormente.

```
1  # vault-kubernetes-kms.hcl
2  path "auth/token/lookup-self" {
3    capabilities = ["read"]
4  }
5
6
7  path "transit/encrypt/kms" {
8    capabilities = [ "update" ]
9  }
10
11
12 path "transit/decrypt/kms" {
13   capabilities = [ "update" ]
14 }
15
16
17 path "transit/keys/kms" {
18   capabilities = [ "read" ]
19 }
```

Excerto de código 4.2: Configuração de política no Vault

Estabelecida esta política, foi então criada uma `AppRole`, que será o recurso utilizado para autenticação e manipulação das chaves de encriptação geridas pelo Vault. Este processo foi conduzido utilizando os comandos apresentados em 4.3, onde vale realçar a utilização da política `vault-kubernetes-kms` criada previamente. O *output* dos comandos nas linhas 3 e 4 consiste nas chaves de autenticação que serão utilizadas pelo Kubernetes.

[2] Este processo segue, em grande medida, aquele apresentado pelos desenvolvedores do `vault-kubernetes-kms`, descrito posteriormente. Pode ser consultado em: `https://falcosuessgott.github.io/vault-kubernetes-kms/configuration`

```
$ vault auth enable approle
$ vault write auth/approle/role/kms token_num_uses=0 token_period
   =3600 token_policies=vault-kubernetes-kms
$ vault read auth/approle/role/kms/role-id
$ vault write -f auth/approle/role/kms/secret-id
```

Excerto de código 4.3: Criação de AppRAppRole para autenticação no Vault

Nesta fase, o Vault encontra-se totalmente preparado, pelo que apenas resta a configuração da interação da API do Kubernetes com esta ferramenta. Ora, esta interação não é suportada por defeito, pelo que se deve recorrer a um componente adicional - o KMS *plugin*. Dada a utilização do Hashicorp Vault, o *plugin* adotado foi o `vault-kubernetes-kms` (FalcoSuessgott e Morelly 2024). Este *plugin* consiste apenas num Pod que estabelece a conexão desejada, e cujo manifesto é disponibilizado pelos autores da ferramenta [3]. Devido a algumas particularidades da arquitetura desejada - concretamente, a utilização de `k3s` e um ambiente *bare-metal*, foram necessárias algumas alterações, que serão agora descritas.

Primeiramente, foi necessária a configuração de `hostAliases` no manifesto do Pod, para que a conexão ao Vault possa ser feita recorrendo inicialmente à resolução DNS, e como tal validando o certificado TLS apresentado. Isto deve-se apenas à particularidade do ambiente de testes ser disponibilizado numa rede privada, não existindo um servidor de DNS interno para a resolução de nomes. Não é algo necessário por imposição da ferramenta, mas sim do ambiente utilizado, merecendo uma menção particular. De seguida, foi configurado o *container* que executa este *plugin*, conforme apresentado no excerto 4.4. Na linha 7 do referido segmento, define-se a instância de Vault como sendo a que reside no servidor `projects-dbs.jscrambler.com`, exposto no porto 9443. Nas linhas 8 a 10, alteram-se os campos de autenticação para que esta seja bem sucedida e utilize a `AppRole` criada, com o ID e segredo especificados nas linhas 9 e 10, respetivamente.

```
spec:
  containers:
    - name: vault-kubernetes-kms
      image: falcosuessgott/vault-kubernetes-kms:v1.0.3
      command:
        - /vault-kubernetes-kms
        - -vault-address=https://projects-dbs.jscrambler.com:9443
        - -auth-method=approle
        - -approle-role-id=<redacted-role-id>
        - -approle-secret-id=<redacted-secret-id>
```

Excerto de código 4.4: Criação de AppRAppRole para autenticação no Vault

Por fim, foi necessária a configuração de algumas *tolerations* e *nodeSelectors* ao nível do Pod. A interação entre o Kubernetes e o Vault é efetuada pelo API Server, que recorre ao *plugin* configurado. Ora, esta API é um componente do Control Plane, que por sua vez é executado nos *master nodes*. Como tal, o Pod do `vault-kubernetes-kms` deve ser executado num destes *nodes*, e não num *worker node*. As configurações referidas vêm servir esse mesmo propósito: garantir que o *node* selecionado para execução é um *master*; e que as *flags* que este tem ativas não impedem o Pod de ser executado nele. O excerto de código

[3] Pode ser consultado em: `https://falcosuessgott.github.io/vault-kubernetes-kms/configuration/#example-vault-approle-auth`

responsável por esta configuração é apresentado em 4.5, e nele é visível a definição das duas chaves mencionadas. Nas linhas 2 e 3, configura-se o atributo `nodeSelector` para selecionar apenas os nós computacionais em que o valor do atributo `node` seja igual a `"master"`. Nas linhas 4 a 10 definem-se as `tolerations` necessárias. Os valores das mesmas são atributos que os nós constituintes do Control Plane assumem. Logo, ao permitir estes valores, será possível a alocação do Pod a um destes nós.

```
1  spec:
2    nodeSelector:
3      node: master
4    tolerations:
5    - key: CriticalAddonsOnly
6      operator: Exists
7    - effect: NoSchedule
8      operator: Exists
9    - effect: NoExecute
10     operator: Exists
```

Excerto de código 4.5: Configuração de *tolerations* e *nodeSelectors* no `vault-kubernetes-kms`

Concluídas estas alterações, o manifesto do Pod foi criado, executando-o no *cluster*, pelo que apenas restou a configuração do API Server para utilização do KMSv2.

O conceito de `EncryptionConfiguration` torna-se relevante nesta fase da implementação, pois é este que instrui o API Server a utilizar o método de encriptação pretendido. Assim, foi definido o manifesto apresentado no excerto 4.6, especificando que o sistema `kms` deve ser utilizado para a encriptação de objetos do tipo Secret. Note-se a utilização adicional do *provider* `identity`, necessária numa fase inicial para conduzir a transição entre segredos em texto claro (mais concretamente, em base64) e segredos verdadeiramente encriptados.

```
1  kind: EncryptionConfiguration
2  apiVersion: apiserver.config.k8s.io/v1
3  resources:
4    - resources:
5        - secrets
6      providers:
7        - kms:
8            apiVersion: v2
9            name: vault-kubernetes-kms
10           endpoint: unix:///opt/kms/vaultkms.socket
11       - identity: {}
```

Excerto de código 4.6: Recurso `EncryptionConfiguration` utilizado

Tratando-se de um ambiente que utiliza `k3s`, a aplicação deste tipo de recurso não é tão linear quanto a dos restantes. Na prática, foi necessário alterar o ficheiro de configuração principal do `k3s` (`/etc/rancher/k3s/config.yml`), especificando a utilização deste novo recurso como uma propriedade adicional, tal como demonstrado no excerto 4.7.

```
1  kube-apiserver-arg:
2    - 'encryption-provider-config=/etc/rancher/k3s/encryption/
     kmsConfig.yaml'
```

Excerto de código 4.7: Recurso `EncryptionConfiguration` aplicado na configuração do `k3s`

Por fim, reiniciou-se o serviço `k3s` (através do comando `systemctl restart k3s`) para que o sistema carregasse estas alterações. Posto isto, o KMSv2 encontra-se configurado no *cluster*, pelo que todos os segredos serão encriptados recorrendo a este.

4.2.2 Implementação do `kube-score`

Conforme apresentado em 2.5, o `kube-score` é apenas uma interface de linha de comandos, pelo que a sua implementação e subsequente integração não são diretamente suportados. Para esse efeito, foi desenvolvido um Helm Chart intitulado `run-kube-score`, que tem como objetivo realizar uma execução periódica desta ferramenta, com base numa configuração definida pelos administradores.

O Chart define a criação de um CronJob em Kubernetes. Ora, como qualquer carga aplicacional, este deve definir uma imagem Docker a ser utilizada em cada uma das suas execuções programadas. Como tal, essa imagem foi também criada no âmbito da implementação. Esta usa como base a imagem `zegl/kube-score` - disponibilizada pelos desenvolvedores do `kube-score`, e executa um *script* criado para cumprir este objetivo, sendo o núcleo da implementação deste componente.

O *script* `scoreManifests.sh` interpreta configurações especificadas em dois ficheiros - `namespaces` e `resources` - e executa o `kube-score` com base nesta informação. Estes dois ficheiros de configuração permitem aos administradores a especificação de que *namespaces* pretendem monitorizar, bem como que tipos de recursos devem ser analisados - por exemplo, Deployments, Pods, StatefulSets, entre outros tipos de objetos aplicacionais em Kubernetes. O excerto de código relevante do *script* apresenta-se em 4.8, sendo possível verificar como o `kube-score` é utilizado neste contexto.

Com base nas configurações de recursos e *namespaces* interpretadas, são obtidos os manifestos de cada carga aplicacional que vá de encontro a estas, e estes são analisados pelo *kube-score*, que apresenta os resultados de modo formatado nos seus *logs*.

```
1 #!/bin/bash
2 echo "Namespaces were provided, only conducting static analysis
   for: ${NAMESPACES_TO_CHECK}."
3 kubectl api-resources --verbs=list --namespaced -o name | grep -
   Ei "${RESOURCE_REGEX}" | xargs -I{} bash -c "kubectl get {} -n $
   {NAMESPACES_TO_CHECK} -oyaml && echo ---" | kube-score score -o
   ci -
```

Excerto de código 4.8: *Script* de execução do `kube-score`

Com a perspetiva de integração em mente, bem como da instalação desta solução em Kubernetes, requerer o preenchimento de ficheiros de configuração para a utilização da ferramenta torna-se pouco praticável. Desse modo, o Chart foi implementado de modo a receber configurações descritas no seu ficheiro de *values*, e a transformá-las em ficheiros com o formato apropriado para montagem por volume no contentor - que executa a imagem Docker desenvolvida - e consequente interpretação pelo *script*.

O ficheiro de *values* - onde os administradores configuram a ferramenta - assume as opções de configuração apresentadas em 4.9.

```
scanConfig:
  resources:
  - NetworkPolicies
  - Roles
  - Deployments
  - ReplicaSets
  - Pods
  - Services
  allNamespaces: false
  namespaces:
  - default

schedule: "'*/5 * * * *'"
```

Excerto de código 4.9: Configuração do Chart

O Chart desenvolvido interpreta estas configurações e transforma-as num ConfigMap, que por sua vez é montado por volume na definição do CronJob. Assim, quando o momento definido para a execução - na chave `schedule` do *values* - for atingido, o Pod que será criado pelo CronJob também terá a informação deste ConfigMap no seu sistema de ficheiros, sendo possível a sua interpretação pelo *script*.

Instalando o Chart - com a configuração pretendida - é possível verificar a sua execução, de acordo coma definição de `schedule`. Na figura 4.1 são apresentados os *logs* de execução desta ferramenta. Destes, é possível extrair que a ferramente `kube-score` analisou recursos no *namespace* `default` - no qual, no momento de execução, apenas existia um recurso do tipo `Service`. Então, foi conduzida uma análise deste serviço, detetando-se uma falha no mesmo - o facto de o seu `selector` não corresponder a nenhum Pod existente.

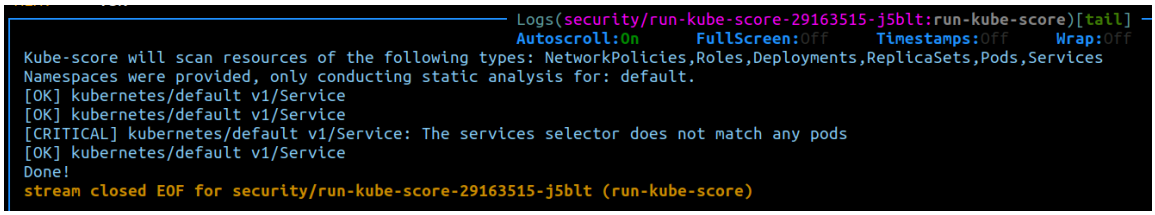

Figura 4.1: *Logs* da ferramenta desenvolvida para executar o `kube-score`

Assim, conclui-se a implementação do `kube-score` no âmbito do projeto, consistindo esta na sua transformação de uma simples linha de comandos, em algo cuja execução é configurável e periódica para objetos que já são executados no *cluster*. Esta implementação também possibilita a subsequente integração, visto a configuração ser realizada através de um Helm Chart, e os seus *logs* serem gerados de modo explícito e legível - algo que será crucial para a interligação deste componente com os restantes.

4.2.3 Implementação do Tetragon

Dada a natureza de baixo nível do Tetragon - operando diretamente com eBPF e funções do kernel - a sua implementação de modo prático para a solução geral teve uma complexidade elevada. Como tal, esta divide-se em duas fases: a configuração inicial do Tetragon; e a criação de políticas de monitorização interpretadas por este. Ambas recorreram a Helm Charts: no primeiro caso, a configuração do Helm Chart existente para o Tetragon, de modo a que

este sirva as necessidades especificadas pelo projeto; e no segundo caso o desenvolvimento de um novo, que crie as políticas de monitorização com base na configuração providenciada pelos administradores.

Configuração do Tetragon

O ficheiro de *values* do Tetragon é bastante extenso, pelo que apenas as secções chave serão apresentadas na presente secção. Como expresso anteriormente, o Tetragon é capaz, por defeito, de detetar alguns ataques através das suas políticas de fábrica. A investigação prática desta ferramenta revelou que estes incidem essencialmente na execução de processos - quer nos Pods em Kubernetes, quer no próprio *node*, se configurado para esse efeito. Ora, este tipo de deteção é bastante propensa a falsos-positivos.

O intuito desta configuração inicial é a redução destes incidentes, garantindo que apenas aqueles relevantes para o ambiente em questão - e o negócio, no geral - são detetados. Para esse efeito, o Tetragon disponibiliza algumas chaves no *values* que podem ser configuradas de modo a definir *blacklists* e/ou *whitelists* de eventos, dependendo da abordagem desejada.

Inicialmente, foi definida uma *whitelist* de tipos de eventos que devem ser exportados, descartando então aqueles que apenas gerariam ruído nos *logs* desta ferramenta. Esta configuração apresenta-se em 4.10.

```
exportAllowList: |-
  {"event_set":["PROCESS_EXEC", "PROCESS_EXIT", "PROCESS_KPROBE", "
   PROCESS_UPROBE", "PROCESS_TRACEPOINT", "PROCESS_LSM"]}
```

Excerto de código 4.10: *Whitelist* de tipos de eventos no Tetragon

De seguida, foi implementada uma exclusão com recurso a uma *blacklist*, complementar à configuração implementada anteriormente, que se apresenta em 4.11. Após definir que tipos de eventos manter, definem-se aqueles que, com base em alguns critérios, devem ser descartados.

```
exportDenyList: |-
  {"health_check":true}
  {"namespace":["security", "kube-system", "calico-system", "calico
   -apiserver", "kube-public", "kube-node-lease", "tigera-operator"
   ]}
  {"namespace": [""], "event_set":["PROCESS_EXIT"]}
  {"namespace": [""], "binary_regex":["^/var/lib/rancher/k3s","^/
   usr/bin/runc$","^/proc/self","/pause","/usr/local/bin/clamdcheck
   .sh","/usr/sbin/clamd","/usr/bin/freshclam"]}
  {"namespace": [""], "binary_regex":["^/usr/sbin/ip6*tables$"], "
   arguments_regex": [".*KUBE.*"]}
  {"namespace": [""], "binary_regex":["^/root/recreate-statefulset.
   sh$"]}
  {"namespace": [""], "parent_binary_regex":["^(/usr)*/bin/bash$"],
    "parent_arguments_regex": ["/opt/scripts/.*monitor.sh","^/root/
   recreate-statefulset.sh$"]}
  {"namespace": [""], "binary_regex":["^(/usr)*/bin/(ba)*sh$"], "
   arguments_regex": ["/opt/scripts/.*monitor.sh","^/root/recreate-
   statefulset.sh$",".*clamdcheck.sh$"]}
```

Excerto de código 4.11: *Blacklist* de eventos no Tetragon

Na linha 3 de 4.11, define-se que os eventos da lista apresentada de *namespaces* devem ser ignorados. Estes incluem o vários *namespaces* de sistema, cuja análise não é relevante, bem como gera ruído desnecessário - focando assim o Tetragon na análise de outros mais relevantes, tais como aqueles que disponibilizem aplicações aos clientes da organização.

A partir da linha 4, o seletor de *namespace* encontra-se vazio. Isto deve-se ao facto de essa configuração permitir a filtragem de eventos que aconteçam nos *nodes* do *cluster*, permitindo a monitorização em tempo real não só do ambiente Kubernetes, mas também da infraestrutura que o hospeda. Estas linhas visam excluir eventos recorrentes nestes *nodes*, que dizem respeito a processos regulares da empresa, tais como processo de monitorização, manutenção, deteção de *malware*, entre outros. Esta configuração permite que este tipo de processo recorrente - e confiável - não gere ruído na solução final. Contudo, as exceções são definidas do modo mais restrito possível, para que não gerem falsos-negativos, não detetando potenciais ataques.

Finalmente, os campos de cada *log* gerado pelo Tetragon foram analisados - e subsequentemente restritos. Dada a quantidade elevada de informação em cada *log*, avaliou-se a importância de cada campo, escolhendo apenas aqueles cuja inclusão é relevante para a análise destes *outputs*. Esta configuração foi, também, realizada sob a forma de uma *allowlist* - semelhante ao primeiro caso demonstrado - e apresenta-se em 4.12.

```
1 fieldFilters: |-
2   {"fields":"process.binary,process.arguments,process.cwd,process.
     pod.name,process.pod.container.name,process.pod.namespace,
     process.flags,process.tid,process.pod.workload_kind,parent.
     binary,parent.arguments,args,status,signal,message,policy_name,
     function_name,event,file.path"}
```

Excerto de código 4.12: *Allowlist* de campos nos *logs* do Tetragon

Por fim, é possível realizar uma preparação do Tetragon para a subsequente integração com outras ferramentas, nomeadamente o Prometheus. O Helm Chart do Tetragon permite a configuração de um `ServiceMonitor`, algo que fará com que as métricas expostas por este serviço sejam automaticamente detetadas e recolhidas por uma instância de Prometheus que se encontre no mesmo *cluster*. A criação deste `ServiceMonitor` é bastante simples, conforme exemplificado pela configuração em 4.13. Apenas basta ativá-lo, algo que cria o respetivo recurso em Kubernetes. Aquando da implementação do Prometheus (em 4.2.5), o modo sob o qual a integração é realizada será explorado.

```
1 serviceMonitor:
2   enabled: true
3   scrapeInterval: ""
```

Excerto de código 4.13: Criação do `ServiceMonitor` para recolha de métricas do Tetragon

Deste modo, a configuração inicial do Tetragon encontra-se implementada, pelo que é possível passar ao segundo ponto mencionado - a implementação de políticas de monitorização a serem utilizadas por esta ferramenta.

