2022 IEEE 15th International Conference on Cloud Computing (CLOUD) | 978-1-6654-8137-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLOUD55607.2022.00022

Stay at the Helm: secure Kubernetes deployments via graph generation and attack reconstruction

Agathe Blaise*, Filippo Rebecchi*
*Thales, Gennevilliers, France. Email: {name.surname}@thalesgroup.com

***Abstract*—In recent years, there has been an explosion of attacks directed at microservice-based platforms – a trend that follows closely the massive shift of the digital industries towards these environments. Management and operation of container-based microservices is automation-heavy, leveraging on container orchestration engines such as Kubernetes (K8s). Helm is the package manager of choice for K8s and provides Charts, i.e., configuration files that define a programmatic model for application deployments. In this paper, we propose a novel methodology for extracting and evaluating the security model of Helm Charts. Our proposal extracts a topological graph of the Chart, whose nodes and edges are then characterised by security features. We carry out risk assessments that refer to the attack tactics of the MITRE ATT&CK framework. Furthermore, starting from these scores, we extract the riskiest attack paths. We adopt an experimental validation approach by analysing a dataset created from multiple publicly accessible Helm Chart repositories. Our methodology reveals that, in most cases, they have vulnerabilities that can be exploited through complex attack paths.**

***Index Terms*—Microservices, cloud computing, containerisation, orchestration, Kubernetes, Helm Charts.**

I. Introduction

Today, containers are widely employed from hyperscalers to private clouds to deploy applications and services, practically replacing traditional Virtual Machines (VMs) as the de-facto DevOps standard due to their inherent scalability and portability advantages [1]. Containers rhyme well with cloud-native microservice architectures, whose operating principle differs considerably from the more traditional monolithic approach [2]. In production environments, the deployment and management of containers are typically delegated to complex automation and orchestration engines such as K8s, Docker Swarm, or AWS Elastic Container Service (AWS ECS). Consequently, this change in paradigm comes with a novel set of challenges and an increased surface of attack [3], exemplified by Man in the Middle (MITM) attacks targeting multi-tenant environments [4], or the recent YoYo attack that deliberately abuses the autoscaling of containers [5].

In this paper we focus on securing K8s deployments, however extending our findings to other engines can follow similar paths. While K8s natively supports some advanced security features, including network segmentation, process isolation, and a Role-Based Access Control (RBAC), in general these policies are not activated by default and require substantial knowledge to be properly configured. Unsurprisingly, insecure practices concerning default configuration are well documented [3], [6], [7]. In this respect, various open source and commercial tools help to provide additional layers of security – however, even the most advanced solutions, such as those recommended by the Center for Internet Security (CIS) [8], simply consist of a list of pass/fail checks, lacking an overall assessment of the likelihood and impact of an attack. It is therefore not possible **to correlate configuration files, user-defined policies, to extract potential points of failure**.

Properly configuring K8s deployments can be quite complex. Deployments can be performed either manually via a series of command-line inputs or, in automation-heavy environments, via deployment files such as Helm Charts. Helm is the package manager of choice for K8s, and Charts consist of deployment-ready collections of YAML manifest files that describe the model for deploying the microservice application containers to K8s [9]. Charts can be quite complex and can result in behemoth configuration files of up to $40K+$ lines of code, virtually impossible to analyse by hand (e.g., [10]). In addition, the same policy can be defined in multiple places via different fields, an error-prone behaviour in collaborative environments.

The main contribution of this paper is a multi-step methodology to evaluate the security and extract the risk model of a Helm Chart deployment as detailed below:

- **Step 1.** A basic topological graph is extracted by parsing, analysing, and correlating various components and policies extracted from the Helm Chart.
- **Step 2.** Extracted data is aggregated into six main features, inspired by CIS K8s best practices [11], [12]. An enriched topological graph encompasses a set of nodes and connecting edges characterised by such features.
- **Step 3.** From the topological graph, the risk associated with each node and edge of the graph is evaluated following the possible attacker tactics as defined in the MITRE ATT&CK framework for K8s [13].
- **Step 4.** Finally, we use a shortest-path algorithm to roam the risk-graph and to extract the (set of) attack path(s) considered the most dangerous/risky on the topology.

We evaluate our methodology via an experimental approach. We have constructed a dataset from multiple publicly accessible Helm Charts repositories. We have implemented a working prototype of our methodology to evaluate the Charts in the dataset. By scoring security features in accordance with CIS benchmark recommendations, we demonstrate that most of them are not secure by default and require substantial modifications to reduce the overall security risks.

The remainder of this paper is structured as follows. Sect. II addresses related work and positions our work. Sect. III

2159-6190/22/$31.00 ©2022 IEEE
DOI 10.1109/CLOUD55607.2022.00022

Tool	Company	Open-source	CIS guide	Custom checks	Pen-testing
kube-bench [16]	Aqua	Yes	K8s		
kube-hunter [17]	Aqua	Yes			✓
sKan [18]	Alcide	No	K8s		
Checkov [19]	Bridgecrew	Yes	K8s		
kAdvisor [20]	Alcide	No		✓	
kubeaudit [21]	Shopify	Yes		✓	
Polaris [22]	Fairwinds	Yes		✓	
Docker Bench [23]	Docker	Yes	Docker		

TABLE I: Classification of state-of-the art tools for microservice descriptor files analysis.

provides hints on how a microservice is built in K8s, outlining the security best practices to adopt. Sect. IV details the graph generation methodology. Sect. V presents the evaluation of the methodology, introducing the dataset that we used, and the analysis carried out. Finally, Sect. VI draws conclusions and points out future work.

II. Related work

We focus our attention on the relevant literature in the fields of K8s security and related threats and attack tactics.

A. Kubernetes security management

While simple applications may be composed of only a few tens of components, the largest ones, such as those operated by Netflix [14] and Uber [15], can run hundreds or thousands of containers. Their composition leads to complex interaction patterns that can be a source of security issues. This complexity is well understood by professionals, who require a thorough security assessment of both K8s clusters and microservice configurations before being deployed in production.

By limiting the scope of our analysis to K8s, different approaches exist to audit the security of such configurations. The first category of tools retrieves configuration files and settings and executes a set of checks based on CIS benchmarks, related both to Docker [11] and K8s [12]. The second category executes custom checks for best practices against provided configuration files and settings. Finally, the third category relies on penetration testing (pen-testing) techniques in the K8s cluster. As a summary, Table I lists the main tools for analysing the K8s configuration descriptors. The *kube-bench* tool strictly follows all the checks listed by the K8s CIS benchmark to discover configuration errors and authorisation and authentication issues, while *kube-hunter* enhances the effectiveness of the analysis with discovery and pen-testing capabilities. Similarly, *sKan* and *checkov* rely on the K8s CIS benchmark to execute a set of checks over the configuration and user-defined policies, whereas *kAdvisor*, *kubeaudit*, and *Polaris* check the conformity of K8s objects using their own sets of checks. Finally, *Docker Bench* limits itself to Docker containers and applies automated checks against common deployment best practices.

