Mitigation of Security Misconfigurations in Kubernetes-based Container Orchestration: A Techno-Educational Approach

by

Md Shazibul Islam Shamim

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 6, 2024

Keywords: security misconfiguration, security practices, kubernetes, model checking, cybersecurity education, authentic learning

Abstract

Kubernetes has emerged as the preferred tool for implementing automated container orchestration, offering significant advantages for IT organizations. However, the presence of security misconfigurations can render Kubernetes-based software deployments vulnerable to security attacks. **The goal of this doctoral dissertation is to help practitioners secure their Kubernetes-based container-orchestration process by adopting a techno-educational approach.** This PhD dissertation advances the science of Kubernetes misconfigurations by conducting three empirical studies. *First*, in order to assist practitioners in enhancing the security of their Kubernetes clusters, a qualitative analysis is conducted on 104 Internet artifacts, including blog posts, resulting in the identification of 11 Kubernetes security best practices. *Second*, to help the practitioners to have a proper understanding of the consequences of security misconfigurations, we build a finite state model based on the interaction of pod life cycle and container states. We extract pod properties by systematically analyzing Internet artifacts related to pod requirements. We use a model checker, NuXmv, to verify the 10 pod security requirements constructed from OWASP's top 10 vulnerabilities and validate 6 sequences of actions that lead to an attack. We extend an open-source tool to construct a new static analysis tool called SLIKUBE+ and identify 23 security misconfigurations. *Finally*, we adopt an educational approach where we construct an authentic learning module for Kubernetes security misconfigurations and a survey to collect feedback. We deploy the authentic learning module into 2 universities in 3 courses in 2 semesters. We evaluate the feedback from the students and observe 73.5% of students find the exercise module to be "Extremely useful" and "Useful". Overall 65.4% of students report that authentic learning is 'Very helpful" and "Helpful" for learning about Kubernetes security misconfigurations.

Acknowledgments

First and foremost, I thank the Almighty Allah. Next, I would like to give my most sincere and deepest gratitude to my supervisor, Dr. Akond Rahman, for his invaluable advice, continuous support, trust and patience during my Ph.D. study. His immense knowledge and ample experience in the area software engineering research have encouraged me all the time in my academic research and daily life. His insightful advice, suggestion tremendously help me to complete my Ph.D. research studies and write this dissertation thesis. I am tremendously grateful to be advised by my committee members, Dr. Drew Springall, Dr. Jakita O. Thomas and Dr. Samuel Mulder who reviewed my proposal documents. I am also grateful to Dr. Mehdi Sadi for serving as the University reader for my PhD dissertation. They gave me a number of valuable suggestions, by which my dissertation substantially improved.

I thank my colleagues from the PASER group at Auburn University for their valuable feedback. I thank Farzana Ahamed Bhuiyan, my former colleague from PASER group for her guidance. I am thankful to graduate writing partner(GWP) Sidharth Suresh Gautam for his valuable suggestion and feedback. I am grateful to all my co-authors whose feedback made my research better. Thanks to the Computer Science and Software Engineering department at Auburn University for funding my Ph.D. research.

Most importantly, none of this could have happened without the support of my family. I would like to thank my mother, Mst Selina Akhter Jahan, who has been a constant source of support and encouragement during my entire academic career. I am grateful to you for your valuable sacrifice. I am thankful to my father for being supportive and always having faith in me. I would like to express my gratitude to my brother, Sakibul Islam Shimul. Last but not the least I am grateful to my wife Marufa Islam for being a consistent source of inspiration for me since our marriage.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Container technologies, such as Docker and LXC are gaining popularity amongst information technology (IT) organizations for deploying software applications. For example, PayPal uses 200,000 containers to manage 700 software applications [49]. For managing these containers at scale, practitioners often use automated container orchestration, i.e, the practice of pragmatically managing the life-cycle of containers with tools, such as Kubernetes [48].

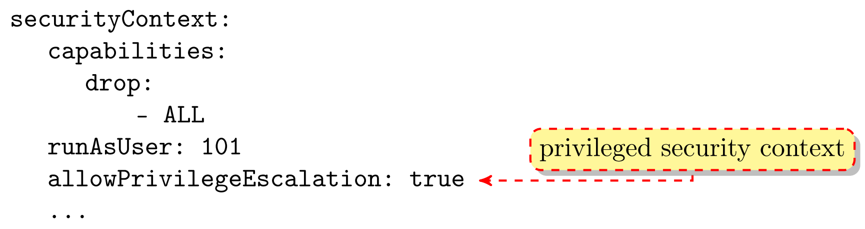
Since its inception in 2014, Kubernetes has established itself as the *de-facto* tool for automated container orchestration [61, 7]. According to Stackrox survey [70], 91% of the surveyed 500 practitioners use Kubernetes for container orchestration. As of Sep 2020, Kubernetes has a market share of 77% amongst all container orchestration tools [73]. Organizations, such as Adidas, Twitter, IBM, U.S. Department of Defense (DOD), and Spotify are currently using Kubernetes for automated container orchestration. Use of Kubernetes has resulted in benefits, e.g., using Kubernetes the U.S. DoD decreased their release time from 3∼8 months to 1 week [15]. In the case of Adidas, the load time for their e-commerce website was reduced by half, and the release frequency increased from once every 4∼6 weeks to 3∼4 times a day [42].

Kubernetes-based container orchestration, similar to every other configurable software, is susceptible to security misconfigurations. However, due to the pervasive nature of Kubernetes-based container orchestration, such misconfigurations can have severe security implications. According to the 2021 'State of Kubernetes Security Report', 94% of 500 practitioners experienced at least one Kubernetes-related security incident, majority of which can be attributed to security misconfigurations [61]. The survey also states Kubernetes-related misconfigurations to "*pose the greatest security concern*" for Kubernetes-based container orchestration [61]. Anecdotal evidence attests to such perceptions: for example, a Kubernetes-related security misconfiguration resulted in a data breach that affected 106 million users of Capital One, a U.S.-based credit card company [39, 75].

Additionally, we observe anecdotal evidence in open-source software (OSS) repositories that provide clues on what security misconfigurations can occur for Kubernetes. For example, in Figure 1.1 we present a code snippet related to Kubernetes manifests, and mined from OSS repositories [19, 71]. In Figure 1.1, we demonstrate a security misconfiguration `allowPrivilegeEscalation:True` in Kubernetes manifest. When a practitioner configures Kubernetes manifests with misconfiguration `allowPrivilegeEscalation:True` in a pod, the misconfiguration allows a child process of a container to gain more privileges than its parent process. As a result, any malicious user can leverage this misconfiguration to gain unauthorized access to the underlying host machine [46] leading the pod to an insecure state. As a Kubernetes practitioner, failure to specify cluster and pod configuration to protect pod isolation boundary can weaken overall security posture of Kubernetes cluster. However, practitioners often lack knowledge needed to mitigate security misconfigurations [46].

All of the evidence mentioned above emphasizes the need for a security analysis of pod and cluster configurations that can weaken security posture of pod at runtime. Such analysis can help practitioners to understand the possible attacks due to their security misconfigurations. The lack of skills among the practitioners demonstrates the importance of designing useful academic educational materials for the next generation of Kubernetes practitioners.

The goal of this doctoral dissertation is to help practitioners secure their Kubernetes-based container-orchestration process by adopting a techno-educational approach.



```
securityContext:
   capabilities:
      drop:
          - ALL
   runAsUser: 101
   allowPrivilegeEscalation: true
   ...
```

privileged security context

Figure 1.1: Anecdotal evidence of security misconfigurations in Kubernetes manifests that shows an example of a security misconfiguration related to privilege escalation in a Kubernetes manifest

This dissertation research will impact the state-of-the-art secure development of Kubernetes. Practitioners may need to be made aware of how a security misconfiguration in Kubernetes can impact the overall security posture of the Kubernetes cluster. We can help the practitioners to integrate best security practices by systematically synthesizing the knowledge related to Kubernetes security best practices. Researchers can leverage our empirical findings and improve the science of secure development of Kubernetes. Moreover, cyber security educators can use the findings from our research and design a curriculum that can help students to learn about Kubernetes security.

In this dissertation, we make the following contributions:

- A systemization of knowledge related to Kubernetes security best practices;
- An empirical analysis of open-source Kubernetes manifests to identify security misconfigurations;
- A formal verification approach of Kubernetes for pod security requirements;
- An empirical validation of the identified attacks from Kubernetes formal verification using security misconfigurations;
- An evaluation of students' perception of authentic learning to design a more effective curriculum for students; and
- An evaluation of the authentic learning module's effectiveness in teaching students Kubernetes security misconfiguration.

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Kubernetes Architecture

Kubernetes is an open-source software for automating management of computerized services such as containers [48]. A Kubernetes installation is colloquially referred to as a Kubernetes cluster [48]. Each Kubernetes cluster contains a set of worker machines defined as nodes. As shown in Figure 2.1, two types of nodes exist for Kubernetes: master nodes and worker nodes.

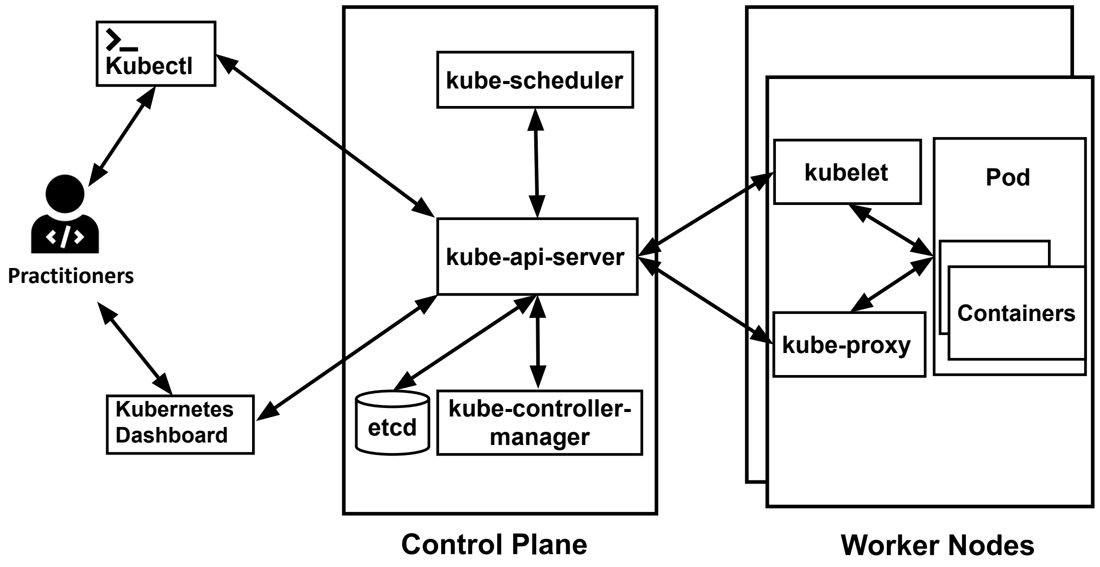


Figure 2.1: A brief overview of Kubernetes. Kubernetes users interact with the installation using the Kubernetes dashboard and 'kubectl'. The purpose of control-plane node is to maintain the desired cluster state and manage worker nodes. Worker nodes are used to run containerized applications inside the pod.