Implementação de políticas de monitorização

De modo a cumprir este requisito, foi desenvolvido um novo Helm chart - `tetragon-policies` - que recebe configurações através do ficheiro de *values* e cria políticas com base nessa informação. Note-se, contudo, que o desenvolvimento apresentado para este subcomponente é pouco abrangente, tratando-se de uma prova de conceito.

O Tetragon, conforme discorrido no processo de revisão de literatura, é uma ferramenta complexa. Na prática, este cria o Custom Resource Definition (CRD) de `TracingPolicy`, que define um objeto em Kubernetes para cada política de monitorização interpretada pela ferramenta. As políticas permitem aos administradores a configuração de processos de monitorização a eventos que se sucedam no *kernel*. Nestas, são definidos dois elementos: os *hook points*, que permitem ao Tetragon executar código ao nível do *kernel* para capturar eventos; e os *selectors*, que permitem filtrar quais devem ser exportados por via de *logs* na ferramenta. De modo a auxiliar a compreensão deste funcionamento, apresenta-se uma `TracingPolicy` de exemplo em 4.14.

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "connect"
spec:
  kprobes:
    - call: "tcp_connect"
      syscall: false
      args:
        - index: 0
        type: "sock"
    selectors:
      - matchArgs:
        - index: 0
        operator: "NotDAddr"
        values:
          - "10.0.0.0/8"
          - "172.16.0.0/12"
          - "192.168.0.0/16"
          - "127.0.0.0/8"
```

Excerto de código 4.14: Exemplo de `TracingPolicy`

A referida política utiliza a função `tcp_connect` para monitorizar conexões a endereços IP que não pertençam às gamas listadas na chave `values` da política. Por exemplo, se algum *node* do *cluster* tentar comunicar, usando o protocolo TCP, com o endereço `1.1.1.1`, isso gerará um alerta por parte do Tetragon, registando o evento sob a forma de *logs*.

Realizada uma exploração inicial deste funcionamento, foram definidas duas políticas: `file-monitoring` e `connect` (esta última sendo uma adaptação do exemplo apresentado). Através do chart `tetragon-policies`, é possível configurar estas duas políticas, utilizando a configuração apresentada em 4.15:

```
1
2  fileMonitoring:
3    enabled: true
4    pathsToMonitor:
5      - /etc/
6      - /var/log/
7
8  networkMonitoring:
9    enabled: true
10   allowedNetworks:
11     - 10.0.0.0/8
12     - 172.16.0.0/12
13     - 192.168.0.0/16
14     - 127.0.0.0/8
15
```

Excerto de código 4.15: Configuração do Chart `tetragon-policies`

Estes valores são interpretados pelo Helm e utilizados para popular um conjunto de *templates* - um por política - com a configuração desejada. no caso de `connect`, por exemplo, o template é definido conforme apresentado em 4.16.

```
1  {{- if .Values.networkMonitoring.enabled -}}
2  apiVersion: cilium.io/v1alpha1
3  kind: TracingPolicy
4  metadata:
5    name: "connect"
6  spec:
7    kprobes:
8      - call: "tcp_connect"
9      syscall: false
10     args:
11     - index: 0
12       type: "sock"
13     selectors:
14       - matchArgs:
15         - index: 0
16           operator: "NotDAddr"
17         values:
18         {{- range .Values.networkMonitoring.allowedNetworks }}
19           - {{ . | quote }}
20         {{- end }}
21 {{- end }}
```

Excerto de código 4.16: *Template* definido para criação de política

Na linha 1, existe a validação que garante que a política apenas é criada se o valor de `enabled` for verdadeiro. Notavelmente, as linhas 18 a 20 interpretam os valores de `allowedNetworks` e populam o *template* com estes, gerando então uma `TracingPolicy` válida, que é criada pelo Helm no *cluster*, sendo então interpretada pelo Tetragon - e gerando eventos nos *logs* quando os critérios são cumpridos. Um exemplo destes registos é apresentado na figura 4.2. Este representa um evento detetado pela política `file-monitoring`, de uma ação ocorrida

no servidor `thesis-m-01`. Trata-se de uma alteração a um ficheiro que reside no caminho `/var/log`, realizada pelo binário `/usr/lib/systemd/systemd-journald`.


```
{
  "process_kprobe": {
    "process": {
      "cwd": "/",
      "binary": "/usr/lib/systemd/systemd-journald",
      "flags": "procFS auid rootcwd",
      "tid": 3237971
    },
    "parent": {
      "binary": "/usr/lib/systemd/systemd",
      "arguments": "--system --deserialize 68"
    },
    "function_name": "security_path_truncate",
    "args": [
      {
        "path_arg": {
          "path": "/var/log/journal/6642118b6c1347d88e7ec391b944146d/system.journal",
          "permission": "-w-r-----"
        }
      }
    ],
    "policy_name": "file-monitoring"
  },
  "node_name": "thesis-m-01",
  "time": "2025-06-20T16:50:07.574467807Z"
}
```


Figura 4.2: Exemplo de *log* do Tetragon, com base nas políticas implementadas

Note-se que os *logs* não são facilmente interpretáveis deste modo, sendo isto algo que é endereçado pela solução final, nomeadamente pela perspetiva de integração com o Grafana - a ferramenta de visualização adotada. Contudo, é possível usar a CLI desenvolvida para o Tetragon, de modo a obter uma leitura mais legível dos eventos, algo que é demonstrado na figura 4.3. Realça-se que o evento apresentado na figura 4.2 é apenas um dos que se encontra neste novo exemplo. Tal se deve ao facto de, no último caso, se apresentar uma pluralidade de eventos capturados pelo Tetragon de forma sumária, contrariamente ao formato extenso do exemplo singular apresentado previamente.

```
luis@luis-Zenbook-UM3402YA-UM3402YA:~$ kubectl logs -n security -l app.kubernetes.io/name=tetragon -c export-stdout -f | tetra getevents -o compact
read    thesis-m-01 /usr/sbin/cron /etc/login.defs
read    thesis-m-01 /usr/sbin/cron /etc/login.defs
write   thesis-m-01 /usr/sbin/auditd /var/log/audit/audit.log
write   thesis-m-01 /usr/sbin/auditd /var/log/audit/audit.log
truncate thesis-m-01 /lib/systemd/systemd-journald /var/log/journal/6642118b6c1347d88e7ec391b944146d/system.journal
write   thesis-m-01 /usr/sbin/rsyslogd /var/log/auth.log
read    thesis-m-01 /usr/bin/python3.10 /var/log/auth.log
read    thesis-m-01 /usr/bin/python3.10 /var/log/auth.log
read    thesis-m-01 /usr/bin/python3.10 /var/log/auth.log
truncate thesis-m-01 /lib/systemd/systemd-journald /var/log/journal/6642118b6c1347d88e7ec391b944146d/system.journal
read    thesis-w-01 /opt/cni/bin/calico-ipam /etc/cni/net.d/calico-kubeconfig
write   thesis-w-01 /opt/cni/bin/calico-ipam /var/log/calico/cni/cni.log
write   thesis-w-01 /opt/cni/bin/calico-ipam /var/log/calico/cni/cni.log
write   thesis-w-01 /opt/cni/bin/calico-ipam /var/log/calico/cni/cni.log
read    thesis-w-01 /opt/cni/bin/calico-ipam /var/log/calico/cni
read    thesis-w-01 /opt/cni/bin/calico-ipam /var/log/calico/cni
write   thesis-w-01 /opt/cni/bin/calico-ipam /var/log/calico/cni/cni.log
write   thesis-w-01 /opt/cni/bin/calico-ipam /var/log/calico/cni/cni.log
write   thesis-w-01 /opt/cni/bin/calico-ipam /var/log/calico/cni/cni.log
write   thesis-w-01 /opt/cni/bin/calico /var/log/calico/cni/cni.log
```

Figura 4.3: *Logs* do Tetragon utilizando a CLI

Conclui-se então a implementação do Tetragon no âmbito da solução desenvolvida. A configuração quer da ferramenta em si, quer de políticas para a mesma é realizada de modo centralizado através de *values* - recorrendo ao Helm. Essas políticas geram *logs*, que devem ser interpretados no processo de deteção de eventos. Estes dois factos possibilitam a integração desta ferramenta com as restantes, no âmbito de uma solução final - quer em termos de configuração unificada, quer em agregação de dados de modo centralizado.

4.2.4 Implementação do Trivy

No escopo da solução final, o Trivy desempenha o papel de deteção de vulnerabilidades, sejam elas aplicacionais (CVEs), quer sejam vulnerabilidades de configuração. Adicionalmente, é também capaz de analisar o *cluster* com base em padrões de *compliance*, reportando esses mesmos resultados.

Conforme mencionado anteriormente, o Trivy permite instalação e configuração através de um Helm Chart - remetendo para o ponto de unificação de configuração apresentado anteriormente. Como tal, esta sub-secção visa apresentar a configuração utilizada para este componente, para que sirva os propósitos pretendidos.

O *values* do Trivy assume bastantes configurações, pelo que apenas serão exploradas aquelas verdadeiramente relevantes para a solução. Começando pela implementação do *operator* do Trivy - um componente que permite controlar que tipos de análises conduzir - e que se apresenta em 4.17:

```
operator:
  vulnerabilityScannerEnabled: true
  sbomGenerationEnabled: true
  configAuditScannerEnabled: true
  rbacAssessmentScannerEnabled: true
  infraAssessmentScannerEnabled: true
  clusterComplianceEnabled: true
  accessGlobalSecretsAndServiceAccount: true
  metricsFindingsEnabled: true
  metricsVulnIdEnabled: false
  exposedSecretScannerEnabled: true
  metricsExposedSecretInfo: false
  metricsConfigAuditInfo: false
  metricsRbacAssessmentInfo: false
  metricsClusterComplianceInfo: false
```

Excerto de código 4.17: Configuração do *operator* do Trivy

Esta secção do *values* consiste essencialmente de definições binárias, escolhendo que componentes do *operator* se pretendem ativar. É importante realçar que todas as análises estão ativas, porém existem algumas métricas respetivas a estas que estão desativas. Deve-se ao facto de estas serem métricas `info`, ou seja, com informação granular sobre cada resultado em cada análise. Em abstrato, seria extremamente vantajoso ativar todas estas, contudo, a granularidade da informação também aumenta a cardinalidade das métricas expostas, algo que pode ser problemático para a integração com o Prometheus, nomeadamente no que toca a consumo de recursos por parte deste. Este tema será explorado na secção de testes à solução. Contudo, sendo uma ferramenta configurável via Helm - e, subsequentemente, integrada com a configuração das restantes ferramentas - os administradores podem ativar

ou desativar estas *flags* conforme desejado. Por exemplo, em certos contextos poderá ser necessário saber informação granular sobre os relatórios de *compliance*.

De seguida, foram alteradas algumas chaves do *values* respetivas ao processo do Trivy. Consistem apenas de definições simples - apresentadas em 4.18 - que delimitam a abrangência de alguns *scans*.

```
1 trivy:
2   createConfig: true
3   severity: UNKNOWN,LOW,MEDIUM,HIGH,CRITICAL
4   slow: true
5   ignoreUnfixed: false
6   supportedConfigAuditKinds: "Workload,Service,Role,ClusterRole,
     NetworkPolicy,Ingress,LimitRange,ResourceQuota"
```

Excerto de código 4.18: Configuração do processo do Trivy

A chave `severity`, na linha 3, define que todas as vulnerabilidades devem ser reportadas - pois apresenta uma lista com todos os valores possíveis. A sua inclusão visa apresentar a possibilidade de restringir as vulnerabilidades reportadas consoante o seu nível de severidade. No caso de `supportedConfigAuditKinds` na linha 6, esta é usada para definir que tipos de recursos serão analisados pelo *scanner* de configurações do Trivy - o `configAuditScannerEnabled`, ativado na linha 4 em 4.17.

Finalmente, foi configurado o módulo de *compliance* do Trivy - que executa o `kube-bench`, conforme investigado no decorrer da revisão de literatura. O resultado deste processo apresenta-se em 4.19.

```
1 compliance:
2   reportType: summary
3   cron: 0 */6 * * *
4   specs:
5     - k8s-cis-1.23
```

Excerto de código 4.19: Configuração do módulo de *compliance* do Trivy

A chave `reportType` da linha 2 pode assumir dois valores - `summary` ou `all`. Para o projeto, foi utilizada a opção de gerar um relatório sumário, devido à preocupação com o consumo de recursos previamente descrita. Em `cron` (linha 3), configura-se a frequência mediante a qual executar as análises de *compliance* do *cluster*. Por fim, `specs` permite listar que conjuntos de análises devem ser executados. No atual caso, adotou-se a execução dos Center for Internet Security (CIS) *benchmarks* desenvolvidos para Kubernetes, sendo esta definição efetuada nas linhas 4 e 5 do referido excerto.

Com a perspetiva de integração em mente, configurou-se também o `ServiceMonitor` nos *values* do Trivy, conforme exposto em 4.20. Ao criar o ServiceMonitor, a instância de Prometheus será automaticamente capaz de realizar a auto-descoberta desta ferramenta, bem como das métricas por ela exportadas, armazenando-as. Este `ServiceMonitor` foi criado no mesmo *namespace* onde será instalado o Prometheus, para garantir que a conexão entre os dois é realizada com sucesso.

```
serviceMonitor:
  enabled: true
  namespace: security
```

Excerto de código 4.20: Configuração `ServiceMonitor` para recolha de métricas do Trivy

Concluída a configuração e instalação do Trivy, foi possível verificar a geração dos relatórios por parte do mesmo. Estes são gerados sob a especificação de CRDs criados pela ferramenta, e podem ser consultados no próprio *cluster*. Na figura 4.4 é possível observar os relatórios de vulnerabilidades gerados para cada imagem que está a ser executada no *cluster*, reportando-as de modo agrupado por severidade. Já em 4.5, apresenta-se o sumário da postura de *compliance* do *cluster*.

vulnerabilityreports(security)[16]

NAME	REPOSITORY	TAG	SCANNER	CRITICAL	HIGH	MEDIUM	LOW	UNKNOWN	AGE↓
statefulset-prometheus-kube-prometheus-prometheus-prometheus	bitnami/prometheus	2.53.1-debian-12-r1	Trivy	7	23	65	76	6	22h
statefulset-cbc85cd6b	bitnami/prometheus-operator	0.75.2-debian-12-r1	Trivy	3	19	59	73	6	22h
statefulset-7997ffc9fc	bitnami/alertmanager	0.27.0-debian-12-r17	Trivy	4	20	67	73	6	22h
statefulset-76dbd8857b	bitnami/prometheus-operator	0.75.2-debian-12-r1	Trivy	3	19	59	73	6	22h
statefulset-68c8c5fc7c	bitnami/prometheus-operator	0.75.2-debian-12-r1	Trivy	3	19	59	73	6	22h
statefulset-5cd4955b67	bitnami/prometheus-operator	0.75.2-debian-12-r1	Trivy	3	19	59	73	6	22h
replicaset-76944dd6c5	bitnami/prometheus-operator	0.75.2-debian-12-r1	Trivy	3	19	59	73	6	22h
replicaset-7856b5d444	bitnami/kube-state-metrics	2.13.0-debian-12-r2	Trivy	3	18	50	73	4	22h
replicaset-5bdd75d764	bitnami/blackbox-exporter	0.25.0-debian-12-r12	Trivy	3	18	50	73	4	22h
daemonset-kube-prometheus-node-exporter-node-exporter	bitnami/node-exporter	1.8.2-debian-12-r1	Trivy	3	18	50	73	4	22h
replicaset-tetragon-operator-7fddfd8fb7-tetragon-operator	cilium/tetragon-operator	v1.4.0	Trivy	0	0	4	0	2	21h
daemonset-tetragon-tetragon	cilium/tetragon	v1.4.0	Trivy	0	0	16	0	10	21h
daemonset-tetragon-export-stdout	cilium/hubble-export-stdout	v1.0.4	Trivy	0	0	0	0	0	21h
daemonset-fluent-bit-fluent-bit	fluent/fluent-bit	4.0.1	Trivy	1	2	15	42	0	21h
replicaset-trivy-operator-54f8c89d66-trivy-operator	aquasec/trivy-operator	0.26.1	Trivy	0	0	1	0	2	20h

Figura 4.4: Relatórios de vulnerabilidades gerados pelo Trivy

clustercompliancereports(all)[1]

NAME↑	FAIL	PASS
k8s-cis-1.23	13	103

Figura 4.5: Sumário de *compliance* gerado pelo Trivy

É evidente que a informação sumária apenas oferece uma indicação básica da situação de segurança do ambiente Kubernetes em questão. Manipulando as definições do *operator* em 4.17 os dados gerados são mais granulares na forma de métricas. Por exemplo, ativando o `metricsClusterComplianceInfo`, o Pod do Trivy expõe métricas relativas a cada teste do *benchmark* escolhido, reportando quantas falhas existem por teste. No excerto 4.21, apresenta-se um exemplo de uma destas métricas. O nome da métrica em questão é `trivy_compliance_info`, e assume um conjunto de *labels*, que permitem interpretar o teste que esta efetua, bem como o seu resultado. Trata-se do teste com o identificador 4.1.5 do CIS *benchmark* que, tal como o valor da *label* `compliance_name` indica, verifica se o ficheiro `kubelet.conf` assume permissões de valor 600, ou mais restritas. O atributo `severity` permite categorizar os diferentes testes pelo seu nível de importância no *cluster*, sendo o presente exemplo um caso altamente relevante, dado assumir o valor `HIGH`. Finalmente, esta métrica cataloga o sucesso deste teste - algo verificado pela *label* `status`. Ora, sendo que o valor da métrica é 1 - conforme visível no final da mesma, e o `status` é definido como `Pass`, conclui-se que este teste suscitou um resultado positivo, ou seja, o ficheiro `kubelet.conf` está com as permissões estipuladas pelo CIS *benchmark*.