Such tools have a number of shortcomings that reduce their effectiveness. First, they only perform a set of checks, without any correlation of the results. For this reason, it is often not possible to diagnose the global impact that a misconfiguration has on the cluster or on the running microservice. Instead, a correlation and aggregation of information from various analyses performed at different levels could allow one to reconstruct a more coherent view of the system. Furthermore, the results of the checks carried out are not put into perspective with the techniques and tactics employed by real-world attackers, for instance following some well-known attack matrices [13].

Recent works focus on K8s manifests and Helm Charts to point out misconfigurations affecting security. [24] examines Helm Charts to assess the quality of declarative chart artefacts, e.g., detecting Charts with no maintainer or with a name collision. [25] applies a qualitative analysis technique called closed coding to spot security shortcomings of K8s manifests, where two users rate collected commits to determine which ones are related to security defects, e.g., looking for keywords as '*injection*' or '*vulnerability*'. However, the methodologies developed in both articles require human intervention, and analyse only the commit logs and not the actual content of K8s manifests.

B. Kubernetes threat matrix

The MITRE Adversarial Tactics, Techniques and Common Knowledge (ATT&CK) framework is an up-to-date database of attack techniques grouped by objectives known as *tactics*. The ATT&CK framework, originally generic, has been specialised for K8s, highlighting multi-stage cyber-attack patterns and techniques that attackers can use to infiltrate and perform damage on K8s clusters [13]. The threat matrix is made up of ten distinct families of tactics that an attacker can combine to reconstruct and exploit possible attack paths.

Up to our knowledge, the K8s threat matrix has been very rarely used in the literature as a means to efficiently design systems considering real-world attack scenarios. Authors in [3] and [26] introduce several attack scenarios derived from the attack life-cycle introduced in the K8s threat matrix.

C. Our contribution

With respect to related works, our methodology can be assimilated to auditing microservice deployment descriptor files through custom compliance checks. However, our proposal goes far beyond the simple execution of compliance checks, as it generates decorated graphs of the deployment, correlating to each other and enriching the components defined in the configuration file. Additionally, our methodology extracts the most significant risks in the deployment and identifies the riskiest attack paths. In this way, it also acts as a useful tool for the decision-making process, as opposed to other solutions that do not provide any prioritisation of the actions to be taken. Furthermore, our methodology embeds natively the K8s threat matrix to model the security of the deployment and detect potential issues with accurate mapping to real-world attacks.

III. K8s deployment model

K8s automates the management of containers and microservices. However, complexity shifts from properly running ap-

plications, to properly configuring their deployment model via descriptor files. In this section, we review the K8s architecture outlining the security best practices to follow.

A. Kubernetes architecture

A container packs together one or more executables and their dependencies. A K8s cluster allows containers to run transparently across multiple (virtual, physical, on-prem, or cloud-based) machines. Each cluster contains at least a master node and one or more worker nodes. Masters have the task of keeping the cluster in the desired state, scheduling the execution of containers on worker nodes, which execute them.

K8s abstracts the configuration of the cluster and the applications deployed on it through a resource-based model. A minority of resources are global, e.g., nodes and cluster roles, but most are scoped through a namespace. The main types of resources defined in K8s include pods, deployments, daemon sets, network policies, (cluster) roles, (cluster) role bindings, persistent volumes, persistent volume claims, secrets, and services.

All interactions between entities (including between administrators, developers, and deployed containers) in the K8s cluster go through an *API server*, which is the one-stop shop for declaring and managing cluster resources, with the server also handling authentication. Although the API server is the interface, the K8s brain resides in *etcd*, a distributed key-value store that holds all the states of the cluster (all other components are stateless). Data, including secrets, is not encrypted by default in *etcd*. Other key components for cluster operation are the *scheduler*, which assigns execution pods to worker nodes; the *kubelet*, which runs on each node to ensure the proper functioning of pods programmed by the scheduler; the *kube-proxy*, which ensures the routing of the traffic for a service IP to the correct pods; and finally the *Container Network Interface (CNI)*, which guarantees the traffic routing between the nodes, the application of network policies, and possibly traffic encryption[1].

B. Security best practices

As with any system, container engines and orchestrators may have vulnerabilities. Nonetheless, most of the critical security issues are attributable, directly or indirectly, to the exploitation of vulnerabilities and misconfigurations in user-deployed applications. As a matter of fact, although the abstraction layer added by K8s reduces the complexity of managing software and hardware resources, it can help increase the attack surface if set up improperly [27]. For this reason, it is important to clarify the best practices to follow in the management of K8s clusters and the applications deployed on them [6]. These best practices lay the foundation for the analyses that constitute the methodology for evaluating the security of Helm Charts that we propose.

[1]It is worth noting that CNIs are not part of the core K8s components and are enabled through external plugins.

Pod Security Policy (PSP): The PSP mechanism allows defining security policies on pods running on the cluster. These policies are useful to restrict the resources one can define, e.g., prohibiting containers from accessing the Linux namespaces of the host, prohibiting the use of the *root* accounts in containers, or controlling the Linux capabilities of pods.

Secrets management: Secrets store sensitive information that can be accessed by running pods, such as passwords, authentication tokens, or keys. Secrets should be accessible only by the applications needing them. Network access to *etcd* and other administrative interfaces should also be filtered through *Network Policies (NPs)*. Access to *etcd* by API servers should be protected by mutual authentication. The local mounting of secrets and control sockets stored on nodes should be prohibited via a PSP.

Access to K8s services: Core K8s services, such as the DNS and the API server, are deployed in the *kube-system* namespace. By default, they are accessible from any other namespace. Therefore, a best practice is to prevent any traffic from other namespaces to reach *kube-system* via explicit *NPs*, with the exception of the applications that really require access to those services.

Access Control: The permissions granted to users and applications should be reduced as much as possible to prevent them from interacting with the cluster configuration, in particular the API server. Service accounts handle users with authentication based on tokens, which are automatically mounted by default and made available in the containers. RBAC policies control the access to APIs exposed by the API server. Roles can be defined with (cluster) role resources to authorise a list of paths (endpoints) and possible actions on each path (HTTP verbs: get, list, watch, update, etc.). Roles are assigned to service accounts, users or groups, by the declaration of (cluster) role bindings.

IV. Model reconstruction from microservice deployment descriptors

The overall methodology is composed of 4 steps:

- **Step 1**: Parse the descriptors files and correlate various resources to extract a basic topological graph composed of nodes and edges;
- **Step 2**: Enrich the topological graph through six main features in line with the CIS K8s best practices, namely: 1) vulnerabilities of containers, 2) pods accessibility, 3) the security policies in places among pods, 4) access control (RBAC) rules, 5) firewall rules, and 6) affinities among deployed components;
- **Step 3**: Score the level of danger associated to the topological graph for each attacker tactic from the K8s threat matrix;
- **Step 4**: Identify the riskiest attack paths according to each step of the attack scenarios defined at the previous step.

Fig. 1 provides an overview of the overall methodology, with the four main steps detailed in the following.