Each control-plane node includes the following components: ‘API server’, ‘scheduler’, ‘controller’, and ‘etcd’ [48]. The ‘API server’ is responsible for orchestrating all the operations within the cluster. Kubernetes serves its functionality through an application program interface from the ‘API server’. The ‘controller’ is a component on the control-plane that watches the state of the cluster through the ‘API server’ and changes the current state towards the desired state. The ‘scheduler’ is the component in the control plane responsible for scheduling pods across multiple nodes. The ‘etcd’ is a key-value based database that stores all configuration information for the Kubernetes cluster. Users use a command-line tool ‘Kubectl’ to communicate with the ‘API server’ in the control-plane node.

The worker nodes host the applications that run on Kubernetes [48]. The following components are included in the worker node: ‘kube-proxy’, ‘kubelet’ and ‘pod’. ‘kube-proxy’ maintains the network rules on nodes. ‘kubelet’ is an agent that ensures containers are running inside a pod. The pod is the smallest Kubernetes entity, which includes at least one active container. A container is a standard software unit that packages the code and associated dependencies to run in any computing environment [48].

2.1.2 Pod Life Cycle

In Kubernetes, the pod is the smallest deployable and manageable unit. Each pod goes through certain phases during their life cycle depending on the condition of the containers inside the pod. The ‘kube-scheduler’ in the control-plane node schedules each pod only once. After scheduling a worker node for the pod, the pod runs in the worker-node until the worker node stops or the pod terminates.

Once an authenticated and authorized user creates a valid pod creation request, the Kubernetes API server accepts the request and stores the information in ‘etcd’ database in control plane node. After Kubernetes API accepts the pod creation request the pod goes to ‘pending’ phase. Kube-scheduler assigns the pod to a node and Kubernetes API server stores that information to ‘etcd’. Finally, the ‘kubelet’ agent in the worker node receives the pod specification, pulls the image from the container registry and provides the image to the container runtime to run the container. If container runtime starts at least one container or in the process of starting or restarting then the pod goes to 'running' phase. When the containers inside the pod terminates and at least one container ends with failure such as terminated by system or exited with non-zero status, the pod goes to 'failed' state. The failed container may restart based on restart-policy upon failure if it is created by other workloads such as replica sets. If the containers in a pod ends in success and will not restart then pod reaches 'succeeded' state. The figure 2.2 demonstrates the pod life cycle as described in this section.

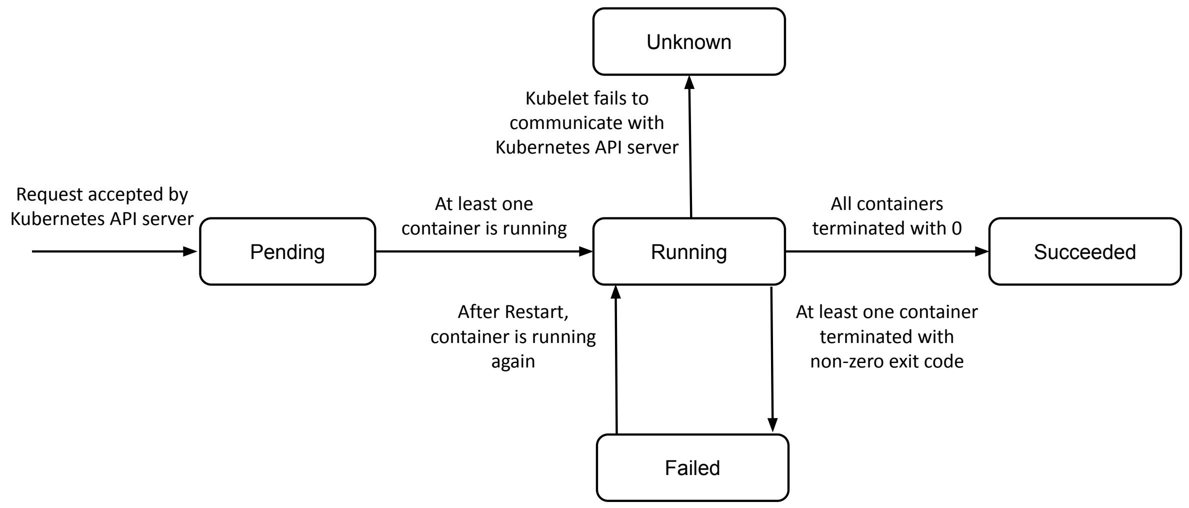

Unknown
Kubelet fails to communicate with Kubernetes API server
Request accepted by Kubernetes API server
Pending
At least one container is running
Running
All containers terminated with 0
Succeeded
After Restart, container is running again
At least one container terminated with non-zero exit code
Failed

Figure 2.2: A simple graphical demonstration of a pod life cycle.

2.2 Related Work

Prior researchers have primarily investigated the usage and maintenance of Kubernetes. Burns et al. [10] described the evolution of container management systems at Google and described how two initial internal systems, Borg and Omega, evolved into Kubernetes. Brewer [8] conducted a case study on Kubernetes and discussed how critical concepts of Kubernetes can be used to simplify scaling of containers. Medel et al. [47] used actual data collected from Kubernetes and applied formal modeling to characterize performance and resource management in Kubernetes. Chang et al. [13] constructed a monitoring platform to dynamically provision cloud resources using Kubernetes. Vayghan et al. [1] investigated the availability of Kubernetes using a set of experiments and reported that service outages can occur frequently. Shah and Dubaria [67] compared orchestration management features of Docker Swarm, Kubernetes, and Google Cloud Platform and observed that Kubernetes provide features such as deployment, monitoring, and easy scalability. Takahashi et al. [74] proposed a portable load balancer for Kubernetes and reported improved portability without sacrificing performance. Song et al. [69] used Kubernetes to construct an auto-scaling system for API gateways. The authors [69] report that their constructed system improves the utilization of system resources while ensuring high availability. Muralidharan et al. [50] constructed a Kubernetes-based system to monitor and manage Internet of Things (IoT) applications for smart cities. Wei-guo et al. [79] constructed a resource scheduling algorithm for Kubernetes using ant colony and particle swarm optimization techniques. The scheduling algorithm proposed by Wei-guo et al. [79] outperforms the original algorithm used in Kubernetes. Overall, we observe that most publications related to Kubernetes are in the area of 'performance evaluation,' 'scheduling and resource allocation', and 'Internet of things (IoT)'.

However, we observe a few security-related research in Kubernetes. Security publications investigate techniques to mitigate security weaknesses for Kubernetes. Anomaly detection is one security-related topic that researchers have addressed. Hariri et al. propose an anomaly detection tool for detecting anomalies in astronomy data analysis tools deployed with Kubernetes [28]. Tien et al. [76] use neural network approaches to implement 'KubeAnomaly,' a tool for anomaly detection in the Kubernetes cluster. Security-focused frameworks have also gained interest: Bila et al. [6] propose an automated threat mitigation architecture for Kubernetes that continuously scan containers for vulnerabilities to quarantine and isolate vulnerable containers. Ahmedvand et al. [2] built a security framework for integrity protection for microservices-based systems. Surantha et al. [72] propose a zero-trust secure design for a Kubernetes-based data center. Zero-trust refers to the concept that requires all users to be authenticated, authorized, and continuously validated before being granted or keeping access to software and data [21].

Although the researchers have addressed a few challenges related to Kubernetes security, none addressed misconfiguration-related security concerns in Kubernetes. We are addressing this research gap in the doctoral research to help practitioners develop secure Kubernetes manifests.

Chapter 3

Systemization of Kubernetes Security-related knowledge

In this chapter, we describe our research study to systematize Kubernetes-related security knowledge by synthesizing Kubernetes security best practices.

3.1 Kubernetes-related Security Best Practices

Systematizing available knowledge regarding Kubernetes security practices could support practitioners in securing their Kubernetes installations. In addition, such systematization of knowledge can be beneficial for practitioners who (i) want to understand what activities need to be executed to secure Kubernetes components and (ii) can use the derived list of practices as a benchmark to compare their state of security practices. Systematization of knowledge can be conducted by analyzing Internet artifacts, such as blog posts and video presentations. Practitioners often report what practices they use in Internet artifacts [24, 26] rather than in academic forums such as conferences. In this study, we synthesize Kubernetes security practices by conducting a grey literature review [29]. A grey literature review is the process of reviewing and synthesizing content included in Internet artifacts, such as blog posts and video presentations [29]. A grey literature review differs from a systematic mapping study or systematic literature review, as in these types of literature reviews, researchers use peer-reviewed scientific articles indexed in scholarly databases. In prior work, researchers have reported that practitioners use Internet artifacts, such as blog posts to report their experiences, recommendations, and the practices they follow. Previously, researchers have systematically studied Internet artifacts to identify challenges in microservices development, identify practices used in continuous deployment [59], identify security practices used in organization who have adopted DevOps [78], and software testing [22].

3.2 Methodology

The methodology of this study is demonstrated in Figure 3.1. We use the Google search engine to collect our Internet artifacts. We use three search strings: ‘kubernetes security practices’, ‘kubernetes security good practices’, and ‘kubernetes security best practices’. After performing the search, we collect the first 100 search results, as Google displays the results in a sorted order based on relevance. We apply inclusion criteria on the collected search results to identify Internet artifacts that discuss security practices for Kubernetes. The inclusion criteria are listed below:

- The Internet artifact is not a duplicate;
- The Internet artifact is available for reading; and
- The Internet artifact discusses security practices for Kubernetes;