```
trivy_compliance_info{compliance_id="4.1.5",compliance_name="Ensure
    that the --kubeconfig kubelet.conf file permissions are set to
   600 or more restrictive",description="CIS Kubernetes Benchmarks"
   ,severity="HIGH",status="Pass",title="CIS Kubernetes Benchmarks
   v1.23"} 1
```

Excerto de código 4.21: Exemplo de métrica exposta pelo Trivy

O objetivo destas métricas não é serem interpretadas diretamente pelos administradores, mas sim recolhidas e armazenadas pelo Prometheus, e subsequentemente apresentadas de modo organizado e coerente no Grafana, possibilitando uma visão mais legível dos relatórios de cada *scan*, bem como a subsequente correlação destes dados com os apresentados pelas restantes ferramentas.

Configuradas as ferramentas de análise contempladas para a presente prova de conceito, passar-se-á à implementação de mecanismos de recolha e armazenamento dos *outputs* destas ferramentas, começando-se pelo Prometheus.

4.2.5 Implementação do Prometheus

O Prometheus desempenha um papel integral na solução, sendo este o componente responsável pela recolha de métricas das diversas ferramentas, centralizando esta informação e tornando-a disponível para visualização - integração essa que será descrita posteriormente.

Mantendo a linha de raciocínio dos restantes componentes, o Prometheus foi configurado e instalado através de um Helm Chart. Este serviço requereu poucas alterações à sua configuração inicial. Tal deve-se ao facto de o Chart utilizado - `bitnami/kube-prometheus` - configurar esta ferramenta sob a forma de *operator*. Deste modo, o Prometheus é definido como sendo um conjunto de CRDs, notavelmente: o recurso `Prometheus`, que define a instância do mesmo; `ScrapeConfig`, que especifica uma configuração para recolha de métricas, que deve ser interpretada pela ferramenta; e `ServiceMonitor`, que define a monitorização para um ou mais serviços. Ora, a importância do último recurso apresentado é notável. Utilizando `ServiceMonitors`, é possível reduzir a configuração que seria necessária para o *scrape* ao nível do Prometheus, delegando esta responsabilidade aos serviços dos quais se pretendem recolher métricas. Como explicitado em secções anteriores, estes recursos foram ativos quer para o Trivy, quer para o Tetragon, fazendo com que estas métricas sejam automaticamente recolhidas pelo Prometheus - pois este analisa os `ServiceMonitors` de um *namespace*, e utiliza-os para a auto-descoberta de métricas a recolher.

A utilização de *scrape* por via de `ServiceMonitors` não só tem benefícios no que toca ao minimalismo da configuração, como também de escalabilidade. Se alguma nova ferramenta for adicionada à solução, basta configurar um `ServiceMonitor` para esta, garantindo que as suas métricas são automaticamente recolhidas pelo Prometheus.

Alguns `ServiceMonitors` internos encontram-se ativos aquando da instalação do Prometheus. Ora, estes não são utilizados pela ferramenta - pois monitorizam recursos que não dizem respeito ao âmbito da mesma. Como tal, estes foram eliminados através do *values*, conforme explícito no excerto 4.22.

```
kubeApiServer:
  enabled: false

kubelet:
  enabled: false
  serviceMonitor:
    enabled: false
```

Excerto de código 4.22: Eliminação de `ServiceMonitors` desnecessários

Posteriormente à instalação do Prometheus, é possível consultar a interface gráfica disponibilizada pelo mesmo, de modo a aferir o seu funcionamento. Na figura 4.6, é possível verificar que os referidos recursos de monitorização estão a ser interpretados devidamente pela ferramenta, concluindo-se então que esta está a recolher métricas provenientes das restantes ferramentas de monitorização e a armazená-las. É disponibilizada, também, informação quanto ao sucesso ou insucesso do processo de recolha - no caso apresentado, esta foi bem sucedida para todos os *targets* definidos pelo `ServiceMonitor`.

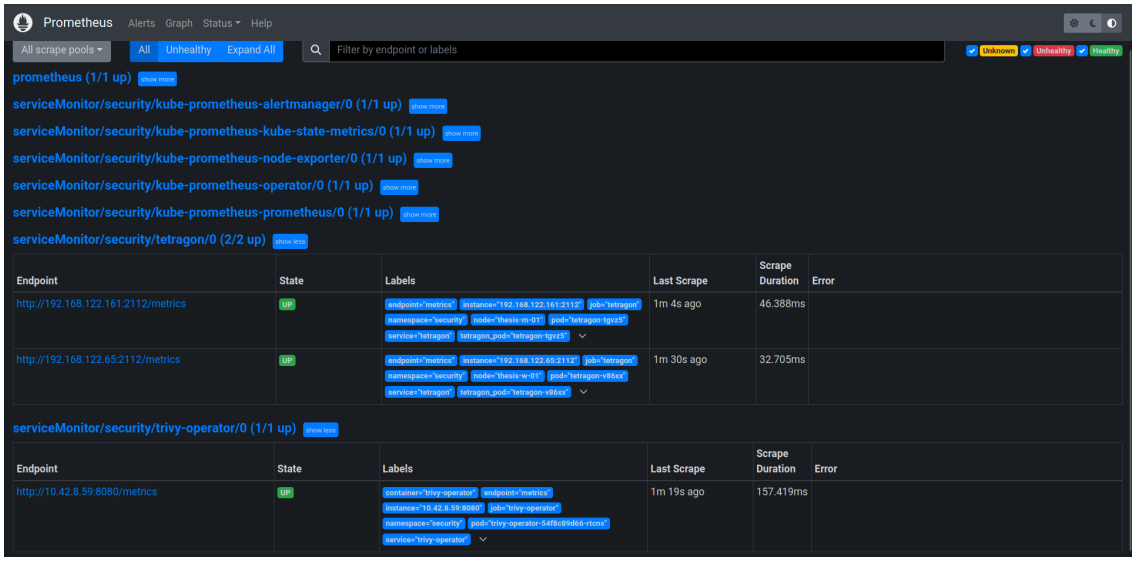

Figura 4.6: `ServiceMonitors` interpretados pelo Prometheus

Assim, o Prometheus encontra-se totalmente implementado no contexto da solução desenvolvida, e realiza uma parte integral no processo de integração entre ferramentas e os seus dados, como foi possível constatar neste último segmento. Estando articulada a integração de recolha de métricas, resta replicar esta função no caso da recolha de *logs*, algo que será explorado na subsecção seguinte - a implementação do `fluent-bit`.

4.2.6 Implementação do `fluent-bit`

Como mencionado anteriormente, o `fluent-bit` realiza a importante função de recolher os *logs* emitidos pelas diversas ferramentas, agregá-los, e encaminhá-los para uma base de dados - no caso do atual projeto, o Elasticsearch, cuja implementação será explorada posteriormente.

A configuração desta recolha pelo `fluent-bit` divide-se essencialmente em três blocos fundamentais: os *inputs*, que definem quais são os *logs* que devem ser recolhidos; os filtros,

que permitem processar os *logs*, aplicando-lhes diversas transformações; e os *outputs*, que configuram para onde devem ser enviados os *logs* transformados. Tratando-se de uma ferramenta com instalação por Helm, o seu Chart permite a configuração destes blocos no seu ficheiro de *values*.

Primeiramente, discorrer-se-à sobre a secção de inputs, apresentada no excerto 4.23.

```
config:
  inputs: |
    [INPUT]
    Name tail
    Path /var/log/containers/*tetragon*.log
    multiline.parser docker, cri
    Tag sec-tetragon.*
    Parser cri
    Mem_Buf_Limit 50MB

    [INPUT]
    Name tail
    Path /var/log/containers/*kube-score*.log
    multiline.parser docker, cri
    Tag sec-kube-score.*
    Mem_Buf_Limit 50MB
```

Excerto de código 4.23: Configuração de recolha de *logs* no `fluent-bit`

É possível verificar que são definidos dois *inputs*: um para *logs* provenientes do `kube-score`; outro para aqueles emitidos pelo Tetragon. A cada um destes é atribuída uma *tag*, que será utilizada ao nível dos filtros. Adicionalmente, definem-se alguns *parsers*: a funcionalidade destes é a deteção automática dos diversos campos existentes nos *logs* e a sua separação, algo que é extremamente útil para fases posteriores, nomeadamente a leitura e interpretação dos *logs* na ferramenta de visualização escolhida.

Definidos os *inputs*, foram então configurados os respetivos filtros - um para cada - que se apresentam no excerto 4.24.

```
config:
  filters: |
    [FILTER]
    Name kubernetes
    Match sec-tetragon.*
    Kube_Tag_Prefix sec-tetragon.var.log.containers.
    Merge_Log On
    K8S-Logging.Parser On
    K8S-Logging.Exclude On

    [FILTER]
    Name kubernetes
    Match sec-kube-score.*
    Kube_Tag_Prefix sec-kube-score.var.log.containers.
    Merge_Log On
    K8S-Logging.Parser On
    K8S-Logging.Exclude On
```

Excerto de código 4.24: Configuração de filtros no `fluent-bit`

Os filtros definidos são bastante simples, e apenas realizam processamento interno ao `fluent-bit` utilizado na definição posterior dos *outputs*. Novamente, configura-se o *parser* ao nível de cada filtro, bem como se adiciona um prefixo por *input*, estabelecendo a correspondência com o `Path` definido.

Finalmente, são configurados os *outputs* que se apresentam em 4.25.

```
config:
  outputs: |
  [OUTPUT]
  Name es
  Match sec-tetragon*
  Host 192.168.122.146
  Port 9200
  Index sec-tetragon
  Suppress_Type_Name On
  Retry_Limit 3
  Replace_Dots On
  HTTP_User ${elastic-user}
  HTTP_Passwd ${elastic-password}
  Trace_Error On
  tls On
  tls.verify On
  tls.verify_hostname Off
  Generate_ID on
  Write_Operation upsert

  [OUTPUT]
  Name es
  Match sec-kube-score*
  Host 192.168.122.146
  Port 9200
  Index sec-kube-score
  Suppress_Type_Name On
  Retry_Limit 3
  Replace_Dots On
  HTTP_User ${elastic-user}
  HTTP_Passwd ${elastic-password}
  Trace_Error On
  tls On
  tls.verify On
  tls.verify_hostname Off
  Generate_ID on
  Write_Operation upsert
```

Excerto de código 4.25: Configuração de *outputs* no `fluent-bit`

Seguindo o padrão definido anteriormente, existem dois *outputs* - um por cada *input*, sendo esta correlação feita através da *tag* definida para cada. No campo de `Host`, é definido o servidor de Elasticsearch para qual enviar os *logs*. Note-se que a definição é realizada pelo endereço IP por limitações do ambiente de testes - no cenário de produção, a identificação seria realizada por um nome de DNS. O servidor destino requer autenticação para a escrita de *logs*, sendo esse propósito servido pelas chaves de configuração `HTTP_User` e `HTTP_Passwd`. Por motivos de segurança, o valor destes dois argumentos não é codificado em texto claro,

mas sim lido de um Secret, que é interpretado pelo `fluent-bit`, carregando então o valor destes segredos sob a forma de variáveis de ambiente, que podem ser interpretadas pelo processo. Ainda sob o tema de preocupações de segurança, a configuração `tls` encontra-se ativa, garantindo que as conexões a este Elasticsearch externo são encriptadas utilizando TLS. Adicionalmente, a configuração `tls.verify` garante que os *logs* apenas serão enviados caso o certificado apresentado pelo servidor de destino seja válido. Note-se, no entanto, que `tls.verify_hostname` está desligado. Tal deve-se, novamente, a limitações associadas à ausência de um nome de DNS para o servidor, de modo a que o certificado apresentado é válido (gerado por um *issuer* confiável e encontra-se dentro do prazo de validade), porém não diz respeito ao *host* que o está a usar. Finalmente, vale realçar a configuração `index`, que especifica para que índice do Elasticsearch os *logs* serão escritos. A utilização e gestão de índices permite uma subsequente filtragem mais ágil, conforme será analisado posteriormente.

Conclui-se, então, a configuração e instalação do `fluent-bit`, passando-se de seguida para a implementação do componente responsável pela receção destes *logs* - o Elasticsearch.

4.2.7 Implementação do Elasticsearch

Conforme descrito na secção 4.1, a instância de Elasticsearch utilizada foi externalizada do *cluster* - sendo então disponibilizada por um servidor distinto, na mesma rede interna. Sendo a sua instalação realizada através de uma *role* de Ansible proprietária da organização, esta não poderá ser amplamente detalhada. Este processo também instala uma instância de Kibana - um *frontend* para visualização de dados do Elasticsearch. Ora, este componente não deve fazer parte da solução, visto desejar-se a centralização da visualização de dados no Grafana. Contudo, foi útil para o processo de configuração necessária para a utilização do Elasticsearch em conjunção com as restantes ferramentas.

Esta base de dados é aprovisionada com um utilizador - `elastic` - por defeito, com permissões de administrador em todos os componentes. Numa perspetiva de segurança, não é sensato recorrer a este utilizador, pois as suas permissões são excessivas face àquelas que são verdadeiramente necessárias para o funcionamento da plataforma.

Para abordar este problema, foi criado um novo utilizador, bem como uma nova Role a ser assumida por este. Na figura 4.7, é possível ver as permissões da nova Role criada - pode efetuar todas as ações em índices começados por `sec`, e pode gerir a instância de Elasticsearch. Ainda que pareça excessiva, esta última permissão é necessária, pois o `fluent-bit` necessita de poder criar os índices, definir as suas propriedades, e efetuar outras operações de gestão.

Definida a Role, foi criado o novo utilizador que a assume como atributo, conforme apresentado em 4.8.

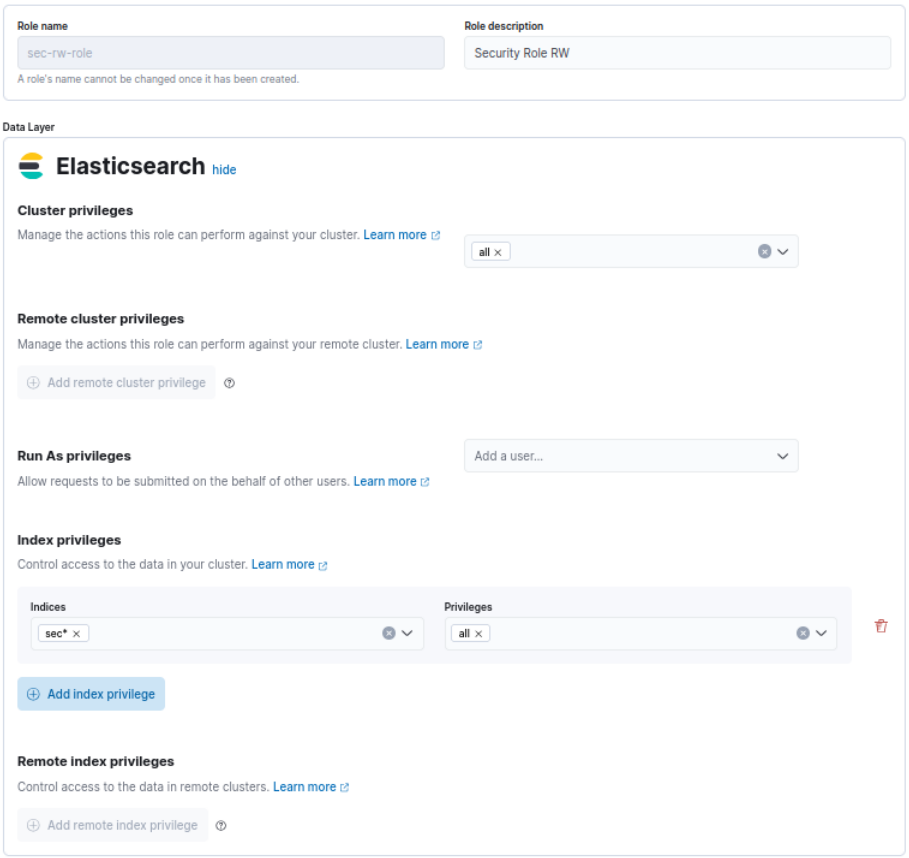

Figura 4.7: Role criada para utilizador no Elasticsearch

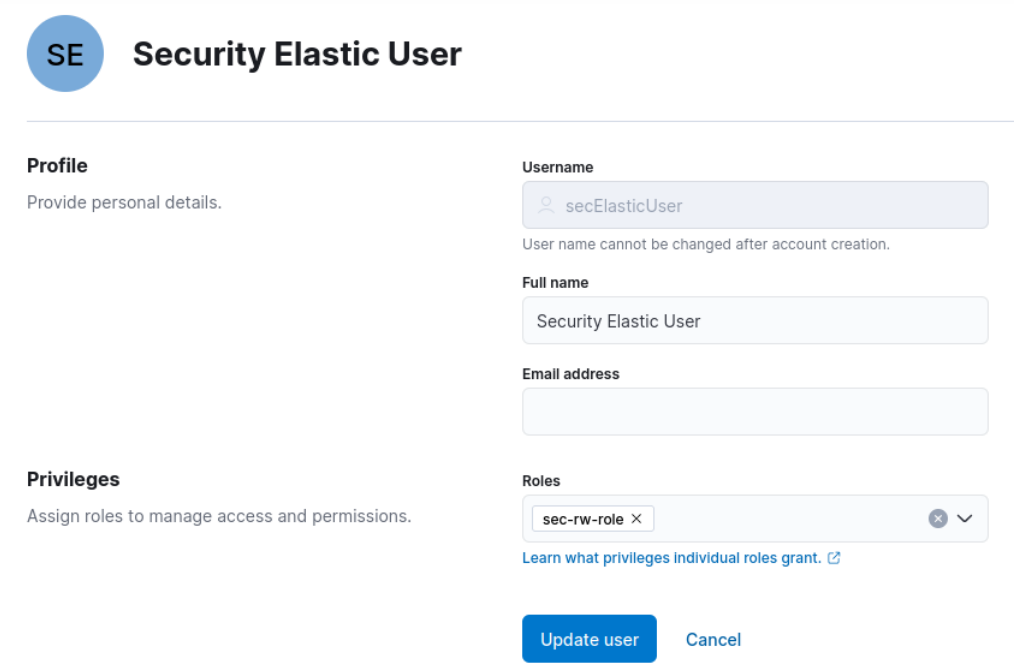

Figura 4.8: Utilizador criado no Elasticsearch

Aquando desta criação - e da configuração do Secret com credenciais, descrito em 4.2.6, o Elasticsearch recebeu automaticamente *logs* provenientes do `fluent-bit` - por sua vez, recolhidos dos diversos componentes da solução. Para efeitos de confirmação, o Kibana foi utilizado para consultar os índices existentes na base de dados, confirmando-se que os dois especificados nos *outputs* do `fluent-bit` se encontram criados e populados com documentos (ou seja, entradas de *logs*). Esta confirmação apresenta-se na figura 4.9.

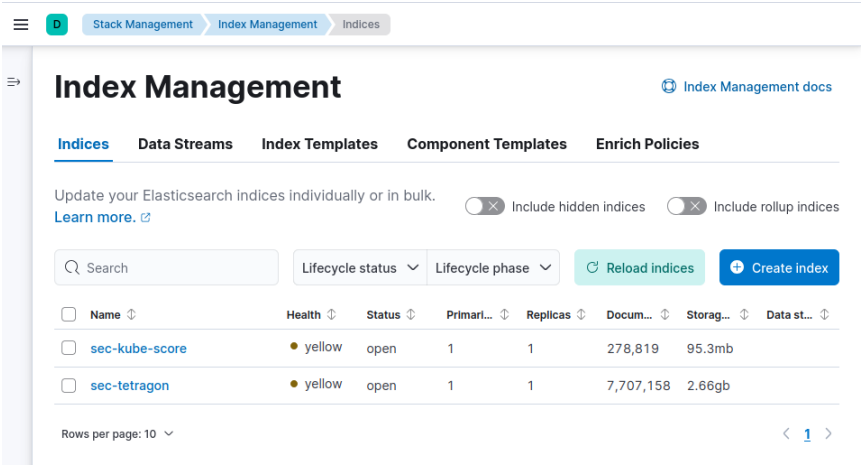

Figura 4.9: Índices criados pelo `fluent-bit` no Elasticsearch

Como tal, encontra-se configurado o Elasticsearch no âmbito do projeto, pelo que apenas resta a instalação e configuração da ferramenta utilizada para a visualização de todos os dados - o Grafana.

4.2.8 Implementação do Grafana

Este componente foi o último a ser configurado na prova de conceito desenvolvida, sendo ele o principal responsável pela interligação entre os dados recolhidos - mais concretamente, as bases de dados onde estes se armazenam. A sua configuração foi realizada por duas vias: pela *values*, através do seu Helm Chart para instalação em Kubernetes; e manualmente, na interface do utilizador disponibilizada por esta ferramenta. Idealmente, toda a configuração seria programática e versionada, não requerendo passos manuais - algo que será discorrido durante a avaliação da solução. Contudo, nesta fase de desenvolvimento, adotou-se esta abordagem mista para a obtenção de um resultado funcional, que possa ser utilizado no processo de análise e avaliação.

No *values*, as alterações foram mínimas. Primeiramente, foi configurado um utilizador `admin` para a gestão da ferramenta, conforme apresentado no excerto 4.26. Vale realçar que a *password* do utilizador não está especificada em texto-claro, mas sim é obtida através de um Secret com essa informação - melhorando a postura de segurança do código da solução.