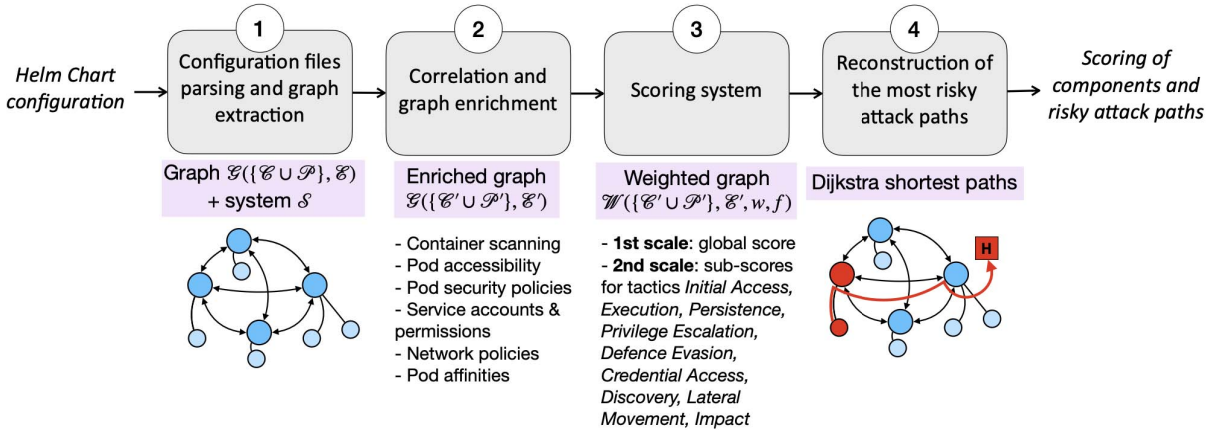


Fig. 1: Overview of the proposed multi-step methodology. Step 1: Parsing, correlation of the components, and basic topological graph extraction. Step 2: Graph enrichment through six analysis. Step 3: Scoring of the level of risk associated to each node and edge of the graph (two distinct scales). Step 4: Identification of the riskiest attack paths.

Symbol	Meaning
$\mathcal{P}$	Set of pods, deployments, daemon sets, and replica sets in S
$\mathcal{N}$	Set of namespaces
$\mathcal{C}$	Set of containers and initial containers
$\mathcal{S}v$	Set of services
$\mathcal{A}$	Set of service accounts
$\mathcal{R}$	Set of roles
$\mathcal{R}c$	Set of cluster roles
$\mathcal{B}$	Set of role bindings
$\mathcal{B}c$	Set of cluster role bindings
$\mathcal{P}s$	Set of pod security policies (PSP)
$\mathcal{N}p$	Set of network policies (Network Policy (NP))
$\mathcal{S}c$	Set of secrets
$\mathcal{V}$	Set of volumes
$\mathcal{M}$	Set of volume mounts
$\mathcal{P}v$	Set of persistent volumes
$\mathcal{P}c$	Set of persistent volume claims

TABLE II: Resources of the K8s model.

Variable	Description
$\mathcal{S}$	Resources from the overall deployment system
$\mathcal{F}$	Functions applied to resources of $\mathcal{S}$
$\mathcal{C}'$	Set of enriched containers
$\mathcal{P}'$	Set of enriched pods
$\mathcal{E}'$	Set of enriched edges
w	Single-weighting function for nodes
w'	Multi-weighting function for nodes
z	Single-weighting function for edges
$\mathcal{G}(\{\mathcal{C}' \cup \mathcal{P}'\}, \mathcal{E}')$	Enriched directed graph
$\mathcal{W}(\{\mathcal{C}' \cup \mathcal{P}'\}, \mathcal{E}', w, z)$	Edge- and node-weighted enriched directed graph
$\mathcal{W}'(\{\mathcal{C}' \cup \mathcal{P}'\}, \mathcal{E}', w, z)$	Edge- and node- multi-weighted enriched directed graph

TABLE III: Notations.

A. Helm Chart parsing and graph extraction

The first step has the double objective of parsing the Helm Chart to feed a data structure holding the overall deployment system resources and to build topological information about how resources of a deployment are correlated together.

Let S represent the overall deployment system, composed of multiple resources as defined in Table II, such that:

$$S = \{\mathcal{P}, \mathcal{N}, \mathcal{C}, \mathcal{S}v, \mathcal{A}, \mathcal{R}, \mathcal{R}c, \mathcal{B}, \mathcal{B}c, \mathcal{P}s, \mathcal{N}p, \mathcal{S}c, \mathcal{V}, \mathcal{M}, \mathcal{P}v, \mathcal{P}c\}$$

We describe in Table III the notations we use in the paper.

These resources can be related together using formal constraints, that are expressed as follows:

- each resource $r \in S$ belongs to a unique namespace $n \in \mathcal{N}$;
- each pod (or deployment) $p \in \mathcal{P}$ encapsulates at least one (initial) container $c \in \mathcal{C}$ and a volume $v \in \mathcal{V}$ shared among the containers[2];
- each volume $v \in \mathcal{V}$ is associated with a volume mount $m \in \mathcal{M}$ that specifies where to mount the volume on the containers;
- each persistent volume claim $c \in \mathcal{P}c$ consumes a persistent volume $v \in \mathcal{P}v$;
- each service $s \in \mathcal{S}v$ targets at least one pod $p \in \mathcal{P}$;
- each (cluster) role binding $b \in \{\mathcal{B} \cup \mathcal{B}c\}$ links a (cluster) role $r \in \{\mathcal{R} \cup \mathcal{R}c\}$ to a list of subjects (users, groups, or service account $a \in \mathcal{A}$);
- each PSP $p \in \mathcal{P}s$ characterises the security context of at least one pod $p \in \mathcal{P}$;
- each pod $p \in \mathcal{P}$ is associated with at least one PSP $p \in \mathcal{P}s$ or with a security context in its definition;
- each NP $n \in \mathcal{N}p$ links one pod $p \in \mathcal{P}$ to others pods $p \in \mathcal{P}$, namespaces $n \in \mathcal{N}$, or IP blocks.

[2]In K8s, a pod is the encapsulation of tightly bounded containers, with shared storage, a unique IP address, and options that control how the container(s) should perform. A deployment provides declarative updates for pods. We will refer as "pod" to talk about pods, deployments, daemon sets, stateful sets, and replica sets in the remainder of the article.

By correlating all the resources of the system together, we can build a topological graph with a coherent set of resources. Let $\mathcal{G}(\{\mathcal{C} \cup \mathcal{P}\}, \mathcal{E})$ be a directed graph with vertices representing the union of containers $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ and pods $\mathcal{P} = \{p_1, p_2, ..., p_m\}$ present in the K8s deployment. The set of edges $\mathcal{E}$ connecting the vertices is created as follow:

- A directed edge $e(p, c)$ is created from pod $p \in \mathcal{P}$ towards the container $c \in \mathcal{C}$ that it encapsulates;
- A directed edge $e(p_1, p_2)$ is created between each pair of pods p_1, p_2 belonging to $\mathcal{P}$. Considering several configurations, including network policies, pod execution permissions, and co-localisation of pods on the same K8s node, pod-to-pod edges model the ability to move from one pod to another.

Following the application of the list of relations between all resources, each resource $r \in \{\mathcal{N}, \mathcal{C}, \mathcal{S}v, \mathcal{A}, \mathcal{R}, \mathcal{R}c, \mathcal{B}, \mathcal{B}c, \mathcal{P}s, \mathcal{N}p, \mathcal{S}c, \mathcal{V}, \mathcal{M}, \mathcal{P}v, \mathcal{P}c\}$ is directly linked to a pod $p \in \mathcal{P}$ or linked to a resource itself linked to pod p, so that a coherent set of the aforementioned resources may be created around a single pod.