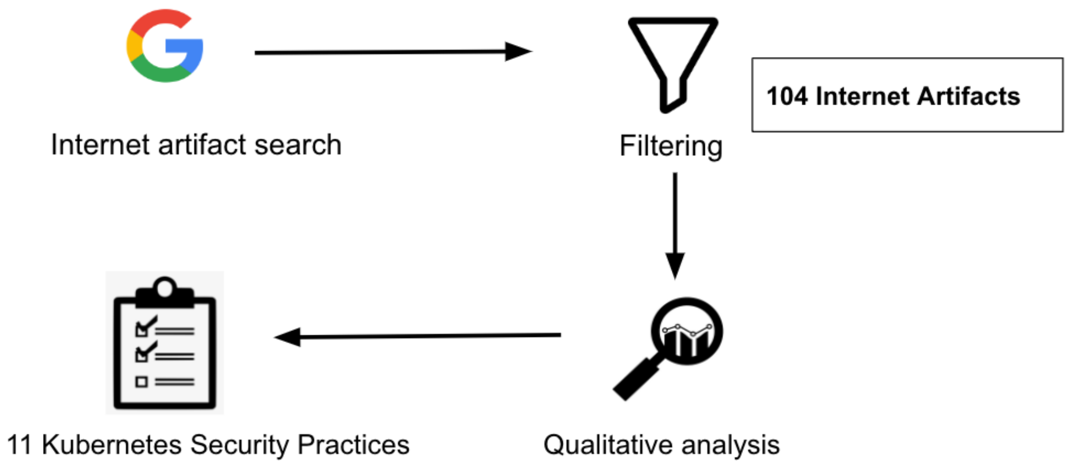
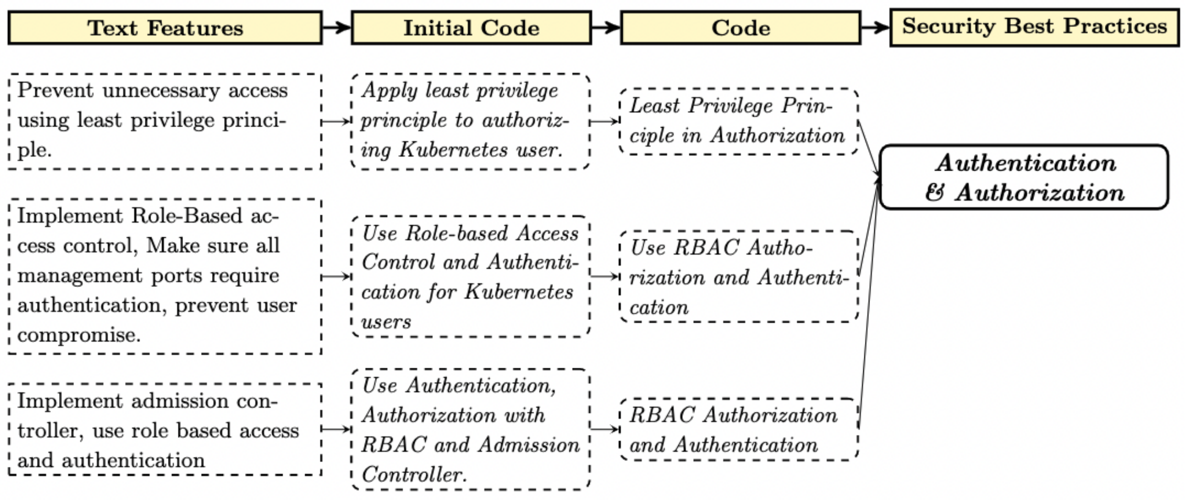


Figure 3.1: A brief overview of the methodology to derive security best practices related to Kubernetes from Internet artifacts.

We use open coding [64], a qualitative analysis technique, to determine the security practices for Kubernetes. In open coding, a rater observes and synthesizes patterns within unstructured text [64]. Figure 3.2 shows an example of open-coding to derive a category of security best practices in Kubernetes. We mitigate this bias by allocating another rater, the second author of the paper, who apply closed coding [17] on a randomly selected set of 50 Internet artifacts. Closed coding is the technique of mapping an entry to a predefined category [17].



Text Features
Initial Code
Code
Security Best Practices
Prevent unnecessary access using least privilege principle.
Apply least privilege principle to authorizing Kubernetes user.
Least Privilege Principle in Authorization
Authentication & Authorization
Implement Role-Based access control, Make sure all management ports require authentication, prevent user compromise.
Use Role-based Access Control and Authentication for Kubernetes users
Use RBAC Authorization and Authentication
Implement admission controller, use role based access and authentication
Use Authentication, Authorization with RBAC and Admission Controller.
RBAC Authorization and Authentication

Figure 3.2: An example of open-coding to derive a category of security best practices in Kubernetes

3.3 Results

After applying open coding on 104 Internet artifacts we derive 11 practices for Kubernetes security. Of the 104 Internet artifacts 90.38%, 4.81%, and 4.81% are respectively blog posts, videos and presentations. Among the 11 Kubernetes security best practices mostly discuss about ensuring Authentication and Authorization and Kubernetes-specific policies. In figure 3.3, we have listed 11 security best practices and their occurrences in the curated 104 Internet artifacts. We describe the 11 identified Kubernetes security best practices as follows where the number between parentheses indicates their occurences in the Internet artifacts:

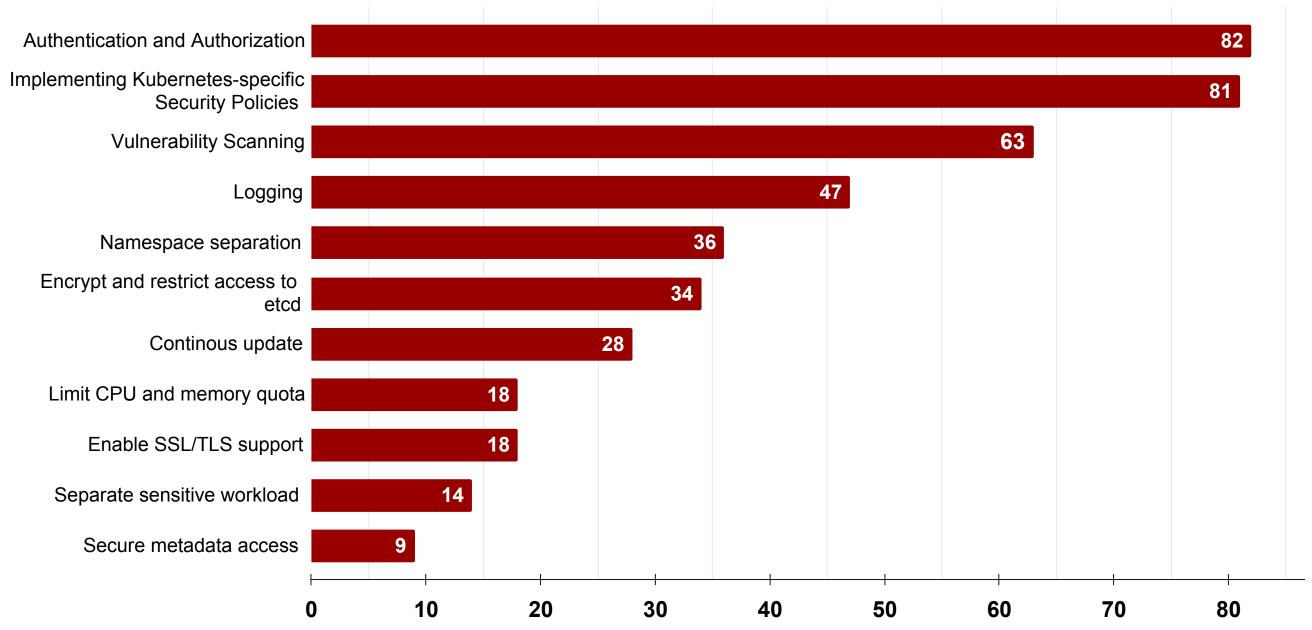


Figure 3.3: The occurrences of security best practices in Kubernetes in the Internet Artifacts

1. **Authentication and Authorization (82)**: The practice of applying authentication and authorization rules to prevent malicious users from getting access and performing unauthorized activities inside the Kubernetes cluster. Authentication in Kubernetes refers to the authentication of API requests through authentication plugins[41]. Authorization in Kubernetes refers to the evaluation of each authenticated API request against all policies to allow or deny the request[41].

2. **Implementing Kubernetes-specific Security Policies (81):** The practice of applying policies to secure Kubernetes components, pods, and networks of Kubernetes clusters to prevent security breaches.

- *Network-specific policies*: The practice of applying a network policy to protect communication between Kubernetes pods from undesirable network communications. By default, all Kubernetes pods can communicate with other pods. Practitioners recommend policies to restrict traffic between pods, restrict API server access and reducing network exposure to secure the network.

- *Pod-specific policies*: The practice of implementing a policy for pods to apply security context to pods and containers. Pod policies determine how the workloads should run in the Kubernetes cluster. Without defining a secure context for the pod, a container may run with root privilege and write permission into the root file system, which can make the Kubernetes cluster vulnerable. Practitioners recommend containers inside a pod must run as a non-root user with read-only permission and enabling Linux security modules.

- *Generic policies*: The practice of applying a generic security policy to protect Kubernetes cluster components from external malicious users. TCP ports for kubelet, API server, etcd, and network plugins should not be left open and should require authentication to have visibility. Every user in the system should have the least privilege by default.

3. **Vulnerability scanning (63):** The practice of scanning Kubernetes components and continuous delivery (CD) components for vulnerabilities.

- Kubernetes components, such as containers can contain vulnerabilities and malicious malware. If vulnerabilities are present in a Kubernetes cluster, then the entire container orchestration system, and the provisioned applications, become susceptible to attacks. For example, in 2017, researchers found Docker images embedded with malicious malware. Practitioners recommended scanning containers for vulnerabilities with tools,such as 'Dockscan' [1] and 'CoreOS Clair' [2].

- If images and deployment configurations within CD components are not inspected, then it can make the Kubernetes cluster vulnerable to malicious users. The malicious users can gain access at a later point when these images are deployed and may exploit

[1] https://github.com/kost/dockscan
[2] https://github.com/quay/clair

the latent vulnerabilities in Kubernetes production environments. Practitioners recommend pulling images from a trusted private registry and checking for the vulnerability of code and images.

4. **Logging (47):** The practice of enabling and monitoring logs for the Kubernetes cluster. Practitioners recommend that logging should be enabled for (i) applications, (ii) the containers within each pod, and for (iii) Kubernetes clusters for system health checking.

5. **Namespace separation (36):** The practice of separating namespaces so that the resource of one namespace are not shared with another. A 'namespace' in Kubernetes is a logically isolated virtual cluster within the same physical cluster.[41] Creation of separate namespaces enables resources to be isolated between namespaces. If a separate namespace is not created for a resource then the resource gets 'default' namespace.

6. **Encrypt and restrict access to etcd (34):** The practice of encrypting and restricting access to 'etcd', the internal database used by Kubernetes[41]. By default, Kubernetes stores secret data as plaintext in 'etcd'[3]. Practitioners recommend using secret management tools for additional security[41], such as 'Vault'[4] for encryption.

7. **Continuous update (28):** The practice of applying security patches to keep the Kubernetes cluster updated with latest security fixes. Practitioners recommend that Kubernetes users apply updates as well as conducting continuous updates for the deployed applications within the Kubernetes pods.

8. **Limit CPU and memory quota (18):** The practice of limiting CPU and memory to a pod or a namespace so that malicious attacks can be mitigated. By default, all resources in Kubernetes start with unbounded memory requests/limits and unbounded CPU access.

9. **Enable SSL/TLS support (18):** The practice of enabling secure sockets layer (SSL) or transport layer security (TLS) protocol to ensure secure and encrypted communication between Kubernetes components. Enabling TLS between kubernetes api server, etcd,

[3]https://ubuntu.com/kubernetes/docs/encryption-at-rest
[4]https://www.vaultproject.io

kubelet and kubectl ensures secure communication between cluster components. Practitioners suggest enabling TLS and SSL certificates for Kubernetes components.