```
admin:
  user: "admin"
  existingSecret: "grafana-credentials"
  existingSecretPasswordKey: password
```

Excerto de código 4.26: Configuração do utilizador `admin` no Grafana

Para além desta configuração foi também definido um volume persistente, para que o Grafana não perca dados sobre os *dashboards* criados, bem como sobre os *datasources* que estes utilizam - no caso, Elasticsearch e Prometheus.

```
persistence:
  enabled: true
  storageClass: "local-path"
  accessMode: ReadWriteOnce
  size: 10Gi
```

Excerto de código 4.27: Configuração de persistência no Grafana

Posto isto, foi realizada a instalação do grafana, que disponibilizou a referida interface. Nesta, foram configurados os *datasources* - isto é, as bases de dados que o Grafana deve consultar. Primeiramente, foi configurado o acesso ao Prometheus conforme apresentado na figura 4.10. Note-se que, visto que ambos são executados no mesmo *cluster*, a comunicação pode ser realizada através dos nomes de DNS internos ao *cluster* definidos pela arquitetura do Kubernetes. Como tal, o URL de acesso ao Prometheus é definido como `kube-prometheus-prometheus.security.svc.cluster.local:9090` - ou seja, o nome do objeto `Service` criado pelo Chart do Prometheus, seguido do *namespace* onde este se insere.

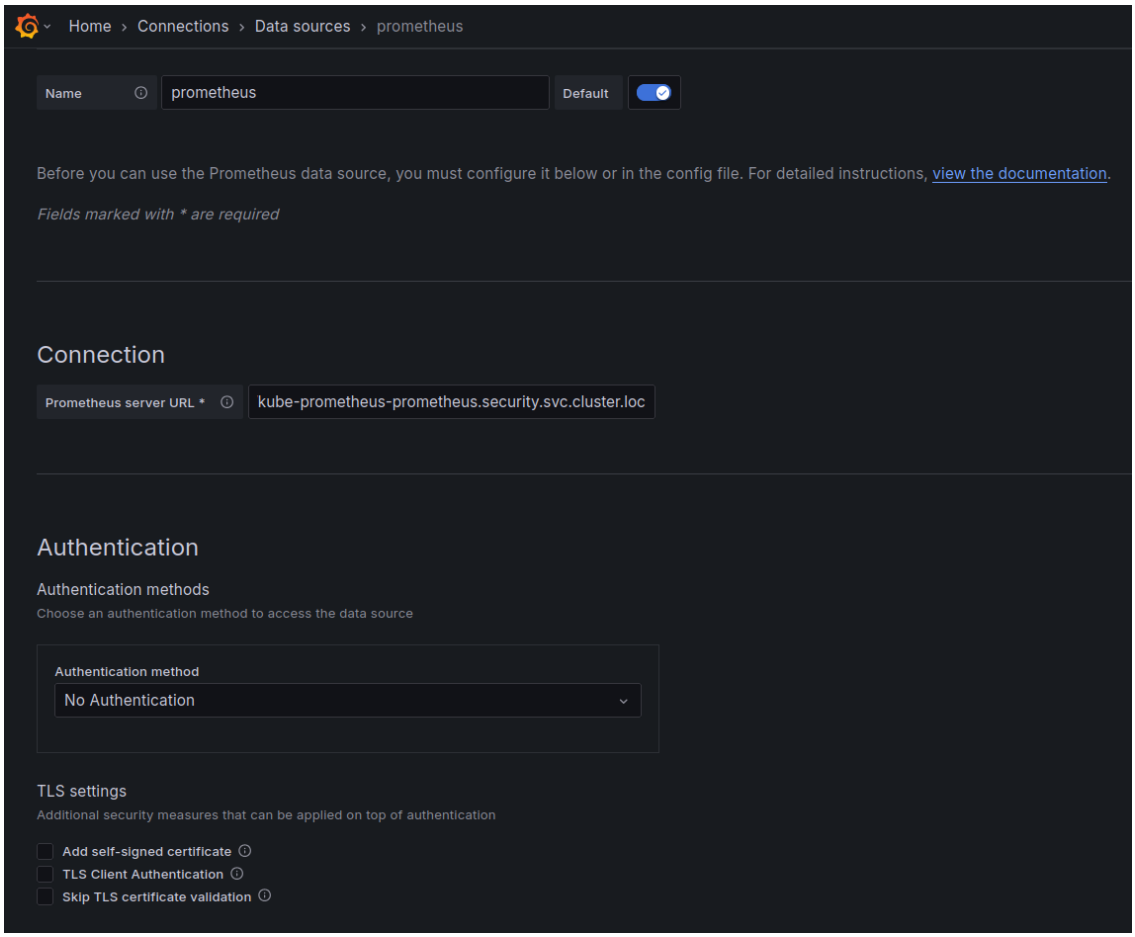

Figura 4.10: Configuração da *datasource* para o Prometheus

De seguida, foi configurada a *datasource* para acesso ao Elasticsearch - e subsequente recolha de *logs*, conforme apresentado em 4.11. Note-se a utilização de TLS mas sem validação do certificado, por limitações do ambiente de testes expressas anteriormente. Vale também realçar que o utilizador mencionado aquando da preparação do Elasticsearch está a ser devidamente usado pelo Grafana, sendo essencial para esta conexão.

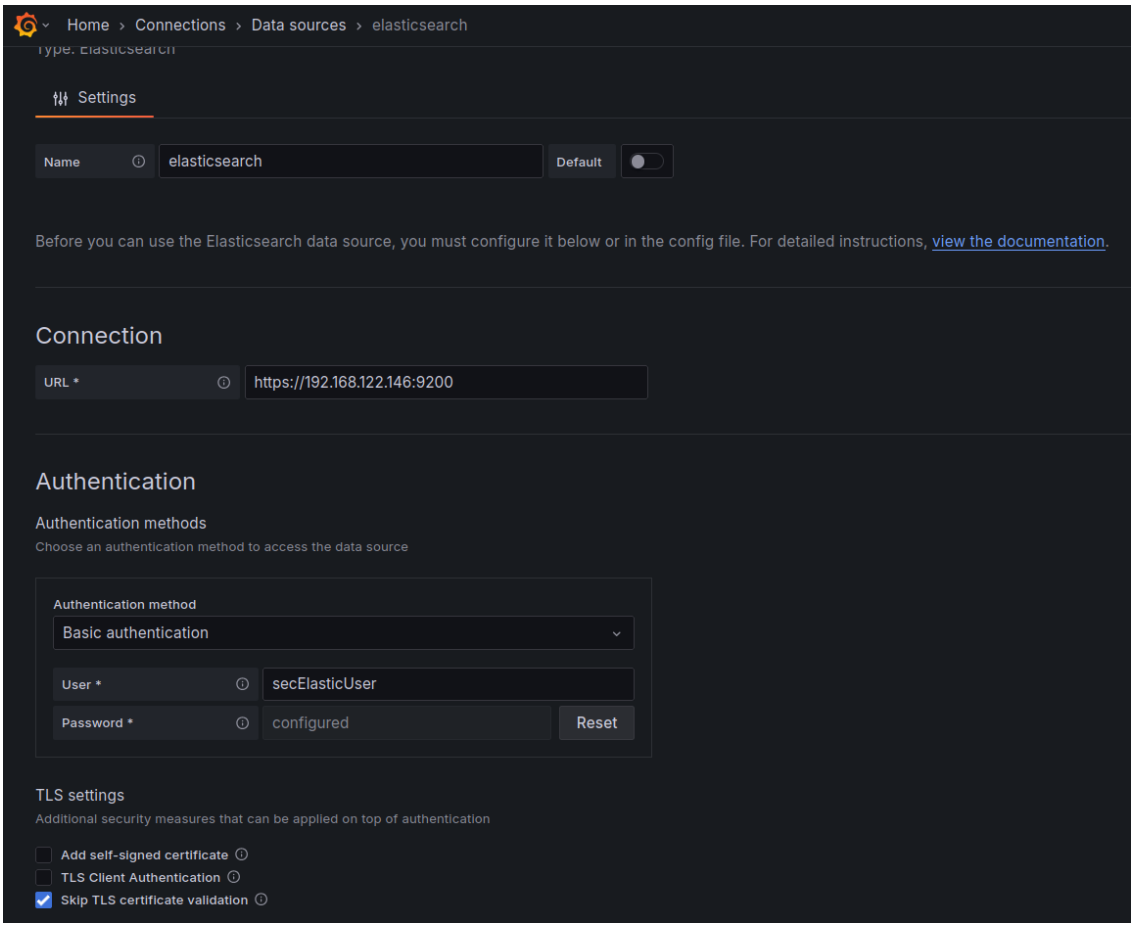

Figura 4.11: Configuração da *datasource* para o Elasticsearch

Deste modo, encontra-se apresentada toda a implementação de componentes realizada no âmbito do atual projeto, tendo já relevado um certo nível de integração entre os mesmos. Este tema será discutido em maior extensão quer na seguinte subsecção, quer na avaliação da solução.

4.3 Análise de resultados

Tendo sido implementada uma prova de conceito da solução final, é agora relevante proceder à sua avaliação. Para esse efeito, foi definido um conjunto de testes, simulando cenários reais e relevantes para o contexto de negócio. Esses testes consistem na simulação de ações que devem gerar deteções por cada um dos componentes, tendo como objetivo confirmar se essa deteção se sucede, e com que grau de sucesso.

Os testes foram definidos e realizados com um foco por componente: `kube-score`, Trivy e Tetragon. Contudo - e sendo a centralização da visualização de dados um ponto significativo do projeto - a confirmação destes foi sempre realizada através do Grafana. Assim, confirma-se para cada componente não só a deteção dos incidentes esperados, como também o fluxo de dados entre a ferramenta, a base de dados respetiva (Prometheus ou Elasticsearch) e a ferramenta de visualização (Grafana).

Apresentar-se-ão agora os testes realizados à solução, começando pela avaliação do processo de encriptação de Secrets, recorrendo ao KMSv2.

4.3.1 Testes à encriptação de Secrets

Para testar o mecanismo implementado, o processo de teste foi bastante linear. Previamente à ativação deste sistema, é feita a criação um Secret, bem como a subsequente consulta no `etcd`, validando que o mesmo está em texto claro. De seguida, ativa-se o mecanismo KMSv2 no API Server, realiza-se a transformação deste Secret para que fique encriptado, e conduz-se a mesma consulta ao `etcd`. Caso o resultado seja diferente, e se confirme que deixa de ser possível interpretar o valor do segredo diretamente, o teste é bem sucedido.

Primeiramente, procedeu-se então à criação de um Secret exemplo, conforme apresentado no texto 4.28. Note-se que o valor secreto não foi redigido deste fragmento, dado ser temporário, como também importante para a validação dos resultados.

```
1 $ kubectl create secret generic secret-test -n default --from-
    literal=demoKey=MyS3cr3tV4lue
```

Excerto de código 4.28: Criação de Secret para testes

De seguida, foi consultado o `etcd` para conduzir a validação inicial. Primeiramente, foi estabelecida uma conexão SSH ao *master node*, para que se possa utilizar o utilitário `etcd-client` para efetuar a consulta, enviando-se como argumento os certificados necessários para o estabelecimento da conexão. O comando utilizado para a consulta necessária ao processo de teste é apresentado no excerto de código 4.29. Vale realçar a utilização do argumento "`get /registry/secrets/default/secret-test`", pois este especifica que se pretende aceder ao Secret do *namespace* `default` com o nome `secret-test`. O resultado da consulta é, então, apresentado na figura 4.12. Nesta, é possível identificar perfeitamente o segredo configurado - `MyS3cr3tV4lue`, extraindo-se também informação sobre o nome do objeto Secret (`secret-test`), bem como da chave que armazena o valor secreto (`demoKey`).

```
$ ETCDCTL_API=3 etcdctl --cacert="/var/lib/rancher/k3s/server/
  tls/etcd/server-ca.crt" --cert="/var/lib/rancher/k3s/server/
  tls/etcd/server-client.crt" --key="/var/lib/rancher/k3s/
  server/tls/etcd/server-client.key" --endpoints="https
  ://127.0.0.1:2379" get /registry/secrets/default/secret-test
  | hexdump -C
```

Excerto de código 4.29: Comando para consulta de Secret no `etcd`

```
00000000  2f 72 65 67 69 73 74 72  79 2f 73 65 63 72 65 74  |/registry/secret|
00000010  73 2f 64 65 66 61 75 6c  74 2f 73 65 63 72 65 74  |s/default/secret|
00000020  2d 74 65 73 74 0a 6b 38  73 00 0a 0c 0a 02 76 31  |-test.k8s.....v1|
00000030  12 06 53 65 63 72 65 74  12 da 01 0a b5 01 0a 0b  |..Secret........|
00000040  73 65 63 72 65 74 2d 74  65 73 74 12 00 1a 07 64  |secret-test....d|
00000050  65 66 61 75 6c 74 22 00  2a 24 32 31 38 32 63 66  |efault".*$2182cf|
00000060  61 64 2d 65 64 63 37 2d  34 34 33 36 2d 61 32 33  |ad-edc7-4436-a23|
00000070  63 2d 35 38 62 63 38 61  33 32 66 36 33 66 32 00  |c-58bc8a32f63f2.|
00000080  38 00 42 08 08 db c7 d5  c2 06 10 00 8a 01 64 0a  |8.B...........d.|
00000090  0e 6b 75 62 65 63 74 6c  2d 63 72 65 61 74 65 12  |.kubectl-create.|
000000a0  06 55 70 64 61 74 65 1a  02 76 31 22 08 08 db c7  |.Update..v1"....|
000000b0  d5 c2 06 10 00 32 08 46  69 65 6c 64 73 56 31 3a  |.....2.FieldsV1:|
000000c0  30 0a 2e 7b 22 66 3a 64  61 74 61 22 3a 7b 22 2e  |0..{"f:data":{".|
000000d0  22 3a 7b 7d 2c 22 66 3a  64 65 6d 6f 4b 65 79 22  |":{},"f:demoKey"|
000000e0  3a 7b 7d 7d 2c 22 66 3a  74 79 70 65 22 3a 7b 7d  |:{}},"f:type":{}|
000000f0  7d 42 00 12 18 0a 07 64  65 6d 6f 4b 65 79 12 0d  |}B.....demoKey..|
00000100  4d 79 53 33 63 72 33 74  56 34 6c 75 65 1a 06 4f  |MyS3cr3tV4lue..O|
00000110  70 61 71 75 65 1a 00 22  00 0a                    |paque..".."|
0000011a
```

Figura 4.12: Consulta ao `etcd` antes da encriptação de Secrets

Transitou-se então para a fase seguinte do teste: a ativação do sistema KMSv2, e subsequente consulta ao Secret. Para esta operação realizaram-se os passos descritos no final da subsecção 4.2.1 - a configuração do API Server, e subsequente reinicio do serviço.

Realizando este procedimento, o Secret criado não é encriptado automaticamente. Tal deve-se ao facto de a encriptação automática de todos os Secrets aquando da ativação deste mecanismo poder corromper as operações habituais do ambiente Kubernetes. Como tal, é necessário reprocessar os segredos desejados para que estes sejam encriptados [4]. O comando utilizado para o reprocessamento é apresentado no excerto 4.30.