B. Graph enrichment

The second step has the objective to enrich the previously generated topological graph $\mathcal{G}$ via the analyses of the resources stored in S.

Let $\mathcal{G}(\{\mathcal{C}' \cup \mathcal{P}'\}, \mathcal{E}')$ be an enriched directed graph with $\mathcal{C}'$ the set of enriched containers and $\mathcal{P}'$ the set of enriched pods. Each $c' \in \mathcal{C}'$ has a set α of attributes derived from the graph enrichment made from various security analyses. An enriched container is a container along with additional information describing it, provided from other resources related to it. Similarly, each $p' \in \mathcal{P}'$ has a set β of attributes derived from the graph enrichment process. An enriched pod has additional information describing it, provided by other resources related to it.

Let $\mathcal{E}' = \{e'_1, e'_2, ..., e'_o\}$ with $o = |\mathcal{E}'|$. Each $e' \in \mathcal{E}'$ has a set γ of attributes derived from the graph enrichment. Let $\mathcal{F} = \{f_1, f_2, ..., f_i\}$ $(i = |\mathcal{F}|)$ be the set of functions applied to resources of $\mathcal{S}$ to perform various security analyses and compute attributes α, β, and γ of resources belonging respectively to $\mathcal{C}'$, $\mathcal{P}'$, and $\mathcal{E}'$. A non-exhaustive list of security analysis functions may include:

- **Vulnerability scan of containers**: $f_1 : \mathcal{C} \rightarrow \mathcal{C}'$.
  The number of high or critical severity vulnerabilities, i.e., those whose Common Vulnerability Scoring System (CVSS) score is greater than 7, associated with each container is recorded. Image vulnerability scanning is the process of reviewing the security state of the container images. Image vulnerability scanners retrieve the Operating System (OS), software packages and libraries used in an image and compare them against vulnerabilities repositories. We leverage on open source tools such as Trivy [28], Docker Scan [29], and Snyk [30];
- **Services associated with pods**: $f_2 : \{\mathcal{P}, \mathcal{S}v\} \rightarrow \mathcal{P}'$.
  The services abstract the accessibility rules of pods: these can be exposed only inside the cluster (*ClusterIP*), or externally to the cluster via a static port (*NodePort*), an external load balancer (*LoadBalancer*), or an HTTP routing service (*Ingress*);
- **Pod security policies and security context** of containers and pods: $f_3 : \{\mathcal{C}, \mathcal{P}, \mathcal{P}s, \mathcal{V}, \mathcal{M}, \mathcal{P}v, \mathcal{P}c\} \rightarrow \mathcal{P}'$.
  This category embeds information on the pods' and containers' environment, including the authorised and prohibited Linux capabilities, the different types of authorised volumes, the permissions associated with files (read-only or write), the authorised users, e.g, `root` account or not, the permission or restriction to launch privileged containers, the authorised or unauthorised use of the host machine's network, the list of authorised ports, the authorised or unauthorised use of the host Inter-Process Communication (IPC) and Process ID (PID) namespaces, etc.;
- **Service accounts associated with pods** and their permissions: $f_4 : \{\mathcal{P}, \mathcal{A}, \mathcal{R}, \mathcal{R}c, \mathcal{B}, \mathcal{B}c\} \rightarrow \{\mathcal{P}', \mathcal{E}'\}$.
  As defined in Sect. III-B, these permissions include a list of endpoints and possible actions on each endpoint, e.g., creating new pods and workloads, reading secrets, executing other pods, etc.;
- **Network policies**: $f_5 : \{\mathcal{P}, \mathcal{N}p\} \rightarrow \mathcal{E}'$.
  They define firewall rules governing the interactions between the pods, e.g., which pods can communicate with each other, on which port and which protocol;
- **Pod affinities between different pods**: $f_6 : \mathcal{P} \rightarrow \mathcal{E}'$.
  They represent the fact that two pods must be located on the same machine (pod-affinity), or on the contrary on two different machines (anti pod-affinity).

C. Graph weighting and scoring

The third step has the objective of quantifying the security level of a Helm Chart, deriving information from the enriched graph computed at the previous step. The process of scoring is actually related to computing weights associated with each node and edge of the enriched graph. Two distinct scoring systems are provided, reflecting different levels of detail.

1) Single-weight scoring system: The first scoring technique associates a single weight to each node and each edge of the enriched graph. This aims to model the overall risk level associated with each component, aggregating all the security assessments into a single value.

Let $\mathcal{W}(\{\mathcal{C}' \cup \mathcal{P}'\}, \mathcal{E}', w, z)$ be an edge- and node-weighted enriched directed graph derived from $\mathcal{G}(\{\mathcal{C}' \cup \mathcal{P}'\}, \mathcal{E}')$. Let $w : \{\mathcal{C}' \cup \mathcal{P}'\} \rightarrow \mathbb{R}$ and $z : \mathcal{E}' \rightarrow \mathbb{R}$ be the single-weighting functions respectively for nodes and edges of the graph. If $(u, v) \in \mathcal{E}'$, $z(u, v) \in [0, 1]$ as weights represent probabilities and are multiplicative. For convenience, we define $z(u, v) = 0$ if $(u, v) \notin \mathcal{E}'$.

2) Multiple-weight scoring system: The second scoring technique is more complex, as it evaluates the components with respect to the offensive tactics and techniques of the K8s threat matrix [13] introduced in Sect. II-B. For each node and edge of the enriched graph, a sub-score is assigned for each tactic of the matrix.

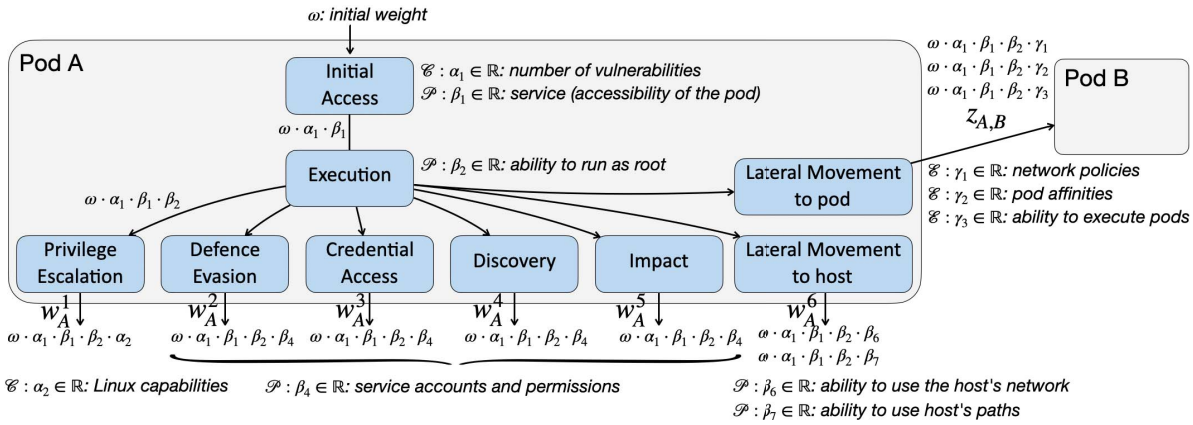


Fig. 2: Multiple-weight scoring system.