10. **Separate sensitive workload (14):** The practice of running sensitive applications on a dedicated set of machines to limit the potential impact of a security breach.

11. **Secure metadata access (9):** The practice of securing the sensitive metadata of the Kubernetes cluster. Practitioners state that the Kubernetes metadata APIs provide a gateway to expose 'kubelet' admin credentials.

Chapter 4

Motivating Example

We motivate our empirical study further by using Figure 4.1, where we present an example Kubernetes manifest. The manifest is used to specify configurations for a pod called 'sample' with 'nginx' container images, using the namespace 'sample-app-space'. We also observe the manifest to include specifications for role-based access control (RBAC) using `kind: RoleBinding`, `kind: ServiceAccount`, and `kind: Role` objects. In the case of configurations, such as `name` and `namespace`, a practitioner can assign any strings so that the pod 'sample' is deployed with adequate RBAC configurations, However, prior to execution, in the case of nine configurations, as indicated with the green circles, the practitioner must determine if one or a combination of these configuration values can yield security attacks. Of these nine configurations, (i) 5 are Boolean, (ii) 2 are of type Integer, each yielding 2^{32} possible values, (iii) one configuration with 3 values, and (iv) 1 configuration with 7 possible strings. To determine if these nine configurations cause attacks, a Kubernetes user can manually explore all possible combinations for the 9 configurations by accounting for the semantics of pods, RBAC policies, and their interactions. However, such manual exploration is practically impossible as the user has to provision the pod for 1.2×10^{22} possible configuration combinations. Hence, an automated approach is required that can aid in automated determination of what pod-related configurations can cause security attacks. As the focus is on identifying configurations that can cause pod-related attacks, the automated approach should also account for pod states, i.e., the states that a pod traverse upon execution. In the context of Figure 4.1, prior to executing the pod with the provided configurations, a pod will undergo the through following states: 'request initiated', 'request authenticated', and ‘request authorized’ [41]. Therefore, the automated approach must account for these states unique to pods to determine attack-akin configurations.

To that end, we use model checking to determine attack-akin configurations. Model checking leverages finite state machines, which will allow us to account for the pod-related states [14]. Our hypothesis is that use of model checking will be useful to determine: (i) if the 9 configuration combinations can lead to a security attack, and (ii) what configuration values can be used to demonstrate the attacks. We describe our model checking-based approach and findings in Section 5.

```
kind: Pod
metadata:
  name: sample
  namespace: sample-app-space
spec:
  securityContext:
     runAsGroup: 3000   1
     fsGroup: 2000   2
     readOnlyRootFilesystem: false   3
     runAsNonRoot: false   4
  containers:
  - image: nginx
     name: kubectl
     hostIPC: false   5
     hostNetwork: true   6
     hostPID: false   7
...
kind: Role
metadata:
  name: sample-app-role
  namespace: sample-app-space
rules:
  - apiGroups:   8
        - batch
        - extensions
        - policy
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]   9
...
kind: ServiceAccount
metadata:
  name: app-service-account
  namespace: sample-app-space
...
kind: RoleBinding
metadata:
  name: app-rolebinding
  namespace: sample-app-space
roleRef:
  kind: Role
  name: app-role
subjects:
- namespace: sample-app-space
  kind: ServiceAccount
  name: app-service-account
```

Figure 4.1: Example code snippet to demonstrate attack-akin configurations.

Chapter 5

Identification of Attacks for misconfigurations in Kubernetes manifests

Despite the reported benefits, security is identified as one of the prime concerns for practitioners who utilize containers for the construction and deployment of applications in Kubernetes. According to the State of Kubernetes and Container Security Report 2020, published by Stackrox, it is suggested that containerized application deployment was delayed by 44% of organizations due to security concerns. Furthermore, it is stated in the report that security incidents were experienced by 94% of organizations in the last 12 months, with 69% of these incidents being attributed to misconfiguration-related security issues.[60].

To identify security misconfigurations in source code, researchers use static analysis tools to detect code smells, such as in Infrastructure as Code (IaC) scripts [55], [56] and misconfiguration in Kubernetes manifests [57]. However, the precision required for evaluating source code soundness and completeness cannot be achieved by static analysis tools alone [37]. The generation of potential attack scenarios or the demonstration of a realizable attack path when identifying a security misconfiguration in a source code file is not within the capabilities of static analysis tools. Researchers use model checking to identify prior attacks and uncover new attacks in cellular network protocols [36], [37]. In this research, we combine model checking with static analysis tool to identify known attacks for security misconfigurations in Kubernetes manifests.

We answer the following questions:

- RQ 2.1 What are the configurations that can lead a pod to an unsafe states?
- RQ 2.2 How frequently do the configurations appear in OSS repositories?

5.1 Methodology for RQ 2.1

5.1.1 Threat Model

In our threat model, we assume that Kubernetes manifests are developed by system administrators without malicious intent, but still can include security misconfigurations. Our assumption is consistent with prior research that shows the existence of known security misconfigurations [57]. In our threat model, a malicious user, i.e., an attacker attempts to launch attacks against Kubernetes-based containers by leveraging one or multiple combinations of these misconfigurations. The goal of the attacker is to (a) gain unauthorized access and/or (b) disrupt availability for any Kubernetes-based container infrastructure. If successful, the attacker may also perform other pernicious attacks including crypto-mining attacks and stealing intellectual property. Disruption in availability can cause large-scale outages for end-users.

5.1.2 Derivation of Pod Properties

The phase of a pod during its life cycle is influenced by cluster configurations and environment variables, as described in Section 2.1.2. To replicate the behavior of a pod throughout its life cycle, we develop a finite state machine using NuXMV. In our model in NuXMV, we incorporate the Kubernetes cluster configurations and environment variables as pod properties, which define the pod's behavior. We use open coding to identify codes for pod properties related to pod security requirements from the Internet artifacts. Open coding is a qualitative analysis technique that identifies the underlying code from unstructured text data [65].

Pod Security Guideline Extraction from Internet Artifacts

We conduct a grey literature review [23] on available Internet artifacts that discuss the pod security requirements. We use the Google search engine to collect the internet artifacts in incognito mode. We use two search strings: “Kubernetes pod security guidelines” and “Kubernetes pod security rules”. We start our Internet artifact search with the initial search string “Kubernetes pod security guidelines”. We add the later search string as we observe the practitioners often refer pod security rules instead of pod security guidelines. Then, we search with the search string “Kubernetes pod security rules”. We collect the first 100 search results for each of the search strings. To perform the filtering for the Internet artifacts, we apply the following exclusion and inclusion criteria according to the guideline of Garousi et al. [23].

Exclusion Criteria: We adhere following guideline to exclude an Internet artifact.

- The Internet artifact is not written in English.
- The Internet artifact is published before 2014, as the initial version of Kubernetes was released in 2016. [48].

Inclusion Criteria: We apply the following criteria to include an Internet artifact.

- The Internet Artifact is available for reading.
- The Internet artifact is not a duplicate.
- The content of the Internet artifact explicitly describes the Kubernetes security guideline that includes pod related security guidelines.

After combining two search results for our search strings, we remove the duplicates. We read each of the Internet artifact, and filter 21 Internet artifacts to gather pod properties related to pod security requirements. Figure 5.1 illustrates our Internet artifact collection process.

We apply an open coding technique to derive pod properties related pod security requirements from the Internet artifacts. In Figure 5.2, we illustrated our open coding process. First, we collect the text from the Internet artifacts that discuss pod-related security and form initial code. In Figure 5.2, we create initial code *“Admission controller*

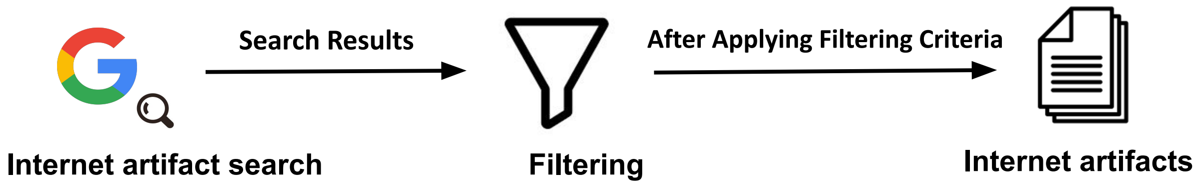


Figure 5.1: Internet artifact search and filtering process

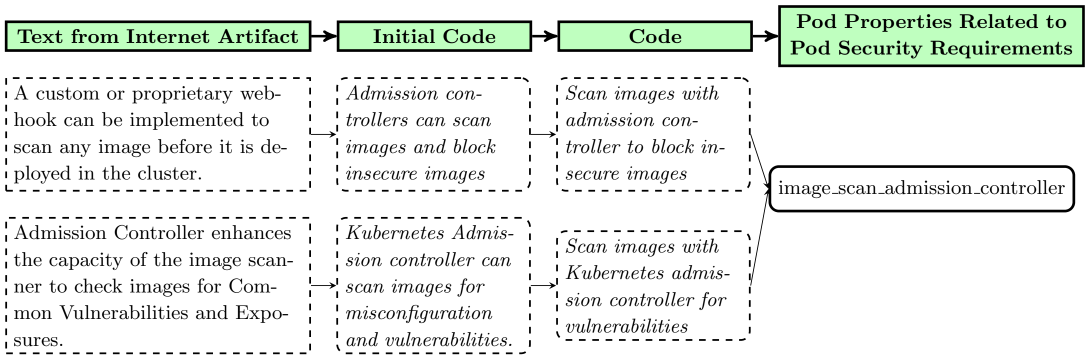

Text from Internet Artifact
Initial Code
Code
Pod Properties Related to Pod Security Requirements
A custom or proprietary web-hook can be implemented to scan any image before it is deployed in the cluster.
Admission controllers can scan images and block insecure images
Scan images with admission controller to block insecure images
image_scan_admission_controller
Admission Controller enhances the capacity of the image scanner to check images for Common Vulnerabilities and Exposures.
Kubernetes Admission controller can scan images for misconfiguration and vulnerabilities.
Scan images with Kubernetes admission controller for vulnerabilities

Figure 5.2: A description of our open coding process to derive pod properties related parameters from Internet artifacts.

can scan images and block insecure images" and *" Admission controller can scan images for misconfiguration and vulnerabilities."* respectively. In the next step, we identify codes *"Scan images with Kubernetes admission controller to block insecure images"* and *"Scan images with Kubernetes admission controller for vulnerabilities"* from the initial code. Finally, we construct the pod properties as parameters related to pod security requirement as a `boolean` variable, such as `image_scan_admission_controller`. If the value of the `image_scan_admission_controller` is `true`, then the admission controller for scanning container image is present in the Kubernetes cluster. Altogether, we derive 71 pod security related properties from the Internet artifacts. We represent our derived pod properties with their appearance frequency in Table 5.1.