```
$ kubectl get secret secret-test -n default -o json | kubectl
  replace -f -
```

Excerto de código 4.30: Reprocessamento de Secret para utilização de encriptação

Por fim, resta executar novamente a query ao `etcd` e conferir o resultado, representado na figura 4.13.

[4] Este processo é documentado pelos próprios desenvolvedores do Kubernetes: `https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/#ensure-all-secrets-are-encrypted`

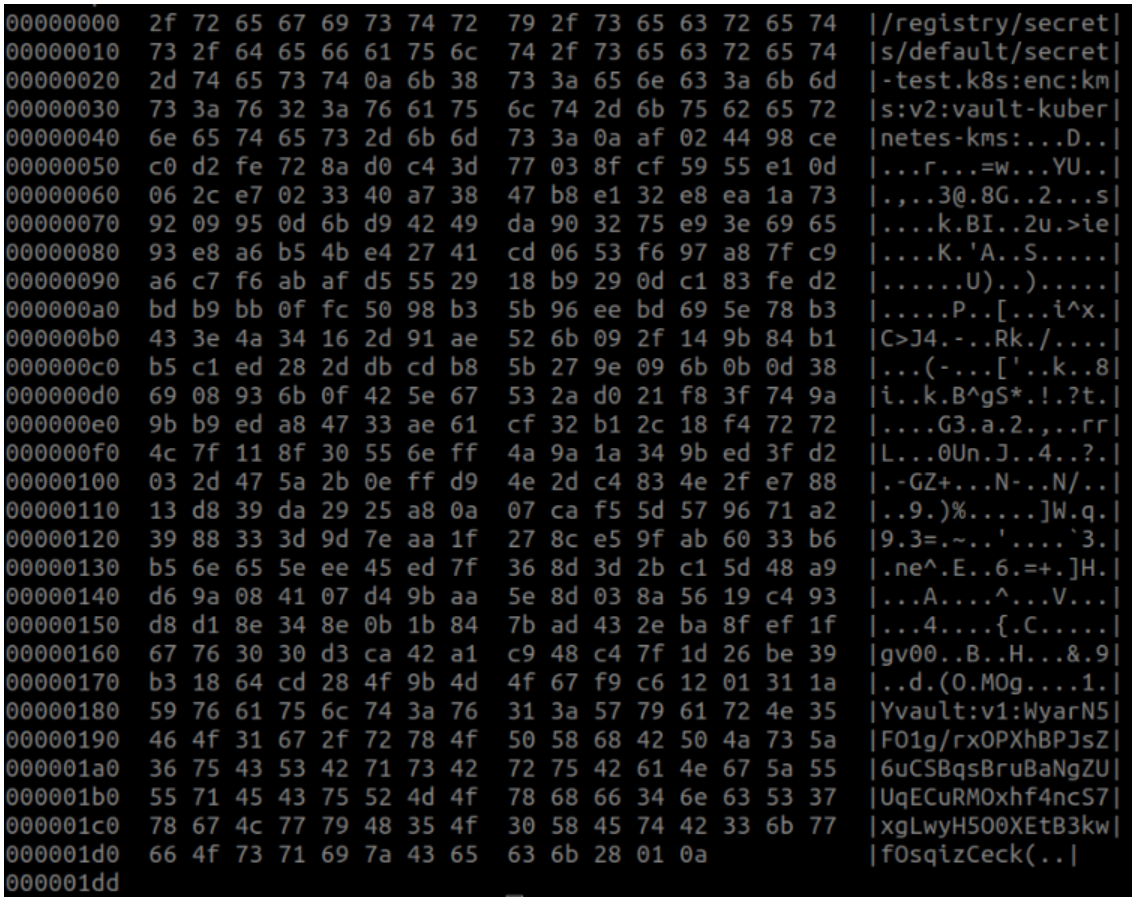

Figura 4.13: Consulta ao `etcd` após encriptação de Secrets

Note-se que o resultado da *query* é diferente, indicando que foram efetuadas alterações. No resultado obtido, deixa de ser possível identificar o valor em texto claro do Secret configurado, evidenciando-se apenas uma sequência de caracteres indiscernível, prefaciada com o indicador `"vault:v1"`, comprovando a utilização deste método. Vale realçar que este `"v1"`não se refere à versão do KMS, mas sim à do *plugin* utilizado para a interação entre o API Server e o Hashicorp Vault.

Desse modo, considera-se bem sucedido o teste preconizado, visto ter deixado de ser possível interpretar um Secret através de consultas ao *etcd*, obtendo o valor em texto claro. Assim, passar-se-á ao teste dos componentes de deteção de incidentes envolvidos na solução final, começando-se pelo `kube-score`.

4.3.2 Testes ao `kube-score`

Relembrando alguns pontos chave relativamente a esta ferramenta, a sua função é a deteção de configurações incorretas e inseguras em manifestos de objetos interpretados pelo Kubernetes, tais como `Pods`, `StatefulSets`, `Deployments`, entre outros. Como tal, para a avaliação deste componente, foram definidos alguns manifestos expressamente inseguros, subsequentemente aplicados no *cluster*. O objetivo destes testes é avaliar se o `CronJob` implementado para a execução do `kube-score` deteta estas anomalias, reportando-as por via de *logs* que serão subsequentemente observados no Grafana.

O processo de teste foi o seguinte: a criação de um manifesto inválido, de entre os vários tipos de objeto definidos anteriormente; a definição de deteções esperadas; e a observação dos resultados do *scan* no Grafana, verificando quantos dos esperados foram detetados. Testaram-se três tipos de objeto, sendo estes os mais relevantes para o contexto atual: `Pod`, `Service`, e `Deployment`.

Deteção de configurações incorretas em `Pods`

Para o presente teste, foi definido o manifesto de um `Pod` apresentado em 4.31.

```
apiVersion: v1
kind: Pod
metadata:
  name: flawed-pod
  namespace: default
spec:
  containers:
    - name: app-container
      image: nginx:latest
      ports:
        - containerPort: 80
      securityContext:
        privileged: true
      resources:
      livenessProbe:
        httpGet:
          path: /
          port: 80
```

Excerto de código 4.31: Pod inválido para teste ao `kube-score`

A definição apresentada foi realizada com o intuito de gerar algumas deteções, sendo estas:

- Uso da *tag latest*: esta utilização é pouco recomendada, podendo constituir uma falha de segurança em alguns casos;
- Aviso sobre *container* com `privileged` ativo: a utilização de *containers* com privilégios de sistema elevados deve ser evitada o máximo possível;
- Aviso sobre ausência de *Requests* e *Limits*: o Pod especificado não define a sua utilização de recursos esperada através destes dois campos, algo que pode ser prejudicial para o *cluster*, podendo alocar demasiados recursos a este Pod que poderiam ser utilizados para outros fins;
- Aviso sobre ausência de Security Context: o Pod deveria especificar a chave de `securityContext` - ao nível do Pod, e não apenas ao nível do *container* - para configurar certas propriedades de segurança, como por exemplo o utilizador para execução (distinto do `root`).

Definidos estes objetivos, foi aplicado o manifesto apresentado no *cluster*, aguardando-se pela execução automática do `kube-score`. Aquando desta, os resultados foram visualizados no Grafana, e apresentam-se na figura 4.14. Note-se que os resultados foram filtrados para incluir apenas as linhas relativas a deteções relevantes - ou seja, as validações que forem validadas como estando em conformidade são excluídas destes resultados. Tal decisão foi feita para obter uma melhor interpretação dos resultados relevantes.

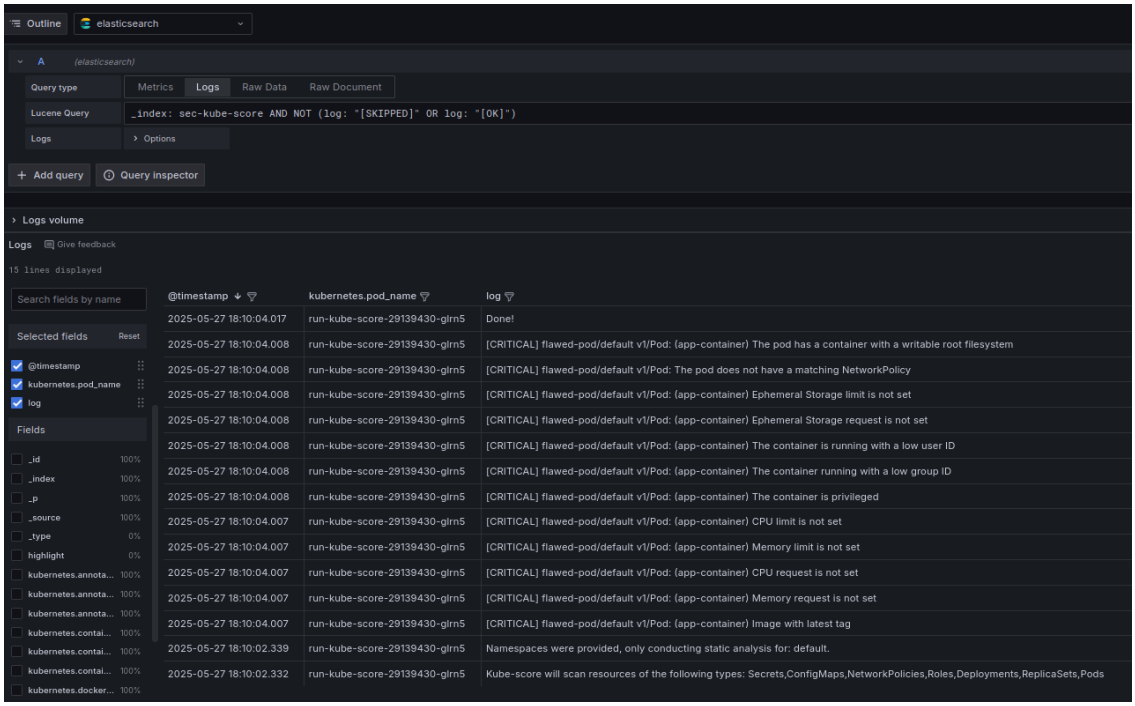

Figura 4.14: Deteções de Pod inválido realizadas pelo `kube-score`

É possível confirmar que o `kube-score` detetou 3 das 4 condições definidas para o atual teste. Gerou avisos relativamente a: utilização da *tag latest*; ausência de *Requests* e *Limits* de CPU, RAM e armazenamento, sendo estes os três tipos de recurso válidos para estas definições; execução de um *container* no modo *privileged*. Note-se que não gerou nenhum evento específico relativamente à ausência do `securityContext` ao nível do Pod, sendo esta a única condição que não foi atingida.

Contudo, foram gerados alguns *logs* cuja deteção não foi esperada, mas é ainda assim valiosa. O `kube-score` reportou a ausência de uma `NetworkPolicy` para o Pod, bem como o facto de este ter um sistema de ficheiros `root` com permissões de escrita. O primeiro evento, num cenário de produção, pode suscitar a algum ruído, visto ser uma prática comum a adoção de políticas de rede para segregação por *namespace*, e não especificamente por Pod. Não obstante, a filtragem por Pod é a prática de segurança mais aconselhada, tornando a deteção relevante. No caso do sistema de ficheiros, este facto advém da execução do *container* no modo *privileged*, mas é uma deteção relevante por si só, pois realça uma preocupação de segurança distinta.

Deteção de configurações incorretas em `Services`

Para testar a deteção de configurações inapropriadas em objetos do tipo `Service`, foi definido o manifesto de apresentado em 4.32.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: flawed-service
  namespace: default
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080
      protocol: TCP
  selector:
    app: non-existent-app
```

Excerto de código 4.32: Service inválido para teste ao `kube-score`

Para este caso, esperam-se as seguintes deteções:

- Aviso sobre utilização do `NodePort`: tipicamente, este tipo de serviço não deve ser utilizado, visto abrir um porto nos próprios *hosts* responsáveis pela execução das cargas de trabalho;
- Aviso sobre `selector` incorreto: a condição definida para o `selector` neste Service não corresponde a nenhum Pod em execução no *cluster*, fazendo com que a configuração seja incorreta.

Seguindo o mesmo processo do teste anterior, foram geradas as deteções apresentadas na figura 4.15. Note-se que os resultados foram filtrados apenas para incluir deteções relativas a `Services`, novamente, para uma melhor interpretação dos resultados obtidos.

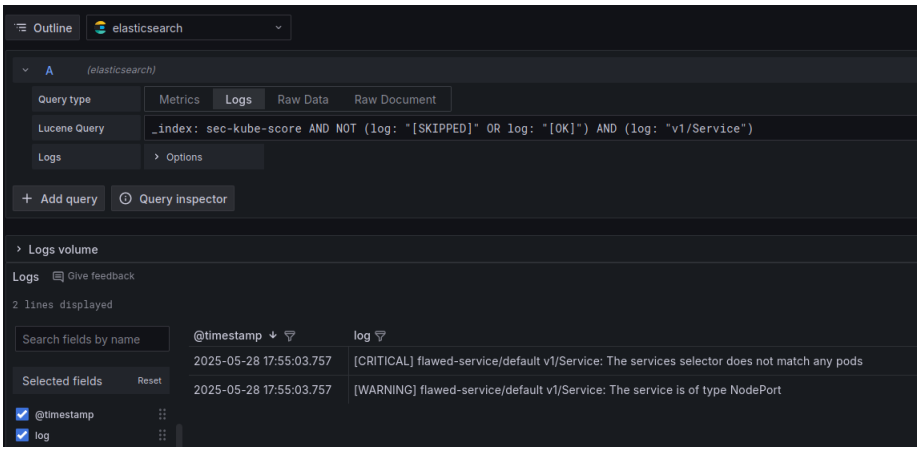

Figura 4.15: Deteções de Service inválido realizadas pelo `kube-score`

As deteções foram exatamente as esperadas, confirmando o funcionamento esperado da ferramenta para o caso dos `Services`. Adicionalmente, foram também realizadas novas deteções ao Pod definido para o teste anterior, nomeadamente a ausência de um `readinessProbe`, como apresentado na figura 4.16. Tal se sucede pois passou a existir um `Service` no mesmo *namespace* - ainda que este esteja incorreto. Esta deteção é benéfica para a solução, pois se trata de um ponto de configuração relevante que deve ser

endereçado. A sua deteção após a criação de um `Service` é lógica, pois este objeto é responsável por providenciar acesso externo ao Pod, pelo que é crucial garantir que este está a pronto a receber pedidos - algo que o `readinessProbe` permite realizar.

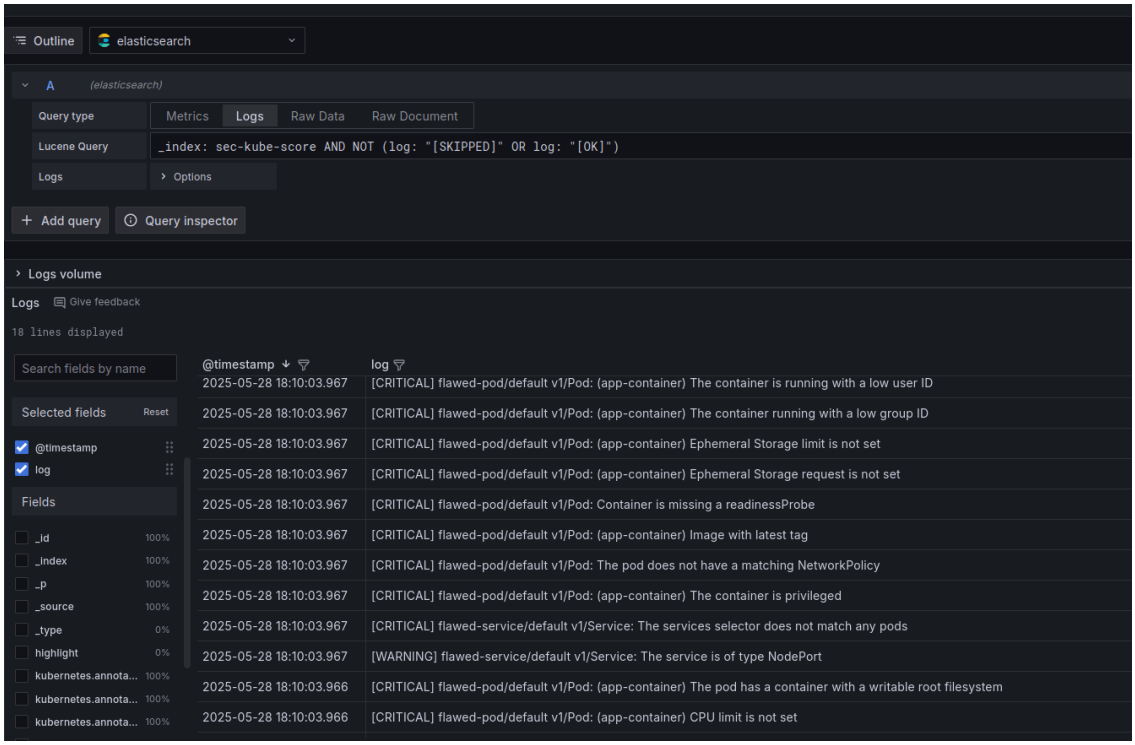

Figura 4.16: Deteções de Pod inválido após criação do Service realizadas pelo `kube-score`

Deteção de configurações incorretas em `Deployments`

Finalmente, para aferir a eficácia da deteção de configurações incorretas em `Deployments`, foi definido o manifesto de apresentado em 4.33.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flawed-deployment
  labels:
    app: flawed-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flawed-app
  template:
    metadata:
      labels:
        app: flawed-app
    spec:
      containers:
        - name: nginx
          image: nginx:latest
```