Let $\mathcal{W}'(\{\mathcal{C}' \cup \mathcal{P}'\}, \mathcal{E}', w', z)$ be an edge- and node- multi-weighted enriched directed graph derived from $\mathcal{G}(\{\mathcal{C}' \cup \mathcal{P}'\}, \mathcal{E}')$. Let $w' : \{\mathcal{C}' \cup \mathcal{P}'\} \rightarrow \mathbb{R}^q$ be the multi-weighting function for nodes of the graph, with q the number of tactics of the K8s threat matrix that are evaluated, and $z : \mathcal{E}' \rightarrow \mathbb{R}$ the single-weighting function for edges of the graph. Especially, $w'(n) = \{w'^1(n), ..., w'^t(n), ..., w'^j(n)\}$ with $w'^t(n)$ the multi node-weights of node n for tactic t. As before, let $z(e)$ be the single edge-weight of edge e.

The weighting functions can be customised by the administrator. As an example, Fig. 2 illustrates a possible weighting system. In this setting, nine tactics from the K8s threat matrix are considered. The *Initial Access* and *Execution* tactics are first employed to model the ability for an attacker to enter a pod of the cluster and execute malicious actions. Then, several end-goal tactics that the attacker may perpetrate on the cluster are considered, and in particular *Privilege Escalation*, *Defence Evasion*, *Credential Access*, *Discovery*, *Impact*, and *Lateral Movement to host*. The ability to move to other pods is also considered as an end-goal tactic via the *Lateral Movement to pod*. The final weight for each end-goal tactic of Pod A equals the product of the weights associated to each sub-node on the path. For example, the final weight denoted $w'^{pr\_esc}(A)$ of the end-goal *Privilege Escalation* (pr_esc) is the product of ω, β_1, β_2, and α_2. Also, the initial weight of Pod B equals the output weight of the *Lateral Movement to pod* from pod A, so that $z'(A, B) = w'^{lat\_mov}(A)$.

D. Identification of dangerous attack paths

The fourth and final step has the objective to find a set of one or more attack paths that are considered the most risky. By employing the weights defined in step 3, the objective is to find the path $s = \langle n_1, n_2, n_3, \ldots, n_{k-1}, n_k \rangle$ that maximises the weight $z(n_1, n_2) \cdot z(n_2, n_3) \cdot ... \cdot z(n_{k-1}, n_k)$. Given that edge weights represent probabilities, the objective is to find the set of acyclic paths with the largest aggregate score.

$$\max_{p} \prod_{e \in p} z(e) \tag{1}$$

Inverting the edge weight transforms it into a minimisation problem, which consists of finding the shortest path by product of edge distances. Further, by taking the logarithm of edge distances to make summation, the final problem becomes a shortest path problem, so that:

$$\min_{p} \sum_{e \in p} \log(\frac{1}{z(e)}) \tag{2}$$

We employ Dijkstra's algorithm to find the shortest path between several nodes of our graph [31], [32].

V. Evaluation

For evaluating our methodology, we have built a dataset of Helm Chart configurations from multiple publicly available repositories. The evaluation employs our methodology to assess the adherence of the Charts in the dataset to common K8s security best-practices. We also highlight the results of the analysis on specific Helm Charts showing how our methodology can help pinpointing typical misconfigurations.

A. Dataset

Helm [9], [24] is the equivalent of a package manager for K8s and employs a packaging format called Charts. A Helm Chart packs a set of (deployment-related) K8s resources into a single YAML configuration file. Thus, a single Chart might be used to deploy something simple, like a Memcached pod, or even a full web app stack with multiple server replicas, databases, caches, load balancers, and so on.

The dataset is built by scraping and assembling the Helm Charts from 14 open-source repositories[3], namely *prometheus-community*, *kube-state-metrics*, *renovate*, *artefact*, *hashicorp*, *enix*, *klustair*, *camunda*, *hpe-storage*, *nutanix*, *bitnami*, *halk-eye*, *ibm-charts*, and *helm-charts*. In total, the collected dataset consists of 592 official Helm Charts, totalling 1076 pods and 1318 containers. These represent the basic targets of our analysis. We observe that configuration files of Helm Charts

[3]Helm Charts from a given repository can be downloaded via the command line `helm repo add <repo-name> <repo-url>`.

CIS rule	# vuln. containers	Proportion
Root containers (#5.2.6)	206	32.2%
Privileged containers (#5.2.1 & #5.2.5)	55	4.2%
Default Linux capabilities (#5.2.8 & #5.2.9)	1137	86.3%
NET_RAW capability (#5.2.7)	1170	88.8%
Get list of secrets (#5.1.2)	129	9.8%
Get list of pods (#5.1.3)	183	13.9%
Creating pod (#5.1.4)	107	8.1%
Host path volumes (#5.1.4)	13	1.0%
Host network (#5.2.4)	10	0.8%

TABLE IV: Number and proportion of containers (out of 1318 in total) not complying with K8s CIS benchmark rules.

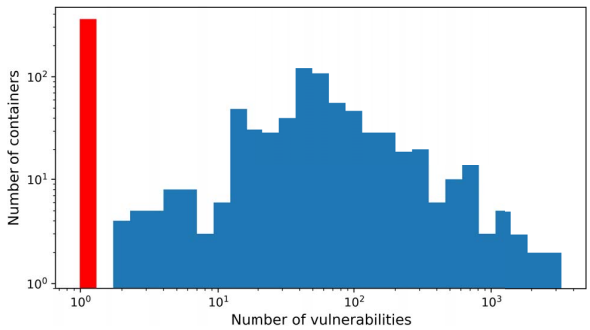


Fig. 3: Distribution of the number of vulnerabilities per container for all the container images in the Helm Charts.

in the dataset can contain up to 43,628 lines, with an average of 564 lines per Chart. Of course, the interesting ones are those with the largest numbers of pods, as they have the most complex interactions between pods and the largest numbers of lines to analyse, which therefore require considerable effort in the case of manual analysis.

B. Compliance with the Kubernetes CIS benchmark

The K8s CIS benchmark proposes prescriptive guidance for establishing a secure configuration posture for K8s [11]. Among other configurations, the benchmark covers RBAC and service accounts, PSPs, network policies and CNIs, secrets management, extensible admission control, and general policies. We aim to assess how the Helm Charts in our dataset comply with these sets of rules. Table IV summarises the number of containers we identified as not compliant with the rules of the CIS benchmark, using the parsing of K8s resources made by our tool. We observe that for several rules a large share of containers does not comply with them.

In fact, root containers are around a third of the total, which is quite concerning as they greatly facilitate the job of escalating privileges once an attacker has reached a container. The proportion of privileged containers is also concerning. Furthermore, in most configurations, all 14 Linux capabilities granted by default to containers are allowed, with no restrictions on the most harmful ones. For example, the NET_RAW capability allows the exchange of ICMP traffic between containers, granting as well a container the capability to craft raw packets. Regarding RBAC and service accounts, around 10% of all containers have the capability to retrieve the list of secrets, the list of running pods, and to create pods. Finally, the abilities to access the host network and to mount volumes on the host are granted in around 1% of all cases, which is quite reasonable.