Pod Security Requirement Construction

We define the pod security requirements by examining the OWASP Top 10 Kubernetes vulnerabilities that directly relates to pod. OWASP Top 10 is a non-profit open-source project that raises awareness about application security by prioritizing critical security risks of an organization [32]. Organizations such as Defense Information Systems Agency (DISA) and standards such as PCI DSS refer to OWASP Top 10 for their security assessment [80]. OWASP Top 10 in Kubernetes helps practitioners prioritize the list of security risks by identifying the most critical risks in the Kubernetes cluster [32] [80]. To construct the pod security requirement, we translate the pod property to corresponding NuXMV LTL formulas for pod security requirements. Table 5.2 describes our pod security requirements and corresponding NuXMV LTL formulas. We use the parameters related to pod security requirements gathered from the Internet artifacts to construct the NuXMV LTL formulas for the pod security requirements. For instance, in the first row of the Table 5.2 we describe pod security requirement, "*Any container running inside a pod with unnecessary privilege will not be in an unsafe state at runtime.*". We construct corresponding LTL formula in NuXMV such as if the container has `CAP_SYS_ADMIN` privilege then there will not be a `over_privileged_container` while the `pod_state = pod_running`.

We translate the pod security requirements to the NuXMV LTL property as follows:

```
 LTLSPEC G (!TC_18 & !TC_19 & new_pod_creation_request) -> G ((CAP_SYS_ADMIN)
-> G X(!over_privileged_container & pod_state=pod_running))
```

In the propositional LTL formula, we use `TC_18` and `TC_19` & `new_pod_creation_request` as constants. Here `TC_18` and `TC_19` are transition conditions and both are set to `False` and state variable `new_pod_creation_request` is set to `True`.

5.1.3 Translation of Pod Life Cycle to Finite State Transitions

A pod is considered as the unit entity in the Kubernetes-based container orchestration. In Kubernetes, each pod has a definite life cycle as described in Section 2.1 in Figure 2.1.2.

When the Kubernetes API server accepts pod creation requests, it creates a pod object, and the pod goes to the `Pending` phase. The scheduler in the Kubernetes API server schedules the pod object to a node. In the first step, the scheduler finds a set of candidate nodes and assigns ranks to the candidate nodes to find the most suitable node for the pod object. In the second step, the scheduler binds the feasible node for a pod object. A pod remains in the `Pending` phase until the kubelet in the assigned node receives the pod object specification and provides the container runtime engine with the image to start a container. The pod has its IP address in the `Running` phase and can communicate with all other pods on the node or any other node in the Kubernetes cluster. The pod can be accessed outside the Kubernetes cluster as a service with a service IP address managed by kube-proxy. Each pod is assigned storage while in the `Running` phase, and the Kubernetes volume abstracts the storage of a pod. Kubernetes destroys the ephemeral volume of a pod when a pod terminates. However, a pod can have a persistent volume that exists beyond the life cycle of a pod. If at least one container terminates with a non-zero exit code, the pod goes to `Failed` phase. If the container runs again after the restart, the pod goes to the `Running` phase. If the kubelet fails to communicate with the Kubernetes API server from the node, then the pod goes to `Unknown` phase. The pod controller replaces the pod in the node in case of pod failure or another node in case of node failure. If all the container terminates with zero exit code the pod terminates in `Succeeded` phase.

When a practitioner sends a request to the Kubernetes API server to create a pod, we observe a temporal ordering of events for a pod when there is a transition of the pod phase. Each event depends upon pod configurations, conditions and Kubernetes cluster states. In the sequence diagram in Figure 5.3, we illustrate only the sequence of events at the pod creation time. In section 2.1, we discuss the events while a pod reaches its `Running` phase from the `Pending` phase. We list the temporal events for the scenario as follows:

Event 1: The practitioner initiates a pod creation request to the API server.

Event 2: The API server authenticates the request. This event can lead to two events: successful or failed authentication.

Event 3: Upon successful authentication, the API server can authorize or fail to authorize the request.

Event 4: The API server writes the information to the etcd database and returns a response to the practitioner.

Event 5: Upon successful authentication, and authorization, the Kubernetes API server creates a pod object and sends the pod specification to Scheduler, which watches for a new pod.

Event 6: Upon successful node allocation to the pod, the Kubernetes API server binds the pod to a node and stores the desired pod state in the etcd database.

Event 7: The kubelet agent in the worker node watches for the pod bound to it.

Event 8: The API server sends the pod specification to the kubelet worker node.

Event 9: Upon receiving the pod specification, the kubelet attempts to pull the container image from the registry. If the kubelet in the worker node can pull the image from the registry, it sends the image to a container runtime such as Docker engine. Upon failure to pull the image from the registry, kubelet reports an error to the Kubernetes API server.

Event 10: If the container runtime, such as the Docker engine, can create a container from the image or encounter any issue, the kubelet updates the API server regarding the pod update.

5.1.4 Encoding Logic Formula for Finite State Transitions and Requirements

Mitigation of Challenges

We address two primary challenges while encoding a finite state model and constructing logic formula for the pod life cycle. We organize our challenges as below:

Challenge #1: Search-space explosion: Kubernetes pod-related events depend upon the configurations of the pod and the Kubernetes cluster. Each pod event in Kubernetes

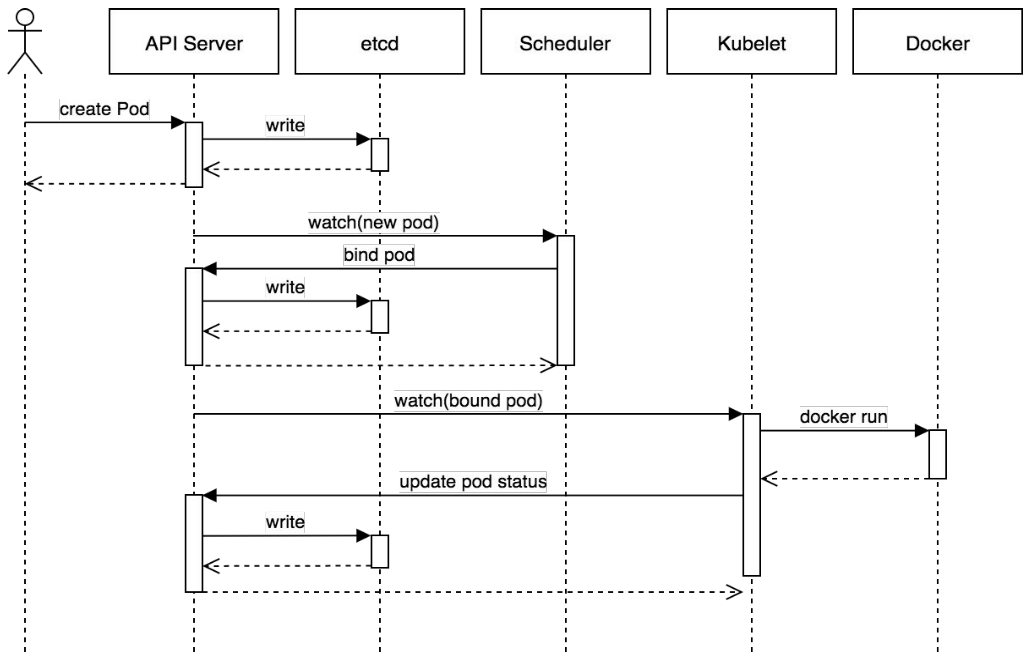


Figure 5.3: The sequence diagram of pod events after a practitioner requests for pod deployment

occurs due to a specific combination of configurations. In the Kubernetes cluster, the number of combinations of configurations of the pod and the Kubernetes cluster makes the search space computationally expensive. We construct 125 state variables to abstract pod and Kubernetes cluster configurations. Each state variable is a configuration in Kubernetes cluster that helps in describing the pod behaviour a pod state. Among 125 state variables, 71 of the state variables are the pod properties related to pod security requirements as described in Table 5.1. The remaining 54 state variables helps describe pod behaviour in Kubernetes cluster. In total, our FSM for a pod has 33 states, 125 state variables, and 43 transition conditions. The search space of our FSM for a pod has 2^{125} search space. Hence, verifying the pod security requirements as a propositional formula built from a set of 125 state variables relates to boolean satisfiability problem (SAT) [16]. The SAT problem is an NP-complete problem. NP-complete problems can not be solved in polynomial times but can be verified in polynomial time. To verify SAT problem, the SAT solver is used as a verification tool. The SAT solver uses approximation algorithms to verify a boolean formula in polynomial time. We use NuXMV model checker that uses SAT solver to reduce the search space of our FSM for a pod to verify pod security requirements in polynomial time.

Challenge #2 Intertwined component interactions: Identifying the pod events due to the complex interaction between the Kubernetes components and pods is one of our primary challenges in building a finite state model (FSM) for a pod. Each pod phase in the pod life cycle depends on the container state, Kubernetes components, configurations and pod conditions. We mitigate this challenge by identifying the transition conditions between the FSM states for a pod as a form of propositional logic.

Construction of Finite State Machine

We model the events of the pod life cycle as a deterministic finite state machine. Our state machine is 3-tuple (S, Σ, Γ) where S is the finite nonempty set of states, Σ is a finite nonempty set of transitions and Γ is a finite nonempty set of transition conditions. We define $s_i \epsilon S$ is the initial state of the pod,and $s_o \epsilon S$ is the final state of the pod. transition action $\alpha \epsilon \Sigma$ is a finite set of transitions and transition conditions $\gamma \epsilon \Gamma$ is a set of transition condition.

We construct the FSM model for a pod and transition condition from one state to another state with the combination of state variables defined as propositional logic formula. Model checking is a method to check if a system's finite state model (FSM) fulfils a particular set of specifications [5]. Model checking method explores all possible states of a system and all possible values for the state variables [5]. We define a pod state as the status of a pod in the Kubernetes cluster. In addition, we define state variables as configurations that can describe a pod state. A transition condition is an expression where a combination of state variables allows the transition from a state to a subsequent state. A valid transition triggers a pod-related event to transition into a subsequent pod state. For instance, when the Kubernetes API server starts the pod creation request, the pod enters into `request_pod_creation_initiated` state. If the pod stays in the

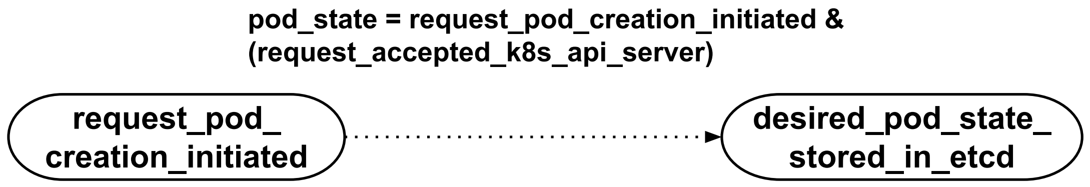

pod_state = request_pod_creation_initiated & (request_accepted_k8s_api_server)
request_pod_ creation_initiated
desired_pod_state_ stored_in_etcd

Figure 5.4: A pod state transition of event

`request_pod_creation_initiated` state and the state variable `request_accepted_k8s_api_server`, then the transition condition for the subsequent state `desired_pod_state_stored_in_etcd` will be `true`. In this case, a valid transition and pod event will occur, as Figure 5.4 describes. If a state in the FSM is found under the property or specification that violates the property then the model generates a counter-example. A counter-example describes the execution step from the initial step to where the system violates the specific property. We use 71 pod properties related to pod security requirements in our pod state model as described in Table 5.1. Apart from these 71 properties related to pod security requirements, we use 54 additional state variables as a parameter to define the transitions and transition conditions of our FSM for a pod. We grouped the 71 pod properties related to pod security requirements into 16 groups.