```
20            ports:
21              - containerPort: 80
22            securityContext:
23              runAsUser: 0
```

Excerto de código 4.33: Deployment inválido para teste ao `kube-score`

Com este teste, esperam-se que sejam detetados os seguintes problemas de configuração:

- Imagem de *Container* executado pelo `Deployment` especificada com a tag *latest*;
- Ausência de *Requests* e *Limits* para utilização de recursos;
- Ausência de `readinessProbe`: contrariamente ao caso em que apenas se criou um Pod, esta deteção é esperada quando se trata de um Deployment, pois esta chave é crucial para definir o bem-estar da aplicação gerida por este recurso;
- Execução com utilizador `root`: conforme pode ser visualizado na linha 26, o *container* será executado com o utilizador cujo ID é 0, sendo este o `root` nos sistemas Linux, pelo que esta definição deve gerar um alerta.

De seguida, foi então aplicado o manifesto no *cluster*, aguardando-se pela execução do `kube-score`. Aquando desta, foram geradas as deteções apresentadas na figura 4.17.

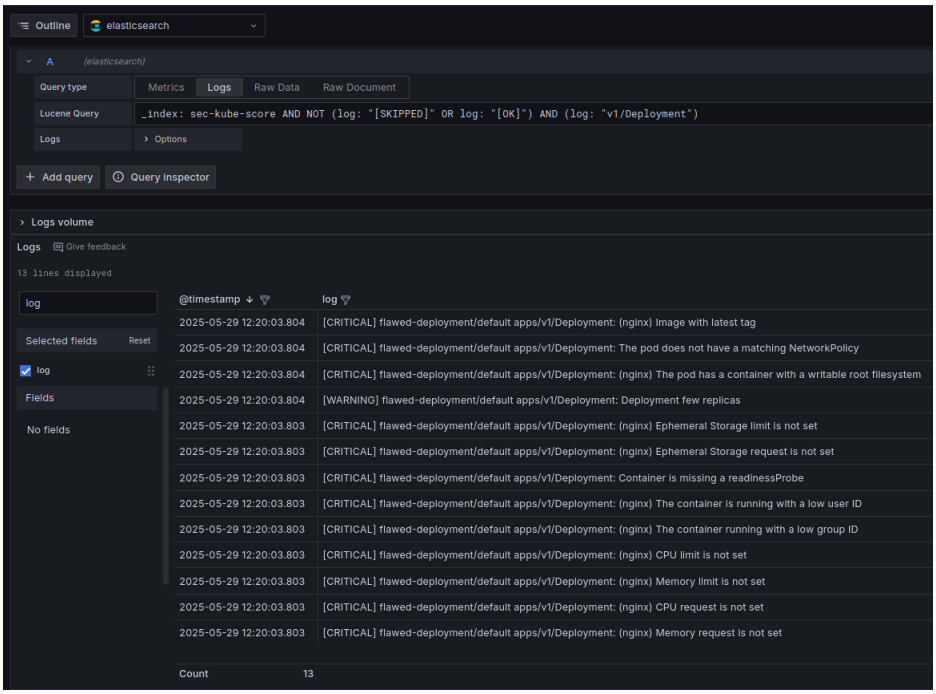

Figura 4.17: Deteções de Deployment inválido após criação do Service realizadas pelo `kube-score`

Conforme é possível comprovar pelos resultados, todas as condições esperadas foram reportadas pelo `kube-score`, acrescentando-se também algumas deteções adicionais - previamente referidas aquando da análise na deteção de erros em Pods. Por fim, a ferramenta gerou um aviso de menor nível, porém importante - o facto de o `Deployment` assumir poucas réplicas. Em casos onde a alta disponibilidade aplicacional é importante, esta deteção é relevante, alertando os administradores para que possam aumentar o número de réplicas.

4.3.3 Testes ao Tetragon

No desenvolvimento do Tetragon, foram implementadas duas políticas de exemplo - uma que monitoriza acessos a ficheiros sensíveis, outra que rastreia conexões a redes externas à do *cluster*. Desse modo, essas devem ser testadas, avaliando-se a sua eficácia.

Para cada teste, adotou-se o seguinte processo: a definição da ação a executar, de acordo com a política a testar; a delineação dos resultados esperados; a execução da ação; e a subsequente avaliação dos resultados. De modo análogo ao que se sucedeu com o `kube-score`, a visualização dos resultados é feita centralmente no Grafana, avaliando-se não só a eficácia da ferramenta, como o funcionamento de todo o percurso dos dados.

Serão, então, apresentados os dois testes realizados, bem como uma discussão dos seus resultados.

Teste da política `file-monitoring`

A política de `file-monitoring` tem como objetivo a deteção de acessos e modificações a ficheiros ou diretórios definidos como sensíveis. Na implementação atual, estes foram configurados como sendo o `/etc` e o `/var/log`. No primeiro caso, pretende-se detetar acessos indevidos a ficheiros de configuração, enquanto que no segundo se pretendem reportar alterações a *logs*, numa perspetiva de assegurar a sua integridade - algo definido como importante para a organização por requisitos de *compliance*, entre outros fatores.

Devido à existência de bastantes falsos-positivos (tema que será amplamente explorado no decorrer da análise de solução), os testes efetuados incidiram essencialmente no diretório `/etc`. Foram realizados dois testes: um ao nível do Kubernetes; outro ao nível do *node*. No primeiro caso, foi lançado um Pod de testes, e na *shell* do mesmo acedeu-se ao ficheiro `/etc/passwd`, tendo como objetivo a geração de um *log* que reporte este incidente. Já no segundo caso, a ação foi a mesma, porém em vez de ser realizada num Pod, foi sim executada no próprio nó computacional - ou seja, estabelecendo-se uma conexão SSH ao *worker node* e acedendo ao referido ficheiro.

Começando pelos resultados do acesso a um ficheiro crítico dentro de um Pod, estes são apresentados na figura 4.18.

É possível confirmar que os eventos desejados foram detetados, apresentando-se um nível de detalhe significante sobre os mesmos. O Tetragon consegue coletar informação sobre o Pod que executou a ação, bem como o *namespace* em que este se insere, o comando utilizado para o acesso (neste caso, o `cat`), os argumentos do mesmo, entre outros atributos. Note-se, no entanto, a quantidade de eventos gerados. Para além daqueles relativos ao acesso, são reportados muitos outros - de algum modo despoletados pelo processo de testes, mas não relevantes para este. Podem considerar-se, então, como sendo falsos-positivos.

Relativamente ao teste realizado ao nível do *host*, o resultado do mesmo é apresentado na figura 4.19.

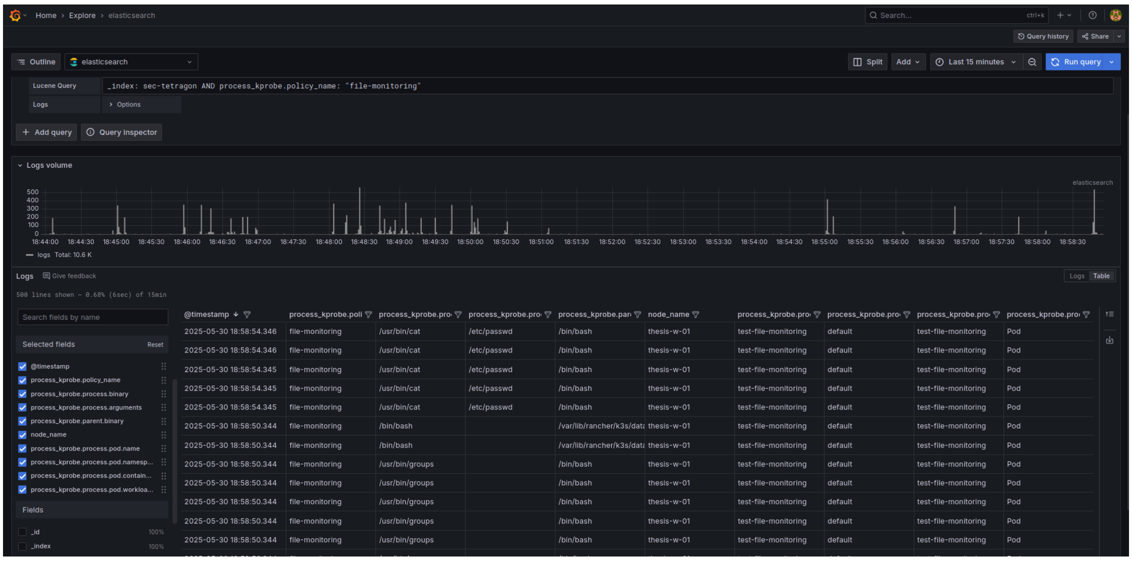

Figura 4.18: Deteções de acesso a ficheiros num Pod pelo Tetragon

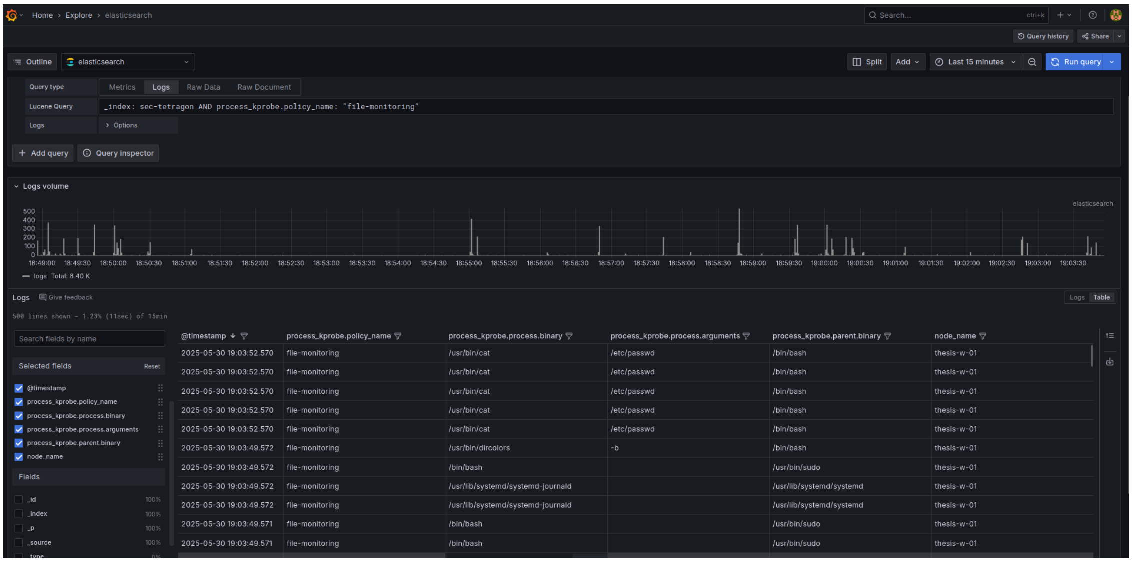

Figura 4.19: Deteções de acesso a ficheiros no *host* pelo Tetragon

Os resultados, bem como as conclusões extraídas dos mesmos, são semelhantes às descritas no primeiro teste. O evento desejado é detetado com sucesso, oferecendo-se bastante informação sobre o mesmo. Contudo, diversos falsos-positivos são apresentados. Algo que no processo de teste foi possível verificar, e considera-se como sendo um positivo, é a possibilidade da distinção entre eventos de *host* e de Pod. Conforme apresentado na figura 4.20, os eventos que se sucedem no próprio nó não preenchem os campos de Pod e *namespace* - algo totalmente lógico, e que pode ser utilizado em *dashboards* para filtragem e exibição de resultados ao utilizador.

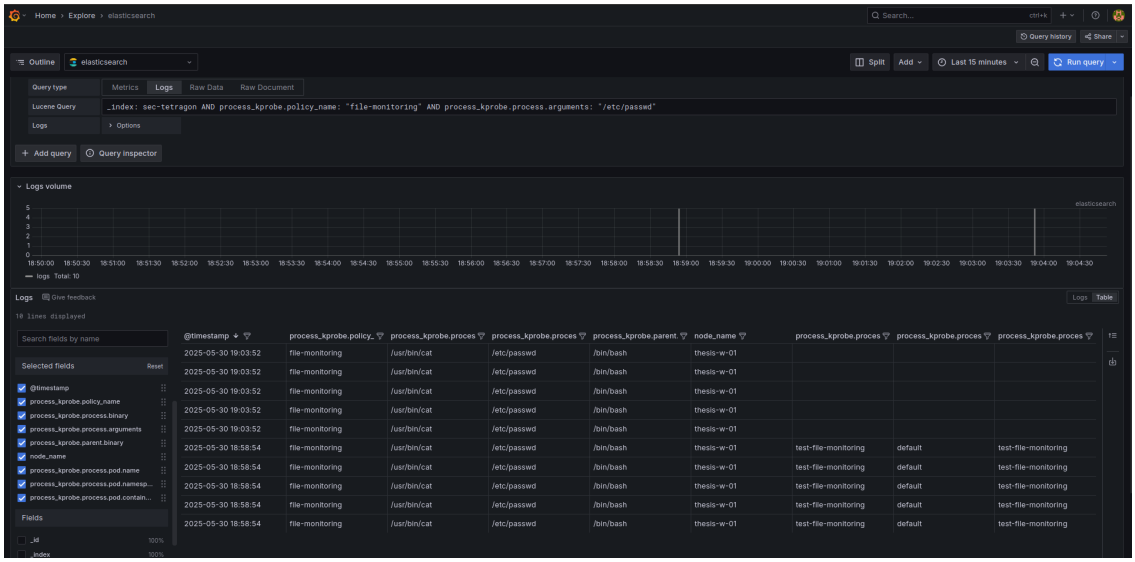

Figura 4.20: Deteções de acesso a ficheiros pelo Tetragon (filtrado para comparação entre *host* e Pod)

A conclusão que se extrai destes testes é a seguinte: o Tetragon deteta os eventos desejados com precisão, granularidade e detalhe. Contudo, esses mesmos fatores levam a que existam muitos falsos-positivos, levando a uma menor eficácia enquanto parte da solução. Se um utilizador souber o que procurar - isto é, por exemplo, saber que pretende encontrar eventos que tenham acontecido ao ficheiro `/etc/passwd`, então os resultados terão a exatidão desejada. No entanto, num cenário de observação rotineira dos eventos, os que são de facto críticos podem ser ocultados pelos falsos-positivos, levando a uma menor eficácia da ferramenta.

Teste da política `connect`

Para o teste da política que monitoriza conexões a redes externas ao *cluster*, o processo foi simples. Foi executado um Pod com uma imagem base - no presente caso, utilizou-se o Ubuntu. Nesse pod, foi instalado o binário `curl`, subsequentemente utilizado para aceder a um endereço externo ao *cluster*. Os resultados deste teste são apresentados na figura 4.21.

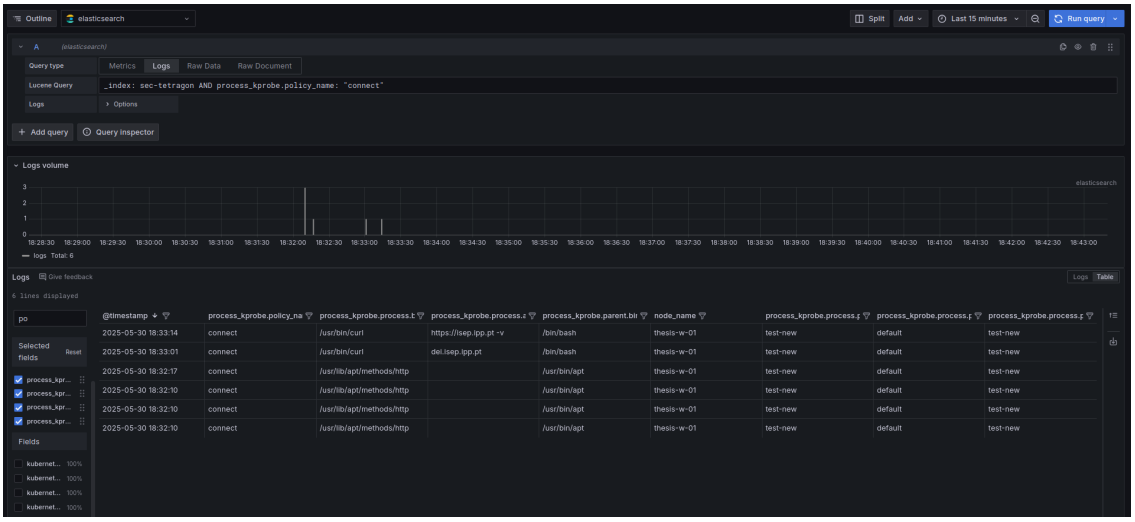

Figura 4.21: Deteções de acesso a endereços de rede externos pelo Tetragon

É possível validar que o Tetragon foi capaz de detetar este acesso de rede, conforme previsto, cumprindo assim o seu propósito definido. Contudo, e de modo semelhante à política anterior, assume também alguns falsos-positivos. Por exemplo, para a instalação do `curl`, foi utilizado o `apt`, que naturalmente tem que aceder a registos de pacotes - sendo estes externos ao *cluster*. Ora, casos como este são bastante frequentes mesmo numa infraestrutura de produção: contacto a serviços externos, a bases de dados, a outros *clusters*, por exemplo. Como tal, eventos potencialmente críticos podem ficar ocultos entre os falsos-positivos, tal como o sucedido na política anterior.

4.3.4 Testes ao Trivy e `kube-bench`

Na solução desenvolvida, o Trivy assume a importante função de conduzir a análise de vulnerabilidades nos artefactos - notavelmente, imagens de *container* - executados no *cluster*. Esta ferramenta deve conduzir análises de vulnerabilidades aos diversos componentes do *cluster* e reportar sobre os mesmos. Adicionalmente, e conforme discorrido em secções anteriores, o Trivy internamente usa a ferramenta `kube-bench` para análise do ambiente Kubernetes face a normas e boas práticas definidas pelo CIS.

Foram executados dois testes ao Trivy: um para aferir as capacidades de deteção de vulnerabilidades; outro para destacar os resultados de *benchmark* do *cluster* ao nível da conformidade com as referidas práticas.

Teste da análise de vulnerabilidades

Para testar a análise de vulnerabilidades, foi definido um processo de testes, bem como um conjunto de resultados esperados. No que toca ao processo, foi definido como: criação de um Pod no *cluster* com uma imagem vulnerável, e visualização central no Grafana das vulnerabiliades reportadas pelo Trivy.

Para atingir a fase inicial deste teste, foi criada uma imagem vulnerável - armazenada no Docker Hub como `lew6s/vulnerable-image-test:1.0` - que nada mais é do que uma extensão da imagem `docker-goof`, disponibilizada pela Snyk Labs [5].

[5] Obtida a partir de: `https://github.com/snyk-labs/docker-goof`

Tendo sido criada a imagem, foi então definido o manifesto do Pod a instalar no *cluster*, apresentando-se em 4.34. Note-se a utilização da imagem criada no campo `image` da definição do Pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: flawed-pod-image
  namespace: default
spec:
  containers:
    - name: app-container
      image: lew6s/vulnerable-image-test:1.0
      ports:
        - containerPort: 80
      securityContext:
        privileged: true
      resources:
      livenessProbe:
        httpGet:
          path: /
          port: 80
```

Excerto de código 4.34: Pod definido para o teste de deteção de vulnerabilidades

O Pod apresentado foi, então, criado no *cluster*. De seguida, o Trivy automaticamente executou uma análise das vulnerabilidades da imagem utilizada por este novo recurso, reportando-as através de CRDs - conforme explorado no decorrer da implementação. O objetivo do teste era não só validar esta geração, mas também a sua visualização através de um ponto central. Desse modo, uma *query* a métricas do Trivy foi efetuada no Grafana, obtendo-se o número de vulnerabilidades reportadas para esta, conforme apresentado na figura 4.22.