Fig. 3 shows the distribution of the number of high or critical severity vulnerabilities (CVSS $>$ 7.0) found per container image present in the Helm Charts, by employing image vulnerability scanners. The red bar on the left represents opaque containers, i.e., container images that cannot be analysed by image vulnerability scanners[4]. Moreover, we observe that the median number of high or critical severity vulnerabilities is centred around 100, reaching in some cases 3000 vulnerabilities for a single container, e.g., '*unguiculus/docker-python3-phantomjs-selenium:v1*' (3282 vulnerabilities), '*halkeye/slack-resurrect:v0.1.4*' (2265 vulnerabilities), and '*k8s.gcr.io/spark:1.5.1_v3*' (1682 vulnerabilities). Vulnerabilities of containers represent critical entry doors that attackers can leverage to infiltrate a cluster, and therefore should be taken into account in the analysis of the overall cluster security.

To summarise, we observe that a large portion of the Helm Charts in our dataset does not conform to some of the recommendations provided by the K8s CIS benchmark. While this is already worrying by itself, we also point out that simple compliance does not guarantee the absence of security problems and potentially harmful attack sequences. In reality, the main problem lies in the non-correlation of the different elements of the analysis into a single global model that synthesises the information. In fact, single scans can overlook important insights such as the most important assets to protect, the most critical security gaps to address, and which problems have to be resolved first. In addition, the resilience of each configuration to various possible attack tactics is not assessed. Therefore, our model correlating the components, outputting graphs and deriving risky attack paths is necessary to get an accurate vision of the overall level of defence and risk of the configuration.

C. Connectivity among components

Hereafter, we aim to illustrate the necessity to take into account the connectivity between components in our analysis. A pod might be reachable by an attacker from the outside if the service associated with it is "*NodePort*" or "*LoadBalancer*", meaning that components located outside the cluster can potentially reach it. Afterwards, the possibility for an attacker to move inside the cluster, e.g., from one pod to another, and

[4]Typically, a container image cannot be analysed for the following reasons: (*i*) the image's footprint has been minimised, (*ii*) the image has been made from scratch, (*iii*) the base image is outdated, or (*iv*) the base image is too recent and not yet handled by the analysis tool.

Configuration	Reach. cont.	Increase factor
Initially reachable containers	134	1
Considering pod execution permissions	135	1.01
Considering the lack of network policies	293	2.19
Considering pod affinities	315	2.35

TABLE V: Number of reachable containers (out of 1318 containers in total) considering various configurations.

thus to reach several pods from a single entry point, should also be considered. This agility can be made possible by the lack of network policies implemented, to high-privileged users permissions enabling to execute other pods, or to pod affinities co-locating several pods on the same node.

Table V reports the number of reachable containers, including the numbers of initially reachable ones (due to their service accessibility configurations) and of the ones reachable by transitivity due to the three aforementioned techniques. Initially, 134 out of the 1318 containers in the dataset are potentially reachable from the outside considering only their service accessibility. Considering pod execution permissions enables to reach only one additional container, which means that most high-privileged pods are well isolated. Then, 293 containers are reachable in total considering the lack of network policies and 315 ones considering pod affinities. In total, 315 containers are reachable considering the ability of the attacker to move in the cluster due to all these factors, which represent an increase factor of 2.35. Note that the ability for an attacker to move across pods can be more accurately weighed depending on the employed technique, e.g., the lack of network policies between two pods would give the attacker 80% of a chance to move, while the pod execution permission would give the attacker 100% success rate.

D. Considering K8s threat matrix tactics individually

As just mentioned, the coefficients associated with each setting can be tuned to weigh the importance of each step in the attack life-cycle, so that the global score equals the weighted sum of all attributes. Table VI shows the values of the coefficients chosen for each setting.

Fig. 4 breaks down the danger scores produced by the scoring module, including the global score (e.g., the single-weight score defined in Sect. IV-C1) and the multiple sub-scores related to the K8s threat matrix tactics (as defined in Sect. IV-C2). The large red dots picture the overall danger level of the containers' configuration, considering the various security configurations related to it. By extension, the container inherits from security configurations of the pod that encapsulates it. The blue dots represent the number of vulnerabilities of the containers. Numerous values around 100 actually represent the containers that could not be scanned by state-of-the-art tools. The orange dots relate to the "*Execution*" tactic, acknowledging the ability of a user to run as root in the container or not. The green dots correspond to the number of Linux capabilities granted to containers, ranging between 0 and 19. The average number equals 14, coinciding with the capabilities granted to Docker containers by default. The red squares relate to the "*Service Accessibility*" tactic, whether the pod may be accessible from the outside or not. The purple squares represent the "*Privilege Escalation*" tactic, whether the user can escalate its privileges or not, e.g., the field '*allow_privilege_escalation*' or '*privileged*' is present in the PSP or the user can create bindings. The brown squares correspond to the "*Permissions*" tactic, including defence evasion, credential access, discovery, impact (create), and impact (delete). Values equal to 0 are not shown in the figure because of the logarithmic scale.

Notation	Description	Value
α_1	Number of vulnerabilities	1 (per vuln.)
α_2	Number of Linux capabilities	5 (per cap.)
β_1	Accessible service	100
β_2	Ability to run as root	100
β_3	Can allow privilege escalation	100
β_4	Permissions (defence evasion, credential access, discovery, impact create & delete)	10 (per tactic)
β_5	Can exec other pods	100
β_6	Can reach the host network	100
β_7	Can reach host paths	100
β_8	Allowed types of volumes on host	10 (per vol.)
γ_1	Network policies	0.8
γ_2	Pod affinities	0.4
γ_3	Pod execution capacity	1

TABLE VI: Values of coefficients chosen for each setting.

We observe that the first 150 containers have a global danger score higher than 300, which is very concerning. Then, the danger scores are slightly decreasing, reaching values located between 100 and 300. Finally, a hundred containers have low danger scores, lower than 200. The global danger score gives a good overview of the security of a pod, but heterogeneous reasons may hide behind high danger scores. Three illustrative examples of pods retrieved from Helm Charts' configurations with rather similar high danger scores are provided hereafter:

(*i*) The pod named "*node-exporter*" (score: 470) from Chart "*kube-prometheus*" released by "*bitnami*" can access the host machine's network on all ports, can share the host PID, and can mount and write on volumes on the host;

(*ii*) The pod named "*pachd*" (score: 411) from Chart "*pachyderm*" released by "*stable*" is privileged and is associated with a service account with many permissions in the cluster, including the abilities to get secrets and to get, delete, and create pods;

(*iii*) The pod named "*ibm-skydive-dev-analyzer*" (score: 232) from Chart "*ibm-skydive-dev*" released by "*ibm-charts*" has been granted 5 additional Linux capabilities, i.e., SYS_ADMIN, SYS_RESOURCE, SYS_TIME, NET_BROADCAST, NET_ADMIN, in addition to the 14 default ones, reaching 19 capabilities in total. A user in the pod can potentially modify the network interface on the host and can get a root session on the host machine.