Each parameters related to pod security requirement can belong to multiple groups. The intuition behind grouping the parameters related to pod security requirements is to cluster them into a similar group so that we can construct propositional logic formula to verify the pod security requirements in the NuXMV [12]. For instance, `hostPID_enabled`, `hostIPC_enabled`, `CAP_SYS_ADMIN`, `host_path_enabled`, `host_port_enabled`, `hostprocess_enabled`, and `hostNetwork_enabled` belongs to one single group `host_namespace_access`, because any of the two parameters can be used to access host namespace. We define the relationships as `hostPID_enabled | hostIPC_enabled |hostprocess_enabled| hostNetwork_enabled | CAP_SYS_ADMIN | host_path_enabled | host_port_enabled` If a container can access the shared namespace, it can potentially extract underlying host information such as host process id, host network, and even host file system. Similarly, all of the variables also belongs to `over_privileged_container`. The privilege to access host namespace gives the container unnecessary privilege to compromise the underlying host.

In the NuXMV, we verify the pod security requirements in our FSM for a pod as a safety property [5]. A safety property dictates that under certain condition bad events will never happen [5]. For model checking, we use counter example guided abstraction refinement (CEGAR) principle [5]. According to the CEGAR principle, we initially use the pod security requirements and our FSM for verification in the NuXMV [12]. If the NuXMV generates erroneous counter-example then we revise our pod security requirements to prevent the erroneous counter-examples. We continue this process for violation of each safety property until we find understandable counter-example. Depending on counter-example output, we set the values of some state variables, transition conditions as constant to prevent erroneous counterexamples. The value of the state variables and transition conditions we set as constant in LTL formula do not change during the execution for verification in NuXMV. For instance, in the following NuXMV LTL formula as described in Table 5.2, we use `TC_18` and `TC_19` & `new_pod_creation_request` as constants. Here `TC_18` and `TC_19` are transition conditions and both are set to `False`.

```
    LTLSPEC G (!TC_18 & !TC_19 & new_pod_creation_request) -> G ((CAP_SYS_ADMIN)
-> G X(!over_privileged_container & pod_state=pod_running))
```

5.1.5 Mapping of Pod Events to Finite State Machines

Kubernetes provides support for practitioners to manage containerized applications at scale [41] [11]. Practitioners can install Kubernetes on-premise, on cloud platforms, or a combination of both. A Kubernetes installation is also colloquially referred to as a Kubernetes cluster [41]. A pod is the most fundamental unit of a Kubernetes cluster [41]. A pod groups one or more containers with shared network and storage resources according to the specifications practitioners provide in their Kubernetes manifest. Kubernetes allows namesapces, which provide a mechanism to isolate groups of resources within a Kubernetes cluster.

Using Figure 4.1, we also provide background information on pod lifecycle and states, as our model checking-based approach to answer RQ1 accounts for pod-related states. The lifecycle of a pod consists of five phases. Each phase consists of one or multiple states, which we encode as described in Section 2.1.2. Transition from one state to another is dependent Boolean conditions, which we refer to as transition conditions.

The first phase of a pod in Kubernetes is 'Pending' with the 'request_initiated' and 'request_pod_creation_initiated' states. The 'kubectl' command line interface is used to create a pod. Next, with two states namely, 'request_authenticated' and 'request_authorized', the Kubernetes API server authenticates and authorizes the request upon receiving the request. If the requests are validated by the admission controller with the 'request_admission_controller_validated' state, then a pod is created and all the configurations for the pod will be stored in etcd with the 'desired_pod_state_stored_in_etcd', 'kubelet_receives_podspec', and 'etcd_updated' states. 'etcd' is a database that uses a key-value mechanism to store all pod-related data [41]. In the context of Figure 4.1, at this phase all configurations for 'sample' will be stored in etcd. Using the 'pod_schedule_bind_phase' state, the Kubernetes scheduler schedules the pod, and the container runtime engine pulls the container image for running the container using the 'image_pulled_from_registry' state. For example, for Figure 4.1, the container runtime engine will pull the 'nginx' image. A container runtime engine is the software that run containers in a host machine [41]. Kubernetes scheduler is a component of the control plane node responsible for selecting and binding a node for a newly created pod [41]. Upon completion, the pod will reach 'pod_starting' state. When the Kubernetes scheduler binds the pod to a worker node, and at least one container has started, the pod reaches the 'Running' phase. A pod in the 'Running' phase can communicate with all other pods on the node or any other node in the Kubernetes cluster. As part of the 'Running' phase the following states are executed:

'image_provided_to_cri', 'volume_mounted', 'host_network_access', 'host_system_access', and 'pod_running'.

A container inside a pod can terminate after completing its task or terminate at runtime. If at least one container terminates in failure at runtime, such as terminated by the system, then the pod reaches the 'Failed' phase via the 'pod_failed' state. Upon termination, a container can be restarted based on the container restart policy. A pod reaches the 'Succeeded' phase if all its containers terminate after successful execution via the 'pod_succeeded' state. If the 'kubelet' component of the worker node where the pod is running fails to communicate with the Kubernetes API server in the control plane node, the pod reaches 'Unknown' phase. For both succeeded and failed phases, the 'pod_terminate' state is executed.

Figure 5.5 summarizes the states that a pod encounters. For example, 'REQ_INI' represents the 'request_initiated' state, which transitions to the next state called 'request_authentication' ('REQ_AUTH') if the transition condition T_1 is true. Here, T_1 corresponds to the condition of (Valid_Kubeconfig $\vee$ Auth_Bootstrap_Token)), which means either the pod has a valid configuration or authentication bootstrap token. A description of the transition conditions is available in Table 5.3. Certain configurations are only applicable during certain states of the pod. For example, the configuration `hostNetwork` is applicable when the state of the pod is 'host_network_access'. The context of states for pod configurations requires encoding and analyzing these states in forms of a finite state machine.

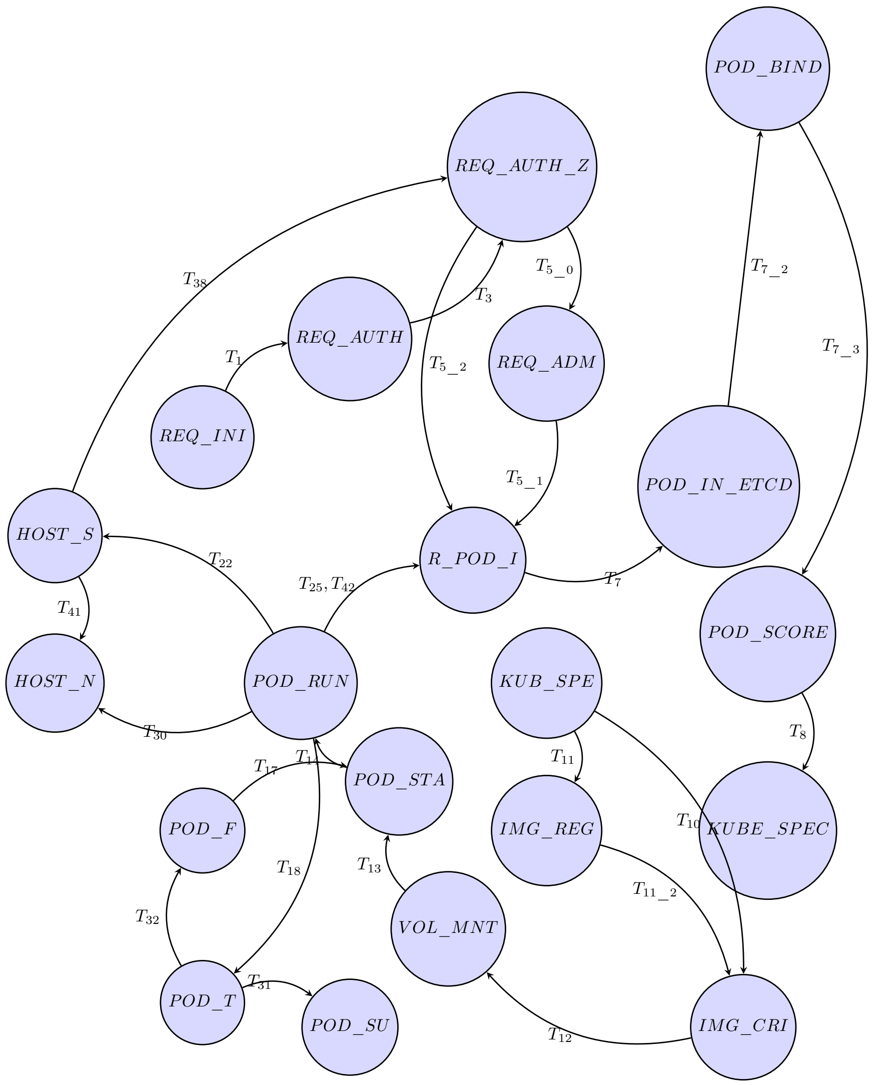


Figure 5.5: A finite state machine representing the states of a pod.