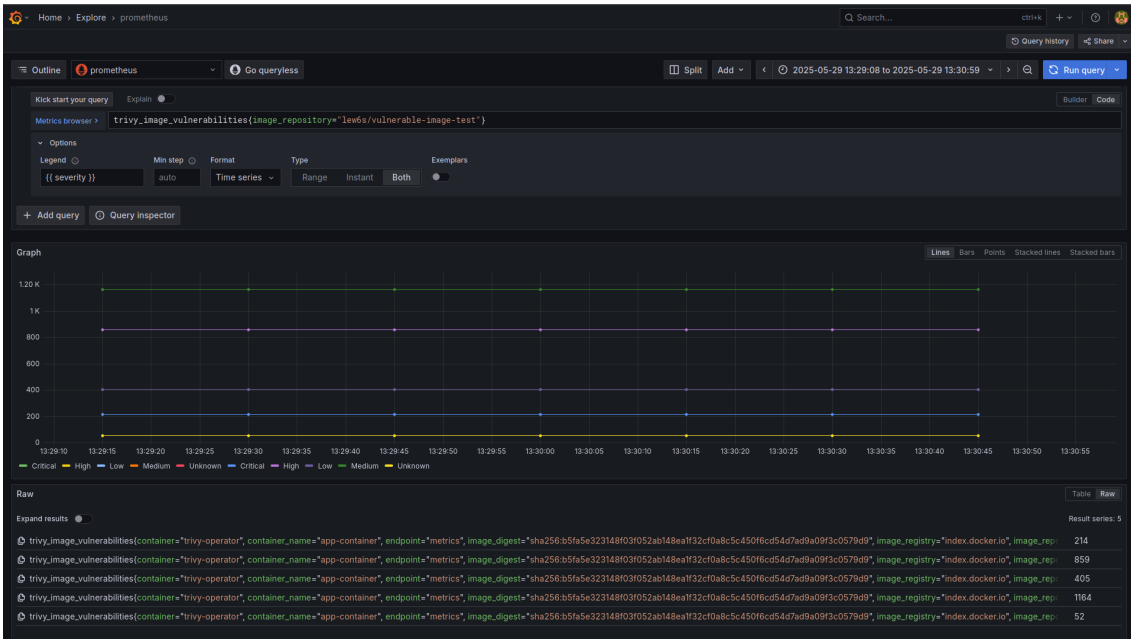

Figura 4.22: Deteção de vulnerabilidades pelo Trivy

Confirma-se o sucesso do teste, dado que o Trivy não só reportou a existência de vulnerabilidades na imagem, como também as categorizou de acordo com a severidade. Note-se, contudo, que as métricas utilizadas estão configuradas de modo sumário, conforme expresso no decorrer da implementação, devido ao aumento da cardinalidade das mesmas caso se adotasse um método de relatório mais extensivo.

Teste da análise do cluster face ao CIS benchmark

O âmbito do atual teste foi avaliar a geração de um relatório de conformidade do *cluster* Kubernetes implementado face ao conjunto de normas definidas pelo CIS. Dada a implementação realizada, o processo de teste foi simplesmente a verificação das métricas relativas a *compliance* através do Grafana, analisando os resultados - que se apresentam na figura 4.23.

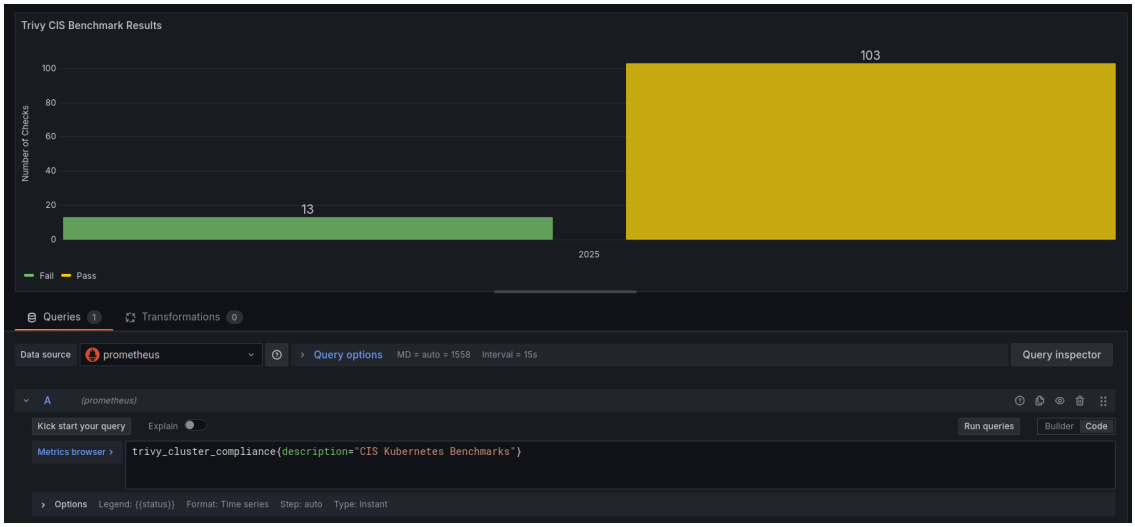

Figura 4.23: Relatório de *compliance* gerado pelo Trivy

É possível validar que os resultados são apresentados conforme esperado. Contudo, a sua forma sumária não é particularmente benéfica para um administrador - apenas informa que 13 testes falharam, mas não discorre sobre eles, de modo a auxiliar na correção do problema que gera estas falhas. A perspetiva sumária é, em todo o caso, útil em diversos cenários: para obter uma visão geral da postura do *cluster* face às boas práticas, para apresentação em auditorias de *compliance*.

No entanto, manifestou-se a necessidade de obter resultados mais granulares destes relatórios. Como tal, o *values* do Trivy foi alterado para definir `metricsClusterComplianceInfo: true`. Ao alterar esta definição, é possível visualizar métricas mais granulares, obtendo-se informação sobre quantas falhas existiram em cada teste, bem como uma descrição do teste falhado. Na figura 4.24, é possível observar a categorização dos testes falhados por severidade, enquanto que na figura 4.25 existe a descrição detalhada de cada teste falhado.

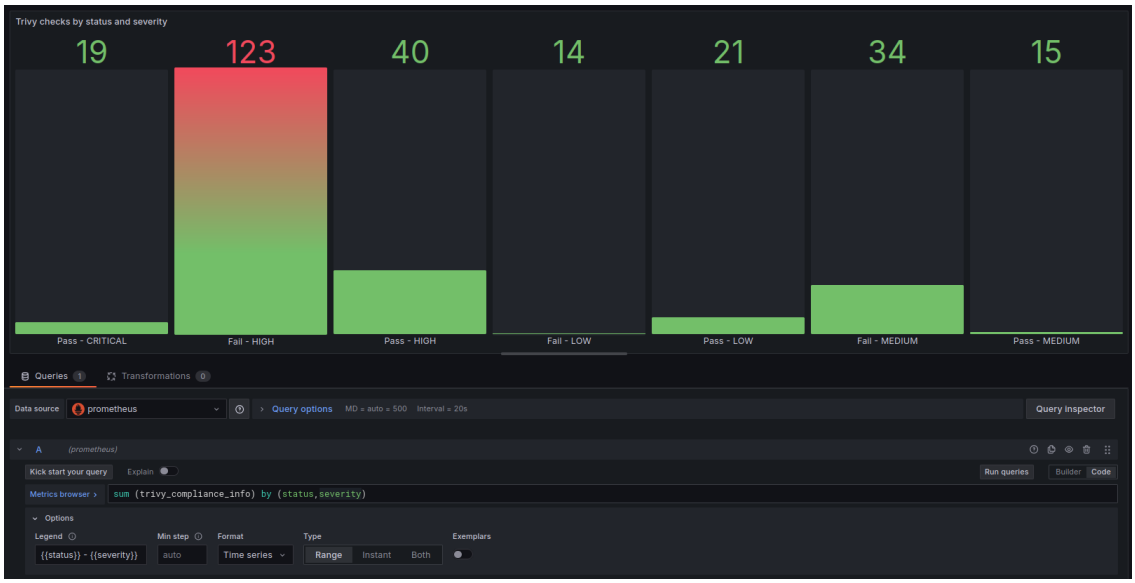

Figura 4.24: Relatório de *compliance* gerado pelo Trivy - falhas por severidade

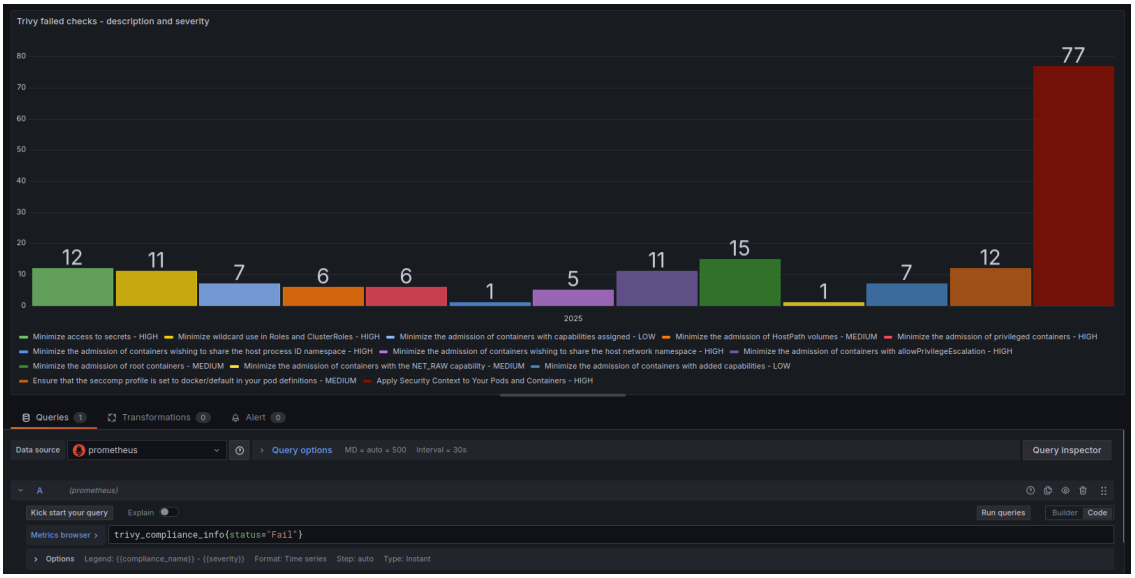

Figura 4.25: Relatório de *compliance* gerado pelo Trivy - falhas com teste detalhado

Assim, conclui-se que os resultados obtidos vão de encontro ao esperado, podendo-se também obter detalhes mais granulares relativamente a estes testes.

Teste do impacto do Trivy no Prometheus

Face à alteração da propriedade do Trivy que aumenta a granularidade dos resultados obtidos - e, por consequência, a cardinalidade das suas métricas - realçou-se a importância de testar esta alteração numa perspetiva de impacto na *performance*.

Conforme apresentado, o Prometheus é a ferramenta responsável pela recolha de métricas. Logo, o aumento do seu número, bem como da cardinalidade, terá imperativamente impacto no Prometheus, mais concretamente no que toca ao uso de memória RAM. Desse modo, definiu-se o seguinte teste: ativar todas as métricas possíveis no Trivy, e manter essa configuração durante pelo menos duas horas; de seguida, utilizar métricas disponibilizadas pelo Prometheus para avaliar a evolução do consumo de memória. O espaço temporal de duas horas foi definido pois este é o intervalo temporal em que o Prometheus realiza o *flush* dos seus dados da RAM para o disco (Prometheus Authors 2025). Passado um período de tempo conforme o especificado, a referida métrica foi consultada, apresentando-se o resultado da mesma na figura 4.26.

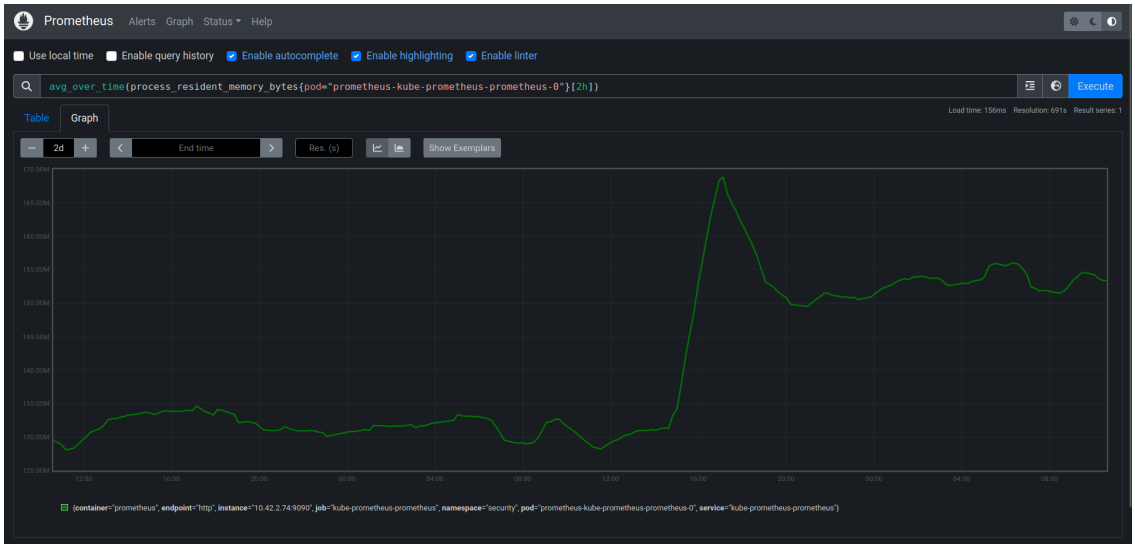

Figura 4.26: Impacto das métricas adicionais do Trivy no consumo de RAM do Prometheus

É notável um aumento no consumo de memória: passando de um valor médio de, sensivelmente, 133Mb; para 153Mb, destacando então um aumento de 15.04%. Neste caso, não se trata de uma mudança drástica, porém há que transpor este acontecimento para um cenário de produção. Num ambiente produtivo, existem muito mais Pods, que executam um ou mais *containers* de diversas imagens. Logo, existindo mais imagens, também existirão mais relatórios distintos, reportando em vulnerabilidades diferentes. Tal cenário pode causar um aumento drástico do consumo de memória por parte do Prometheus levando, em último caso, à disrupção do bom funcionamento da plataforma.

4.4 Resumo

No decorrer do presente capítulo, foi apresentado o processo de implementação da prova de conceito para a solução preconizada. Inicialmente, e tratando-se de uma definição base do *cluster*, a encriptação de segredos recorrendo ao mecanismo KMSv2 foi configurada, usando o Hashicorp Vault externo como mecanismo de geração e armazenamento de chaves utilizadas pelo sistema.

Foi implementado o `kube-score` para análise de manifestos em Kubernetes, recorrendo-se à criação de um Helm Chart novo, executando um *script* desenvolvido para executar periodicamente este utilitário. Foi implementado o Tetragon, ajustando-se a sua configuração aos requisitos definidos e limitando a captura de eventos, bem como um novo Chart para criação de políticas no *cluster*, com base em argumentos providenciados na configuração. Terminando a implementação de ferramentas para análise do *cluster*, o Trivy foi configurado para deteção de vulnerabilidades e de incumprimento com boas práticas definidas pelo CIS *benchmark*.

Para a recolha dos dados emitidos por estas ferramentas, o Prometheus e *fluent-bit* foram implementados para a recolha de métricas e *logs*, respetivamente. No segundo caso, o Elasticsearch foi configurado, servindo como ponto de armazenamento para os *logs* recolhidos e remetidos a este pelo `fluent-bit`. Por fim, o Grafana foi instalado, servindo como ponto

central de visualização dos dados, e neste o Prometheus e Elasticsearch foram definidos como *datasources*, para que os seus dados possam ser consultados.

Com uma instância da prova de conceito em funcionamento, foram executados diversos testes de modo a aferir a eficácia de deteção de ameaças das ferramentas implementadas. A encriptação de segredos foi testada, garantindo que a sua leitura em texto claro deixa de ser possível no `etcd`. Cada uma das ferramentas de deteção referidas - `kube-score`, Tetragon e Trivy - foi testada, obtendo-se resultados geralmente positivos. No caso do `kube-score`, a maioria das deteções esperadas foram verificadas, bem como algumas que não foram inicialmente previstas, mas mostraram-se relevantes. Já no caso do Tetragon, este detetou todos os eventos definidos para teste, porém com muitos falsos-positivos, colocando em questão a verdadeira eficácia das políticas implementadas. Finalmente, o Trivy mostrou-se eficaz na deteção de vulnerabilidades, bem como na execução de testes de *compliance*, sendo possível consultar os seus resultados através das métricas por ele expostas.

Assim, conclui-se o processo de implementação da solução e análise dos seus resultados, transitando-se então para a conclusão do presente relatório, verificando o cumprimento dos objetivos propostos, bem como limitações da solução existente, e descrição do trabalho futuro.

Capítulo 5

Conclusão e Trabalho Futuro

No decorrer do atual capítulo serão apresentadas as conclusões do projeto de dissertação desenvolvido. Primeiramente, serão sumariadas as diversas secções do projeto, bem como os resultados das mesmas. De seguida, serão descritas as contribuições da solução desenvolvida, enquadrando-as no âmbito dos objetivos definidos. Finalmente, serão descritas algumas limitações da prova de conceito implementada, bem como a direção prevista do trabalho futuro no desenvolvimento da plataforma.

5.1 Sumário e Contribuições

O presente trabalho aborda um problema identificado no âmbito da segurança em ambientes Kubernetes - a inexistência de uma solução integrada de modo a reforçar a sua postura de segurança. Mais concretamente no que toca a proteção de dados, gestão de identidade e resposta a incidentes. Este problema é ainda mais prevalente numa infraestrutura *bare-metal*, que é o caso no atual projeto.

Através de uma análise sistemática de literatura, foi possível identificar que práticas e ferramentas existem para reforçar a segurança de cada um destes componentes, refletindo-se também em modos de realizar a sua integração. A implementação de soluções como encriptação do etcd, políticas de RBAC, e ferramentas para análise de vulnerabilidades, controlo de admissão e *logging* demonstrou potencial para elevar a segurança nestes ambientes. Adicionalmente, e de acordo com a visão do projeto, a integração dessas práticas e tecnologias em uma única solução possibilita uma visão mais holística sobre a postura de segurança de um *cluster*. Por fim, a revisão do estado da arte realçou a importância de uma solução multi-camadas para a segurança em ambientes Kubernetes. A condução deste processo permitiu atingir o **Objetivo 2**, descrito na secção 1.2.

Com base no valioso conhecimento obtido pelo processo descrito, foi então preconizada a arquitetura da solução. Esta define que componentes constituem a mesma, justificando o seu papel e função, e detalhando o modo segundo o qual todas estas ferramentas se interligam - métricas e *logs*. Como tal, esta define a base para a concretização dos **Objetivos 2 a 5**.