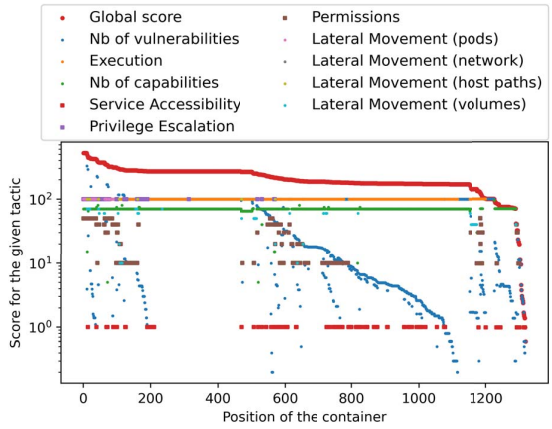


Fig. 4: Breakdown of scores produced by the scoring module (logarithmic scale).

E. Reconstructing complete attack paths

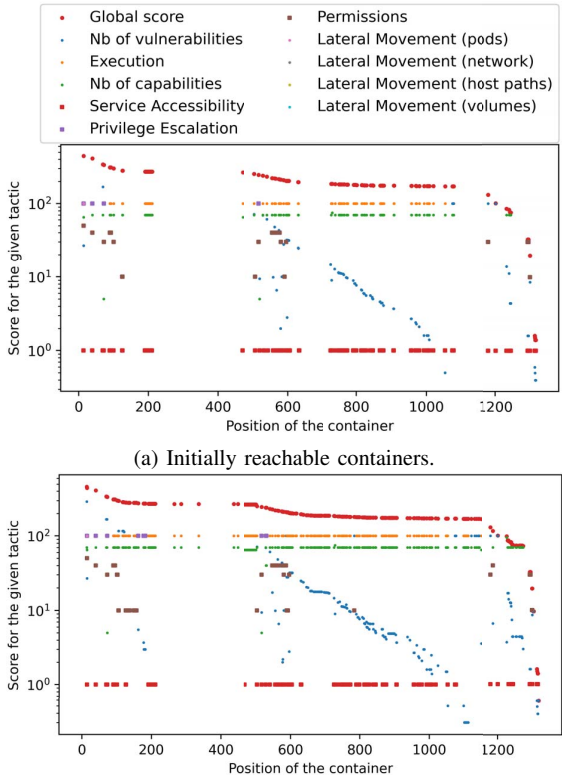
Threat matrix tactics considered individually can help us identify alarming configurations in the Helm Charts, but by themselves they do not provide sufficient visibility to the overall risk of a given attack or threat. Indeed, refined attacks follow complex patterns that involve multiple steps.

In the following, we employ the methodology defined in Section IV-D to assess the number of containers vulnerable to different tactics as end-goals. For this analysis, we distinguish between two settings, namely *initially reachable containers* and *complete attack paths*, defined as follows:

- **Initially reachable containers:** this setting does not consider edges between pods. The weight of each of the nodes equals the product of its service accessibility and the score of the given K8s threat matrix tactic's score. The service accessibility value is Boolean, e.g., either 0 (if the service is '*ClusterIP*') or β_1 (otherwise).
- **Complete attack paths:** this setting considers all edges between pods, so that complete attack paths can be reconstructed. The weight of the initial pod on the path equals the value of its service accessibility, e.g., either 0 or β_1. The weights of the edges are located between 0 and 1, and the weights of intermediate pods are set to 1. The weight of the final pod equals the score of the given tactic's score for the pod.

Fig. 5 illustrates a breakdown of the global scores calculated for each container by attack tactic. Fig. 5a plots the results for the *initially reachable containers* setting, while Fig. 5b considers the *complete attack paths* setting. In Fig. 5b, the plotted values represent the maximum score among all the possible shortest paths by destination container. The differences between the two figures are therefore attributable to the accounting of complete attack paths which extends considerably the scope of an attack to a significantly larger fraction of containers.

An extension to this model recognises the possibility for an attacker to gain access to pods even without them being



(a) Initially reachable containers.

(b) Considering all possible paths toward the containers.

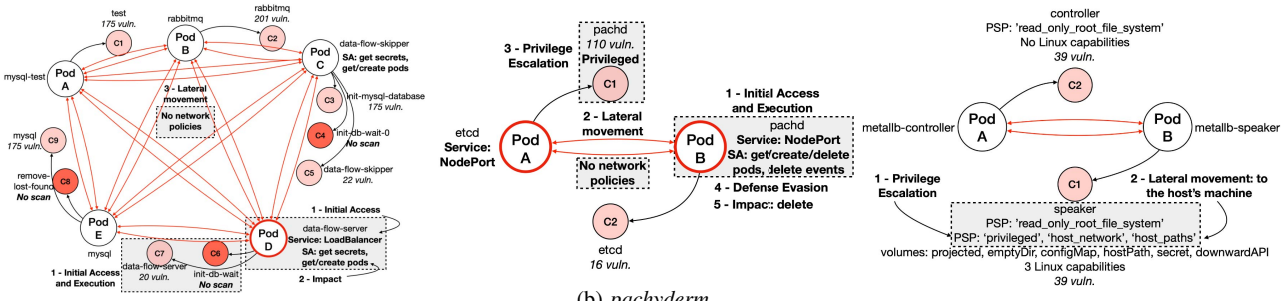
Fig. 5: Comparison between initially reachable containers and complete attack paths (logarithmic scale).

exposed to the outside world. This is possible by exploiting e.g., supply-chain attacks, vulnerabilities present in K8s, or in containerisation technologies. To model this situation, a parameter ω is introduced to represent the default accessibility value of a pod for the attack path computation. Until now, *Service Accessibility* was set to 0 when the service type was defined as '*ClusterIP*'. ω is a probability, e.g., a value of 0.1 would mean that the pod may be accessible by an attacker with a 10% chance even if the service is '*ClusterIP*'. Table VII provides the number of containers vulnerable to the different tactics, for $\omega = 0$ and only *initially reachable containers* (e.g., as in Fig. 5a), $\omega = 0$ and *complete attack paths* (as in Fig. 5b), and $\omega > 0$. Using $\omega > 0$, we observe that far more containers are vulnerable to the various attacker tactics. Risks are even more prominent for the three tactics "*Access to host's network, paths, and volumes*", as respectively 72, 13, and 10 containers may be leveraged to perform such tactics.

Fig. 6 displays three examples of output graphs produced by our methodology and a representation of risky attack paths. Note that only partial information about nodes is provided for better readability. Fig. 6a outputs the graph for the "*spring-cloud-data-flow*" Chart from "*stable*" repository, with 5 pods and 9 containers in total. The complete attack path may be reconstructed from the beginning. Pod D can be reached from

Tactic	Example	$\omega = 0$	$\omega = 0$ with reachability	$\omega > 0$
Initial access	spring-cloud-data-flow (stable); *Fig. 6a*	122	122	1318
Execution	spring-cloud-data-flow (stable); *Fig. 6a*	103	252	1112
Privilege escalation - Capabilities	spring-cloud-data-flow (stable); *Fig. 6a*	4	13	92
Privilege escalation - Run as root	pachyderm (stable); *Fig. 6b*	112	285	1194
Permissions	spring-cloud-data-flow (stable); *Fig. 6a*	23	43	224
Access to host network	metallb (stable); *Fig. 6c*	0	0	72
Access to host paths	metallb (stable); *Fig. 6c*	0	0	13
Access to host volumes	metallb (stable); *Fig. 6c*	0	0	10

TABLE VII: Number of containers vulnerable to each K8s threat matrix tactic, for three configurations.