Table 5.1: Identified Pod Properties from the Internet Artifacts

Pod Properties	Count
Privilege escalation (privilege_escalation)	10
System admin capability (CAP_SYS_ADMIN)	10
Run as user (run_as_user)	10
Privileged security context (security_context_privileged)	9
Drop container capabilities container_DropCapabilities	9
Allow container capabilities (container_AllowedCapabilities)	9
Default container capabilities (container_DefaultCapabilities)	9
Host IPC enabled (hostIPC_enabled)	8
Linux security module SELinux enabled (lsm_SELinux_enabled)	8
Linux security module Seccomp enabled (lsm_SECCOMP_enabled)	8
Read only root file system (read_only_root_file_system)	8
Running as non root (running_as_NON_ROOT)	8
Pod Restricted Admission (pod_admission_RESTRICTED)	8
Host PID enabled (hostPID_enabled)	7
Host Network enabled (hostNetwork_enabled)	7
Default network policy deny everything (default_network_policy_all_ns_deny_everything)	7
FS group (fsGroup)	7
Supplemental group (supplementalGroup)	7
Run as group(run_as_group)	7
Host path enabledhost_path_enabled	6
Admission controller image scan (admission_controller_image_scan)	6
Default namespace (default_namespace)	6
Pod admission baseline (pod_admission_BASELINE)	6
Pod admission privileged (pod_admission_PRIVILEGED)	6
Enforce pod admission controller (pod_admission_ENFORCE)	6
Namespace resource quota enabled (namespace_resource_quota_enabled)	6
Pod CPU and memory request limit enabled (pod_cpu_memory_request_limit_enabled)	6
Pod CPU memory limit enabled (pod_cpu_memory_limit_enabled)	6
Namespace resource quota enabled (namespace_resource_quota_enabled)	6
Host process enabled (hostprocess_enabled)	5
Host port enabled (host_port_enabled)	5
Linux security module apparmor enabled (lsm_APPArmor_enabled)	5
Use of base container images (use_of_base_container_images)	5
Avoid tags and latest image tags (avoid_tags_and_latest_tags)	5
Use sha256 digest for image (use_sha256_digest_for_image)	5
Admission image policy webhook (admission_image_policy_webhook)	5
Avoid default service account (avoid_default_service_account)	5
Default proc mount (default_proc_mount)	4
Avoid environment variables in images (avoid_env_variables_images_images)	4
Use secret in images (use_secret_in_images)	4
Service account automount token (service_account_automount_token)	4
Container network interface supports network policy (cni_supports_network_policy)	4
Pod security exemption for user (pod_security_exemption_user)	4
Pod security exemption for workload pod (pod_security_exemption_workload_pod)	4
Pod security exemption for namespace (pod_security_exemption_namespace)	4
Minimal distroless image (minimal_distroless_image)	3
Unprivileged user for image build (unprivileged_user_for_build_image)	3
Capability NET_RAW (CAP_NET_RAW)	2
Docker socket enabled (docker_socket_enabled)	2
Volume usage permission (volume_usage_permission)	2
Sysctl namespaced (sysctl_namespaced)	2
Image pull policy (image_pull_policy)	2
Use external secret storage (use_external_secret_storage)	2
Network policy between pods (network_policy_between_pods)	2
File system (FS) group change policy (fsGroupChangePolicy)	2
Admission namespace lifecycle (admission_namespace_lifecycle)	1
Get secret (GET_secret)	1
List secret (LIST_secret)	1
Watch secret (WATCH_secret)	1
All verb secret (ALL_verb_secret)	1
All verb role (ALL_verb_role)	1
All verb resources (ALL_verb_resources)	1
Cluster admin (ClusterRole_cluster_admin)	1
Security context enabled (security_context_enabled)	1
Admission always pull images (admission_always_pull_images)	1
Liveness probe enabled (livenessprobe_enabled)	1
Readiness probe enabled (readinessprobe_enabled)	1
Limit node PID (limit_node_PID)	1
Limit pod PID (limit_pod_PID)	1
Pod eviction policy (pod_eviction_policy)	1

Table 5.2: Pod Security Requirements and Corresponding LTL formula

Pod Security Requirements	**Corresponding LTL Formula in NuXMV**
Any container running inside a pod with unnecessary privilege will not be in an unsafe state at runtime.	`LTLSPEC G (!TC_18 & !TC_19 & new_pod_creation_request) -> G ((CAP_SYS_ADMIN) -> G X(!over_privileged_container & pod_state=pod_running))`
Any cluster misconfiguration will not lead to an unsafe state.	`LTLSPEC G (!TC_19 & new_pod_creation_request) -> G((ALL_verb_resources) -> G X(!misconfigured_privilege & !container_breakout & pod_state=host_system_access))`
Lack of centralized policy such as missing admission controller will not lead a pod to an unsafe state.	`LTLSPEC G (!TC_18 & !TC_11_2 & !TC_19) -> G(!admission_image_signature_verification) -> G X (!admission_control_bypass & pod_state = host_system_access)`
Missing TLS encryption for communucation will not lead a pod to an unsafe state	`LTLSPEC G (TC_18 = FALSE & !TC_15 & !TC_25 & !TC_26 & !TC_19) -> G((!mTLS_encryption) -> G X(!network_misconfiguration & pod_state=host_network_access))`
Missing Network Policy will not lead a pod to an unsafe state.	`LTLSPEC G (TC_18 = FALSE & !TC_15 & !TC_25 & !TC_26 & !TC_19) -> G((!default_network_policy_all_ns_deny_everything) -> G X(!network_request_other_workload & pod_state=service_exposed))`
Any unscanned container image or misconfigured container image will not lead a pod to an unsafe state.	`LTLSPEC G( !TC_25 & !TC_19) -> G(!RCE_vulnerability)`
If any attacker can get secret from the container inside a pod then it will not lead a pod to an unsafe state.	`LTLSPEC G( !TC_11_2 & !TC_19) -> G (!container_secret_exfiltration)`
If any attacker with permission to read/watch secret will not lead a pod to an unsafe state.	`LTLSPEC G (!TC_26 & !TC_15 & !TC_19 & !TC_25) -> G(WATCH_secret) -> (G X (!host_secret_exfiltration))`
Any unnecessary permission to host file system will not provide access to underlying host system to exploit property.	`LTLSPEC G (!TC_18 & !TC_19 & !security_context_run_as_user) -> G ((fsGroup) -> G X(!host_file_system_access))`
A misconfigured image will not pull any image from unauthorized registry.	`LTLSPEC G (!TC_15 & !TC_11_2) -> G(!use_sha256_digest_for_image) -> G (!misconfigured_image)`
Any container running in a pod with out resource limit specified will not lead to disruption of other containers.	`LTLSPEC G (!TC_18 & node_resource_quota_enabled) -> G (!pod_eviction_from_node)`

Table 5.3: Implementation of Each Transition Condition (T)

Transition Condition	**Propositional Logic Formula for Transition**
T_1	(pod_state = request_initiated ∧ (Valid_Kubeconfig ∨ Auth_Bootstrap_Token))
T_2	(pod_state = request_initiated ∧ (¬(Valid_Kubeconfig ∨ Auth_Bootstrap_Token)))
T_3	(pod_state = request_authenticated ∧ (((API_request_verb_GET ∨ API_request_verb_LIST ∨ API_request_verb_DELETE ∨ API_request_verb_UPDATE ∨ API_request_verb_PATCH ∨ API_request_verb_CREATE) ∧ API_resource_pod) ∨ ClusterRole_cluster_admin))
T_4	(pod_state = request_authenticated ∧ (¬(API_request_verb_GET ∨ API_request_verb_LIST ∨ API_request_verb_DELETE ∨ API_request_verb_UPDATE ∨ API_request_verb_PATCH ∨ API_request_verb_CREATE) ∧ ¬API_resource_pod))
T_{5_0}	pod_state = request_authorized ∧ (((mutating_validating_admission_controller_available)))
T_{5_1}	pod_state = request_admission_controller_validated ∧ (((mutating_validating_admission_controller_available ∧ mutating_validating_admission_validation)))
T_{5_2}	pod_state = request_authorized ∧ ((¬mutating_validating_admission_controller_available))
T_6	pod_state = request_authorized ∧ ((authentication_authorization_successful ∧ default_admission_controller_pass ∧ mutating_validating_admission_controller_available) ∧ ¬(mutating_validating_admission_validation))
T_7	pod_state = request_pod_creation_initiated ∧ (request_accepted_k8s_api_server)
T_{7_2}	pod_state = desired_pod_state_stored_in_etcd ∧ (desired_state_written_at_etcd_by_api_server ∧ pod_object_created_by_resource_controller)
T_{7_3}	pod_state = pod_schedule_score_phase ∧ (pod_creation_pending_state ∧ pod_schedule_initiated)
T_8	pod_state = pod_schedule_bind_phase ∧ (scheduler_binds_node_to_pod ∧ podspec_poll_by_kubelet)
T_9	pod_state = kubelet_receives_podspec ∧((¬image_present ∧ image_pull_policy=NEVER) ∨ ¬imagepull_registry_valid_credentials)
T_{10}	pod_state = kubelet_receives_podspec ∧ (image_present ∧ (image_pull_policy=NEVER ∨ image_pull_policy=IFNOTPRESENT))
T_{11}	pod_state = kubelet_receives_podspec ∧ ((image_pull_policy=ALWAYS ∨ image_pull_policy =IFNOTPRESENT) ∧ imagepull_registry_valid_credentials)
T_{11_2}	pod_state = image_pulled_from_registry ∧ ((image_pull_policy=ALWAYS ∨ image_pull_policy =IFNOTPRESENT) ∧ imagepull_registry_valid_credentials)
T_{12}	pod_state = image_provided_to_cri ∧ (persistent_volume ∧ persistent_volume_claim ∧ secret_configmap_volume_mount)
T_{13}	pod_state = volume_mounted ∧ (container_initializing_or_ready ∧ pod_has_network)
T_{14}	pod_state = pod_starting ∧ container_poststart_webhook
T_{15}	pod_state = pod_running ∧ (missing_dependency_for_pod ∨ pod_runtime_error)
T_{16}	pod_state = error_crash_loop_backoff ∧ (pod_runtime_error ∧ ¬(livenessprobe_enabled ∨ startupprobe_enabled))
T_{17}	pod_state = pod_failed ∧ (¬sigkill_zero_exit ∧ ((livenessprobe_enabled ∨ readinessprobe_enabled)))
T_{18}	pod_state = pod_running ∧(((pod_disruption_budget ∧ (pod_disruption_allowed ∧ pod_disruption_budget_max_available_min_unavailable_condition_satisfie ∧ container_prestop_webhook ∧ pod_termination_grace_period_default_30s ∧ ¬pod_termination_force_NO_grace_period ∧ API_request_verb_DELETE ∧ API_resource_pod))
T_{19}	pod_state = pod_running ∧ (((¬pod_cpu_memory_limit_enabled ∧ ¬pod_cpu_memory_request_limit_enabled) ∧ node_resource_quota_enabled) ∨ (¬readinessprobe_enabled ∧ pod_eviction_pod_preemtion ∧ node_resource_quota_enabled) ∨ (¬limit_node_PID ∨ ¬limit_pod_PID ∨ ¬pod_eviction_policy) ∨ (namespace_resource_quota_enabled ∧(¬pod_cpu_memory_request_limit_enabled ∧ ¬pod_cpu_memory_request_limit_enabled)))
T_{20}	pod_state = pod_running ∧ (pod_CNI_enabled ∧ clusterIP_NodePort_exposed)
T_{22}	pod_state = pod_running ∧ (CAP_NET_RAW ∨ CAP_SYS_ADMIN ∨ network_request_other_workload ∨security_context_privileged ∨ ¬security_context_run_as_user ∨ lsm_SECCOMP_enabled ∨ lsm_APPArmor_enabled ∨ lsm_SELinux_enabled ∨ hostprocess_enabled ∨ hostNetwork_enabled ∨ hostPID_enabled ∨ hostIPC_enabled ∨ host_path_enabled ∨ ¬container_DropCapabilites ∨ ¬running_as_NON_ROOT ∨ pod_admission_BASELINE ∨ ¬pod_admission_RESTRICTED ∨ pod_admission_PRIVILEGED ∨ ¬pod_admission_ENFORCE ∨ pod_security_exemption_user ∨ pod_security_exemption_namespace ∨ pod_security_exemption_workload_pod ∨ ¬network_policy_between_pods ∨ docker_socket_enabled ∨ default_namespace ∨ service_account_privileged ∨ service_account_automount_token)
T_{23}	pod_state = service_exposed ∧ (¬CAP_NET_RAW ∧ ¬CAP_SYS_ADMIN ∧ ¬security_context_privileged ∧ security_context_run_as_user ∧ lsm_SECCOMP_enabled ∧ lsm_APPArmor_enabled ∧ lsm_SELinux_enabled ∧ ¬hostprocess_enabled ∧ ¬hostNetwork_enabled ∧ ¬hostPID_enabled ∧ ¬hostIPC_enabled ∧ ¬host_path_enabled ∧ container_DropCapabilites ∧ running_as_NON_ROOT ∧ ¬pod_admission_BASELINE ∧ pod_admission_RESTRICTED ∧ ¬pod_admission_PRIVILEGED ∧ pod_admission_ENFORCE ∧ pod_security_exemption_user ∧ pod_security_exemption_namespace ∧ pod_security_exemption_workload_pod ∧ network_policy_between_pods ∧ ¬default_namespace ∧ ¬docker_socket_enabled ∧ ¬service_account_privileged ∧ ¬service_account_automount_token)
T_{24}	pod_state = service_exposed ∧ (CAP_NET_RAW ∨ CAP_SYS_ADMIN ∨ network_request_other_workload ∨ security_context_privileged ∨ security_context_run_as_user ∨ lsm_SECCOMP_enabled ∨ lsm_APPArmor_enabled ∨ lsm_SELinux_enabled ∨ hostprocess_enabled ∨ hostNetwork_enabled ∨ hostPID_enabled ∨ hostIPC_enabled ∨ host_path_enabled ∨ ¬container_DropCapabilites ∨ ¬running_as_NON_ROOT ∨ pod_admission_BASELINE ∨ ¬pod_admission_RESTRICTED ∨ pod_admission_PRIVILEGED ∨ ¬pod_admission_ENFORCE ∨ pod_security_exemption_user ∨ pod_security_exemption_namespace ∨ pod_security_exemption_workload_pod ∨ ¬network_policy_between_pods ∨ docker_socket_enabled ∨ default_namespace ∨ service_account_privileged ∨ service_account_automount_token)
T_{25}	pod_state = pod_running ∧ new_pod_creation_request ∧ (container_breakout ∨ malicious_container ∨ pod_admission_PRIVILEGED ∧ (¬admission_namespace_lifecycle ∧ admission_control_bypass))
T_{26}	pod_state = host_system_access ∧ (container_secret_exfiltration)
T_{28}	pod_state = image_pulled_from_registry ∧ unscanned_container_image
T_{29}	pod_state = container_registry_poisoned ∧ (misconfigured_image ∨ unscanned_container_image)
T_{30}	pod_state = pod_running ∧ network_misconfiguration
T_{31}	pod_state = pod_terminated ∧ sigkill_zero_exit
T_{32}	pod_state = pod_terminated ∧ ¬sigkill_zero_exit
T_{33}	pod_state = pod_unrestricted_communication ∧ (network_misconfiguration ∨ remote_service_connected_from_cluster)
T_{35}	pod_state = host_system_access ∧ (RCE_vulnerability ∨ misconfigured_image)
T_{36}	pod_state = host_system_access ∧ container_breakout
T_{38}	pod_state = host_system_access ∧ service_account_privileged
T_{40}	pod_state = host_network_access ∧ (malicious_container ∨ over_privileged_container ∨ network_request_other_workload)
T_{41}	pod_state = host_system_access ∧ network_misconfiguration
T_{42}	pod_state = remote_service_connected ∧ remote_service_connected_from_cluster ∧ malicious_container
T_{43}	pod_state = remote_service_connected ∧ service_account_privileged
T_{44}	pod_state = kube_api_server_is_updated_by_kubelet ∧ container_breakout
T_{45}	pod_state = kubelet_receives_podspec ∧ container_state = WAITING ∧ pod_schedule_completed