Foi então conduzida a implementação da solução, gerando uma prova de conceito. Tal processo consistiu na investigação, instalação e configuração prática de alguns componentes definidos na arquitetura, bem como da metodologia de integração. Conforme expresso em 4, apenas alguns dos componentes foram implementados, tendo esta decisão sido justificada. Desse modo, os **Objetivos 2, 4 e 5** foram cumpridos.

A solução desenvolvida foi testada - foram conduzidos diversos testes aos componentes individuais da solução de deteção de intrusões, verificando-se os resultados dos mesmos de

modo centralizado, comprovando a viabilidade da integração das ferramentas adotadas, e cumprindo o **Objetivo 6**.

5.2 Limitações

Ainda que o resultado obtido tenha sido largamente positivo, existem certas limitações na prova de conceito produzida. Primeiramente, e conforme expresso anteriormente, não foram implementados todos os componentes, sendo notável a ausência do componente de RBAC, gestão de identidade e políticas preconizado na arquitetura. Adicionalmente, é de notar que ainda que o componente de encriptação tenha sido implementado e testado, este não se encontra integrado com a restante solução - isto é, não existem *logs* nem métricas deste processo que sejam capturados, registados, e integrados com os restantes eventos. Finalmente, vale realçar o escopo limitado das políticas de deteção em *runtime*. Tal se aplica quer para as desenvolvidas para o Tetragon, quer à ausência das *audit policies* apresentadas na arquitetura, sendo este um claro ponto de melhoria da solução.

5.3 Trabalho futuro

Dadas as limitações apresentadas anteriormente, é possível elencar vários pontos de trabalho futuro, transformando a prova de conceito desenvolvida numa solução completa e utilizável por administradores de ambientes Kubernetes.

A implementação dos componentes em falta deve ser o primeiro passo no trabalho subsequente. Deve, então, implementar-se um conjunto de políticas de RBAC, bem como um sistema de autenticação que contemple MFA - algo preconizado na arquitetura. A integração destes sistemas com a restante solução deve ser pensada numa perspetiva prática, avaliando-se a sua possibilidade. O OPA Gatekeeper deve, também ele, ser implementado e integrado com a plataforma, prestando-se especial atenção à sua relação com as políticas de autorização definidas. As *audit policies*, cujo objetivo é auditar ações no *cluster*, devem ser definidas, implementadas, e os seus *logs* devem ser capturados e armazenados centralmente - possibilitando a correlação de informação gerada por esta ferramenta com aquela emitida pela tecnologia de detenção em *runtime*.

Quanto a melhorias aos componentes implementados no decorrer do projeto, a deteção em tempo de execução deve ser abordada. Desse modo, devem ser desenvolvidas políticas de deteção relevantes para o Tetragon, ou até mesmo repensar a substituição desta ferramenta por outra - atualmente, o Falco seria uma boa alternativa.

Por fim, a configuração dos diversos componentes deve ser unificada, para que os administradores possam configurar toda a solução através de um ponto central, abordando a integração quer em tempo de execução, quer no que toca ao aprovisionamento da solução desenvolvida.

Bibliografia

ACM (2024). *ACM Code of Ethics and Professional Conduct*. url: https://www.acm.org/code-of-ethics.

Ali, Aamir et al. (2024). «Implementation of New Security Features in CMSWEB Kubernetes Cluster at CERN». en. Em: *EPJ Web of Conferences* 295. arXiv:2405.15342 [cs], p. 07026. issn: 2100-014X. doi: 10.1051/epjconf/202429507026. url: http://arxiv.org/abs/2405.15342 (acedido em 16/11/2024).

Aqua Security (2022). *Home - Trivy Operator*. url: https://aquasecurity.github.io/trivy-operator/latest/ (acedido em 17/12/2024).

– (dez. de 2024a). *aquasecurity/kube-bench*. original-date: 2017-06-19T13:27:02Z. url: https://github.com/aquasecurity/kube-bench (acedido em 16/12/2024).

– (dez. de 2024b). *aquasecurity/tracee*. original-date: 2019-09-18T13:20:17Z. url: https://github.com/aquasecurity/tracee (acedido em 18/12/2024).

– (dez. de 2024c). *aquasecurity/trivy*. original-date: 2019-04-11T01:01:07Z. url: https://github.com/aquasecurity/trivy (acedido em 17/12/2024).

– (2024d). *Working with Tracee's Policies on Kubernetes - Tracee*. url: https://aquasecurity.github.io/tracee/latest/tutorials/k8s-policies/ (acedido em 18/12/2024).

Budigiri, Gerald et al. (jun. de 2021). «Network Policies in Kubernetes: Performance Evaluation and Security Analysis». Em: *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. ISSN: 2575-4912, pp. 407–412. doi: 10.1109/EuCNC/6GSummit51104.2021.9482526. url: https://ieeexplore.ieee.org/document/9482526/?arnumber=9482526 (acedido em 16/11/2024).

Calvert, David (2023). *Prometheus' performance and cardinality in practice*. en. url: https://medium.com/@dotdc/prometheus-performance-and-cardinality-in-practice-74d5d9cd6230 (acedido em 18/12/2024).

Center for Internet Security (2024). *CIS Kubernetes Benchmarks*. en. url: https://www.cisecurity.org/benchmark/kubernetes (acedido em 16/12/2024).

Cilium (dez. de 2024a). *cilium/tetragon*. original-date: 2022-03-23T10:25:36Z. url: https://github.com/cilium/tetragon (acedido em 18/12/2024).

– (2024b). *Runtime Enforcement*. url: https://cilium.io/use-cases/runtime-enforcement/ (acedido em 18/12/2024).

– (2024c). *Tetragon Observability Policies*. en. url: https://tetragon.io/docs/policy-library/observability/ (acedido em 18/12/2024).

Cloud Native Computing Foundation (2023). *Cloud Native Computing Foundation Annual Survey 2023*. url: https://www.cncf.io/reports/cncf-annual-survey-2023/.

ControlPlane (dez. de 2024a). *controlplaneio/kubesec*. original-date: 2017-10-10T08:00:35Z. url: https://github.com/controlplaneio/kubesec (acedido em 17/12/2024).

– (2024b). *kubesec.io :: kubesec.io*. url: https://kubesec.io/ (acedido em 17/12/2024).

Creane, Brendan e Amit Gupta (2021). *Kubernetes security and observability: a holistic approach to securing containers and cloud native applications*. eng. First edition. Sebastopol, CA: O'Reilly Media, Inc. isbn: 978-1-09-810710-9.

Cruise (nov. de 2024). *cruise-automation/k-rail*. original-date: 2019-10-03T21:13:54Z. url: https://github.com/cruise-automation/k-rail (acedido em 18/12/2024).

Elastic (out. de 2020). *Benchmarking and sizing your Elasticsearch cluster for logs and metrics*. en-us. url: https://www.elastic.co/blog/benchmarking-and-sizing-your-elasticsearch-cluster-for-logs-and-metrics (acedido em 18/12/2024).

– (dez. de 2024). *elastic/elasticsearch*. original-date: 2010-02-08T13:20:56Z. url: https://github.com/elastic/elasticsearch (acedido em 18/12/2024).

Falco (dez. de 2024). *falcosecurity/falco*. original-date: 2016-01-19T21:58:12Z. url: https://github.com/falcosecurity/falco (acedido em 18/12/2024).

FalcoSuessgott e Tom Morelly (nov. de 2024). *vault-kms-plugin - vault-kms-plugin*. url: https://falcosuessgott.github.io/vault-kubernetes-kms/ (acedido em 19/06/2025).

Fluent (nov. de 2024a). *Fluent Bit v3.2 Documentation | Fluent Bit: Official Manual*. en. url: https://docs.fluentbit.io/manual (acedido em 18/12/2024).

– (dez. de 2024b). *fluent/fluent-bit*. original-date: 2015-01-27T20:41:52Z. url: https://github.com/fluent/fluent-bit (acedido em 16/12/2024).

– (nov. de 2024c). *Performance Tips | Fluent Bit: Official Manual*. en. url: https://docs.fluentbit.io/manual/administration/performance (acedido em 18/12/2024).

German, Kazenas e Olga Ponomareva (mai. de 2023). «An Overview of Container Security in a Kubernetes Cluster». Em: *2023 IEEE Ural-Siberian Conference on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)*. ISSN: 2769-3635, pp. 283–285. doi: 10.1109/USBEREIT58508.2023.10158865. url: https://ieeexplore.ieee.org/document/10158865/?arnumber=10158865 (acedido em 16/11/2024).

Gigasearch Engineering (ago. de 2021). *Benchmarking Performance: Elasticsearch vs Competitors*. en. url: https://medium.com/gigasearch/benchmarking-performance-elasticsearch-vs-competitors-d4778ef75639 (acedido em 18/12/2024).

Gkatziouras, Emmanouil et al. (2024). *Kubernetes secrets handbook: design, implement, and maintain production-grade Kubernetes Secrets management solutions*. en. Packt Publishing. isbn: 978-1-80512-322-4 978-1-80512-715-4.

Grafana (dez. de 2024a). *grafana/grafana*. original-date: 2013-12-11T15:59:56Z. url: https://github.com/grafana/grafana (acedido em 18/12/2024).

– (dez. de 2024b). *grafana/helm-charts*. original-date: 2020-08-20T14:55:44Z. url: https://github.com/grafana/helm-charts (acedido em 18/12/2024).

– (dez. de 2024c). *grafana/loki*. original-date: 2018-04-16T09:22:48Z. url: https://github.com/grafana/loki (acedido em 19/12/2024).

Hashicorp (dez. de 2024a). *hashicorp/vault*. original-date: 2015-02-25T00:15:59Z. url: https://github.com/hashicorp/vault (acedido em 18/12/2024).

– (2024b). *hashicorp/vault - Docker Image | Docker Hub*. en. url: https://hub.docker.com/r/hashicorp/vault (acedido em 18/12/2024).

– (2025). *Transit secrets engine | Vault | HashiCorp Developer*. en. url: https://developer.hashicorp.com/vault/docs/secrets/transit (acedido em 19/06/2025).

Helm (2024a). *Charts*. en. url: https://helm.sh/docs/topics/charts/ (acedido em 19/12/2024).

– (2024b). *Helm*. en. url: https://helm.sh/ (acedido em 19/12/2024).

– (2024c). *Helm Architecture*. en. url: https://helm.sh/docs/topics/architecture/ (acedido em 19/12/2024).

Huang, Kaizhe e Pranjal Jumde (2020). *Learn kubernetes security: securely orchestrate, scale, and manage your microservices in Kubernetes deployments*. en. Place of publication not identified: Packt Publishing. isbn: 978-1-83921-650-3 978-1-83921-218-5.

Instituto Politécnico do Porto (nov. de 2020). *Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto*. url: https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf.

Islam Shamim, Md. Shazibul, Farzana Ahamed Bhuiyan e Akond Rahman (set. de 2020). «XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices». Em: *2020 IEEE Secure Development (SecDev)*, pp. 58–64. doi: 10.1109/SecDev45635.2020.00025. url: https://ieeexplore.ieee.org/document/9230176/?arnumber=9230176 (acedido em 16/11/2024).

Kampa, Sandeep (out. de 2024). «Navigating the Landscape of Kubernetes Security Threats and Challenges». en. Em: *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)* 3.4. Number: 4, pp. 274–281. issn: 2959-6386. doi: 10.60087/jklst.v3.n4.p274. url: https://jklst.org/index.php/home/article/view/266 (acedido em 17/11/2024).

kost (dez. de 2024). *kost/dockscan*. original-date: 2015-11-13T04:04:45Z. url: https://github.com/kost/dockscan (acedido em 17/12/2024).

KubeArmor (set. de 2024a). *KubeArmor | KubeArmor*. en. url: https://docs.kubearmor.io/kubearmor (acedido em 18/12/2024).

– (dez. de 2024b). *kubearmor/KubeArmor*. original-date: 2020-11-26T01:59:16Z. url: https://github.com/kubearmor/KubeArmor (acedido em 18/12/2024).

Kyverno (dez. de 2024a). *kyverno/kyverno*. original-date: 2019-02-04T16:25:48Z. url: https://github.com/kyverno/kyverno (acedido em 18/12/2024).

– (2024b). *Performance Benchmarking*. en. url: https://docs.google.com/document/d/1z0_dryL1JH6oJ0-pIenzWw7edCzxj4ZD4go0PzzyMC8 (acedido em 18/12/2024).

– (2024c). *Policies*. en. url: https://kyverno.io/policies/ (acedido em 18/12/2024).

Martin, Andrew e Michael Hausenblas (2021). *Hacking kubernetes: threat-driven analysis and defense*. eng. Sebastopol: O'Reilly Media, Incorporated. isbn: 978-1-4920-8173-9 978-1-4920-8170-8.

Murillo, Jenna (ago. de 2024). *Enhancing container security with Aqua Trivy on IBM Power*. en. url: https://community.ibm.com/community/user/powerdeveloper/blogs/jenna-murillo/2024/04/08/enhanced-container-security-with-trivy-on-power (acedido em 17/12/2024).

Open Policy Agent (2024a). *Introduction | Gatekeeper Library*. en. url: https://open-policy-agent.github.io/gatekeeper-library/website/ (acedido em 18/12/2024).

– (dez. de 2024b). *open-policy-agent/gatekeeper*. original-date: 2018-10-26T21:05:57Z. url: https://github.com/open-policy-agent/gatekeeper (acedido em 18/12/2024).

– (2024c). *Performance Tuning | Gatekeeper*. en. url: https://open-policy-agent.github.io/gatekeeper/website/docs/performance-tuning (acedido em 18/12/2024).

Prometheus (dez. de 2024). *prometheus/prometheus*. original-date: 2012-11-24T11:14:12Z. url: https://github.com/prometheus/prometheus (acedido em 18/12/2024).

Prometheus Authors (2025). *Storage | Prometheus*. en. url: https://prometheus.io/docs/prometheus/latest/storage/ (acedido em 19/06/2025).

Quay (dez. de 2024a). *quay/clair*. original-date: 2015-11-13T18:46:16Z. url: https://github.com/quay/clair (acedido em 17/12/2024).

– (2024b). *Deployment Models - Clair Documentation*. url: https://quay.github.io/clair/howto/deployment.html (acedido em 17/12/2024).

Red Hat (2019). *What is Clair?* en. url: https://www.redhat.com/en/topics/containers/what-is-clair (acedido em 17/12/2024).

Red Hat (2024). *Kubernetes adoption, security, and market trends report 2024*. url: https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview.

Russell, Eoghan e Kapal Dev (ago. de 2024). *Centralized Defense: Logging and Mitigation of Kubernetes Misconfigurations with Open Source Tools*. en. arXiv:2408.03714 [cs]. url: http://arxiv.org/abs/2408.03714 (acedido em 16/11/2024).

Sable, Tabitha (abr. de 2021). *PodSecurityPolicy Deprecation: Past, Present, and Future*. en. Section: blog. url: https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/ (acedido em 30/11/2024).

Sayfan, Gigi (2020). *Mastering Kubernetes: level up your container orchestration skills with Kubernetes to build, run, secure, and observe largescale distributed apps, third edition*. en. 3rd ed. Place of publication not identified: Packt Publishing. isbn: 978-1-83921-125-6 978-1-83921-308-3.

Shopify (dez. de 2024). *Shopify/kubeaudit*. original-date: 2017-09-14T20:37:55Z. url: https://github.com/Shopify/kubeaudit (acedido em 17/12/2024).

SIGHUP (dez. de 2024). *sighupio/permission-manager*. original-date: 2019-10-18T14:48:29Z. url: https://github.com/sighupio/permission-manager (acedido em 16/12/2024).

Slik, Maarten van der e Frank Wiersma (2021). «Validating the replacement filtering features of popular alternative admission controllers for Pod Security Policies». Em: *Research Project 2*.

StackRox (dez. de 2024). *stackrox/kube-linter*. original-date: 2020-08-13T17:05:00Z. url: https://github.com/stackrox/kube-linter (acedido em 17/12/2024).

Sukhija, Nitin et al. (nov. de 2020). «Event Management and Monitoring Framework for HPC Environments using ServiceNow and Prometheus». en. Em: *Proceedings of the 12th International Conference on Management of Digital EcoSystems*. Virtual Event United Arab Emirates: ACM, pp. 149–156. isbn: 978-1-4503-8115-4. doi: 10.1145/3415958.3433046. url: https://dl.acm.org/doi/10.1145/3415958.3433046 (acedido em 21/11/2024).

SUSE (dez. de 2024a). *neuvector/neuvector*. original-date: 2021-12-17T20:15:03Z. url: https://github.com/neuvector/neuvector (acedido em 17/12/2024).

– (2024b). *System Requirements | Neuvector Docs*. en. url: https://open-docs.neuvector.com/basics/requirements/ (acedido em 17/12/2024).

Sysdig Inc. (2024). *29 Docker security tools compared*. en-US. url: https://sysdig.com/learn-cloud-native/29-docker-security-tools/ (acedido em 18/12/2024).

The Linux Foundation (2023). *Controlling Access to the Kubernetes API*. en. Section: docs. url: https://kubernetes.io/docs/concepts/security/controlling-access/ (acedido em 07/12/2024).

– (2024a). *Cluster Architecture*. en. url: https://kubernetes.io/docs/concepts/architecture/ (acedido em 15/12/2024).

– (2024b). *Kubernetes Components*. en. Section: docs. url: https://kubernetes.io/docs/concepts/overview/components/ (acedido em 15/12/2024).

– (2024c). *Secrets*. en. Section: docs. url: https://kubernetes.io/docs/concepts/configuration/secret/ (acedido em 15/12/2024).

Vugt, Thomas Martijn van e Tania Malik (nov. de 2023). «A Practical Analysis of Open-Source Security Tools in Microservice Kubernetes Environments». Em: *2023 Cyber Research Conference - Ireland (Cyber-RCI)*, pp. 1–8. doi: 10.1109/Cyber-RCI59474.2023.10671405. url: https://ieeexplore.ieee.org/document/10671405/?arnumber=10671405 (acedido em 16/11/2024).

Westling, Gustav (dez. de 2024). *zegl/kube-score*. original-date: 2018-09-16T13:19:19Z. url: `https://github.com/zegl/kube-score` (acedido em 17/12/2024).

Wohlin, Claes (mai. de 2014). «Guidelines for snowballing in systematic literature studies and a replication in software engineering». en. Em: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. London England United Kingdom: ACM, pp. 1–10. isbn: 978-1-4503-2476-2. doi: `10.1145/2601248.2601268`. url: `https://dl.acm.org/doi/10.1145/2601248.2601268` (acedido em 23/11/2024).