(a) *spring-cloud-data-flow* (b) *pachyderm* (c) *metallb*

Fig. 6: Output of our methodology, with the description of the components and the reconstruction of attack paths.

the outside through its service *LoadBalancer*, the opaque container C6, and the container C7 with 20 high or critical severity vulnerabilities (*Initial Access & Execution*). Given the service account of the pod, the attacker is able to get secrets in the cluster, to get and create pods (*Credential Access, Discovery, Impact*). Due to the lack of network policies, the attacker may freely navigate through the pods, without being blocked by firewall rules (*Lateral Movement*). The illustration on Fig. 6b from Chart "*pachyderm*" from "*stable*" repository shows an attack path from Pod B associated with a *NodePort* service and encapsulating container C2 with 16 high or critical severity vulnerabilities (*Initial Access & Execution*). Moving to Pod A accounting the lack of network policies (*Lateral Movement*), the attacker can obtain higher privileges (*Privilege Escalation*) and potentially avoid defence systems (*Defence evasion*), create new workloads, and delete resources (*Impact*) with the permissions associated with its service account on Pod B. The illustration on Fig. 6c from Chart "*metallb*" from "*stable*" repository shows a configuration with pod security policies allowing high privileges, e.g., the ability to use the host network and to mount volumes on the host machine (*Lateral Movement*).

VI. Conclusion and Perspectives

The emergence of cloud-native microservices architectures and orchestration engines such as K8s comes with a whole set of security challenges that need to be addressed. In this paper, we present a novel methodology to analyse a K8s Helm Chart deployment. The proposed methodology translates Helm Charts into topological graphs, performing on it various security analyses, outputting a security score and a set of risky attack paths backed by the MITRE ATT&CK framework. We evaluated our methodology via an experimental approach, demonstrating its applicability to hundreds of Helm Charts that contain delicate points whose developers should be aware.

Future research directions are manifold. First, our methodology can benefit from improved mechanisms to analyse container vulnerabilities. Currently, we are not always able to scan some containers which are therefore flagged as "opaque" and risky. Furthermore, attack paths reconstruction opens up interesting perspectives in terms of automated remediation and prioritisation of fixes. Finally, an applied perspective is to integrate our methodology as a step of a CI/CD security pipeline dedicated to K8s.

Acknowledgment

This work is funded under the Assurance and certification in secure Multi-party Open Software and Services (Assure-MOSS) Project (https://assuremoss.eu/en/), with the support of the European Commission and H2020 Program, under Grant Agreement No. 952647.

References

[1] L. A. Vayghan *et al.*, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in *International Conference on Cloud Computing (CLOUD)*. IEEE, 2018.

[2] S. Newman, *Building microservices : designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015.

[3] F. Minna *et al.*, "Understanding the Security Implications of Kubernetes Networking," *IEEE Security & Privacy*, vol. 19, no. 5, pp. 46–56, 2021.

[4] The MITRE Corporation, "CVE-2020-8554 Man in the middle using LoadBalancer or ExternalIPs." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-855

[5] R. B. David *et al.*, "Kubernetes Autoscaling: YoYo Attack Vulnerability and Mitigation," *CLOSER*, vol. 1, 2021.

[6] M. S. I. Shamim *et al.*, "XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices," in *Secure Development (SecDev)*. IEEE, 2020.

[7] D. D'Silva *et al.*, "Building A Zero Trust Architecture Using Kubernetes," in *International Conference for Convergence in Technology (I2CT)*. IEEE, 2021.
[8] Center for Internet Security, "Aqua security." [Online]. Available: https://www.cisecurity.org/partner/aqua-security/
[9] "Helm charts." [Online]. Available: https://helm.sh/
[10] "kube-prometheus-stack." [Online]. Available: https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack
[11] Center for Internet Security, "CIS Benchmark - Securing Kubernetes." [Online]. Available: https://www.cisecurity.org/benchmark/kubernetes/
[12] ——, "CIS Benchmark - Securing Docker." [Online]. Available: https://www.cisecurity.org/benchmark/docker/
[13] Microsoft, "Threat matrix for Kubernetes," 2020. [Online]. Available: https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/
[14] "A Design Analysis of Cloud-based Microservices Architecture at Netflix." [Online]. Available: https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f
[15] Uber, "Introducing Domain-Oriented Microservice Architecture." [Online]. Available: https://eng.uber.com/microservice-architecture/
[16] Aquasec, "Kube-Bench: An Open Source Tool for Running Kubernetes CIS Benchmark Tests." [Online]. Available: https://blog.aquasec.com/announcing-kube-bench-an-open-source-tool-for-running-kubernetes-cis-benchmark-tests
[17] ——, "kube-hunter." [Online]. Available: https://kube-hunter.aquasec.com/
[18] Alcide, "Introducing sKan: Security Hardening and Best Practices for K8s Configuration Files." [Online]. Available: https://www.alcide.io/introducing-skan-security-hardening-and-best-practices-for-k8s-configuration-files/
[19] Bridgecrew by PRISMA CLoud, "checkov." [Online]. Available: https://www.checkov.io/
[20] Alcide, "End-to-End Kubernetes Security Platform." [Online]. Available: https://www.alcide.io/platform/
[21] secure code box, "Kubeaudit." [Online]. Available: https://docs.securecodebox.io/docs/scanners/kubeaudit/
[22] Fairwinds, "Polaris By Fairwinds." [Online]. Available: https://www.fairwinds.com/polaris
[23] Docker, "Docker Bench for Security." [Online]. Available: https://github.com/docker/docker-bench-security
[24] J. Spillner, "Quality Assessment and Improvement of Helm Charts for Kubernetes-Based Cloud Applications," 2019. [Online]. Available: https://arxiv.org/abs/1901.00644
[25] D. B. Bose *et al.*, "Under-reported Security Defects in Kubernetes Manifests," *IEEE/ACM International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, pp. 9–12, 2021.
[26] A. Massoud, "Threat Simulations of Cloud-Native Telecom Applications," Master's thesis, Aalto University, 2021.
[27] K. Jayasinghe *et al.*, "A Survey of Attack Instances of Cryptojacking Targeting Cloud Infrastructure," in *Asia Pacific Information Technology Conference*. ACM, 2020, p. 100–107.
[28] Aqua Security, "Aqua Trivy v0.25.0." [Online]. Available: https://aquasecurity.github.io/trivy/v0.25.0/
[29] Docker, "Vulnerability scanning for docker local images." [Online]. Available: https://docs.docker.com/engine/scan/
[30] Snyk, "Snyk: Develop fast. Stay secure." [Online]. Available: https://snyk.io/
[31] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
[32] J.-R. Jiang *et al.*, "Extending Dijkstra's shortest path algorithm for software defined networking," in *APNOMS*, 2014.