5.2 Answer to RQ 2.1 Identifying Configurations that Lead Pod in an Unsafe State

5.2.1 Identification of Configurations for Pod Unsafe State

If the NuXMV generates counter-examples, we analyze the counter-examples. We trace the sequence of configurations and events that lead to the violation of the property given an initial configuration of pod. For example, the pod security requirement "*An over privileged running pod will never go to an unsafe state*" can be represented as `!over_privileged_container & pod_state = pod_running)` is `false`. We illustrate the generated counter-example for the requirement and only represent its relevant configurations in Figure 5.6.

In the initial `state 1.1`, the pod state is `request_initiated`. We find that the state variables in our model related to pod security requirements `hostPID_enabled`, `service_account_automount_token`, `pod_security_exemption_workload_pod`, `security_context_privileged`, and `service_account_privileged` are all set to `TRUE`. In the next `state 1.2`, the pod state is `request_authenticated` and state variables in our model related to pod security requirements `ClusterRole_clusteradmin`, `host_namespace_accesss` and `RCE_vulnerability` becomes `TRUE`. In the next `state 1.3`, the pod state is `request_authorized` and the state variables in our model related to pod security requirements `API_request_verb_UPDATE`, `API_request_verb_PATCH`, `API_request_verb_DELETE` and `API_resource_pod` becomes `TRUE`. In the `state 1.7`, the pod state is `pod_schedule_bind_phase` and the state variables in our model related to pod security requirements `host_secret_exfiltration`, `host_file_system_access` becomes `TRUE`. Eventually, in the `state 1.12`, the pod state becomes `pod_starting`, which symbolizes that the pod has started. In `state 1.13`, the pod state becomes `pod_running`, and the state variable `container_prestop_webhook` becomes `TRUE`. In prior states, the pod has got access to host secret with `host_secret_exfiltration` and `privileged_service_account`. Hence in the next `state 1.14`, the pod again initiates a pod creation request and reaches state `request_pod_creation_initiated`. Finally, in `state 1.22`, the pod reaches the state `pod_running`, which means the over privileged pod initiated another pod with privileged service account.

We identify 23 attack-akin configuration from our counterexample-based analysis. Definitions for each configuration is provided below. The configurations that are common across all six attacks are: escalated child process, capability abuse, active `hostIPC`, active `hostNetwork`, active `hostPID`, missing admission controller, privileged `securityContext`, inactive read-only for root filesystem, inactive `runAsNonRoot`, inactive `runAsUser`, and default `serviceAccount`.

1. **Active** `hostIPC` - This configuration provides a container within a pod with the privilege to intercept all IPC communications of the host machine.

2. **Active** `hostNetwork` - The configuration to activate `hostNetwork` with `hostNetwork: true`. This configuration provides applications the privilege to ping and intercept all network interfaces of the host machine.

3. **Active** `hostPID` - The configuration to activate `hostPID` with `hostPID: true`. This configuration allows a pod to access the namespace and find all the processes running on the host.

4. **Capability abuse** - The configuration to activate Linux capabilities. This category includes configurations that allow a pod to gain root-level access. This category includes two sub-categories: `CAP_SYS_ADMIN` and `CAP_SYS_MODULE`.

5. **Escalated child process** - The configuration that allows a child process in a container to gain more privilege than its parent process.

6. **Privileged** `securityContext` - The configuration that allows a privileged `securityContext` by disabling all security features of the pod or container.

7. **Missing resource limit** - The configuration that allows a container within a pod to run without CPU and memory limit, in turn consuming all available resources. The `limits` and `resources` keywords are used to specify resource limits for containers within a pod.

8. **Default namespace** - The configuration that activates usage of the default namespace. The configuration allows a pod to be deployed in a shared virtual cluster.

9. **Absent `securityContext`** - The configuration allows a pod to be deployed without specifying the `securityContext` that helps to enable security features of the pod or container.

10. **Missing network policy** - The configuration of not using the `NetworkPolicy` object to specify network policies for pods. This configuration facilitates unrestricted traffic between pods as `NetworkPolicy` is used to control the flow of traffic between pods.

11. **Missing SSL/TLS for HTTP** - The configuration of using HTTP without SSL or TLS support. This configuration allows the transmission of HTTP-based pod traffic between pods inside the Kubernetes cluster without SSL/TLS encryption.

12. **Control plane node selector** - The configuration of using **nodeName: master**. This configuration allows a pod to be deployed in the control plane node.

13. **Missing admission controller** - The configuration of not using an admission controller with `AdmissionConfiguration`. An admission controller intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized. This configuration allows an authenticated and authorized request to bypass the security compliance for pod creation.

14. **Privileged role** - The configuration that allows excessive permission to a role. A role is a code construct that allows permissions for a particular namespace [41]. Using the `verbs: ["*"]` configuration for role-related rules, a role becomes privileged. Verbs in Kubernetes are used to specify all possible permissions for a Kubernetes pod.

15. **Privileged `ServiceAccount`** - The configuration of creating a privileged `ServiceAccount`. A service account is a non-human account that provides an identity for processes that run in a pod. In Figure 5.7, 'sample-sa' is a privileged `ServiceAccount` as it uses a `Role` and `RoleBinding` with a default namespace.

16. **Privileged `RoleBinding`** - The configuration that allows a `RoleBinding` object for a Kubernetes-based cluster to list, create, modify, and delete any resources in the entire cluster

in an unauthorized fashion. `RoleBinding` is a Kubernetes object that grants the permissions defined in a role to a user or set of users. The `cluster-admin` configuration used in the context of `roleRef` allows a `RoleBinding` object to become privileged.

17. **Inactive read-only for root filesystem** - The configuration that allows the mounting of the container's root file system to be writable with `readOnlyRootFileSystem: False`.

18. **Auto mounted token** - The configuration that allows a pod to automatically activate tokens used for service accounts inside the pod with `automountServiceAccountToken: false`. These tokens are used to authenticate requests from processes within the cluster to the Kubernetes API server.

19. **Default** `ServiceAccount` - The configuration that allows a pod to use the default service account.

20. **Privileged default** `ServiceAccount` - The configuration that allows a privileged role to be used by a default service account.

21. **Active** `hostPath` - The configuration that allows mounting of a file or a directory from the host node filesystem into a pod with `active hostPath`.

22. **Inactive** `runAsNonRoot` - The configuration that allows unauthorized write permissions for the filesystem inside a container of a pod using `runAsNonRoot: false`.

23. **Inactive** `runAsUser` - The configuration that allows a container to run as a root user inside a pod using `runAsUser: false`.

Table 5.11 provides a mapping between pod-related security attacks and the derived configurations. We observe, each attack to include multiple configurations, e.g., executing the workload attack requires 18 configurations. The implication of this finding is that a single configuration in a Kubernetes manifest cannot lead to any of the studied attacks listed in Section 5.4.

5.2.2 Identification of Pod Events for Pod Unsafe States

NuXMV generates counterexamples when there is a pod security requirement violation in our FSM for a pod. We observe the change of individual state variables in the counterexamples. We construct transition conditions as propositional logic. The change of state variables causes the change in the transition condition. When the transition condition changes, such as `True` from `False`, we observe a transition to an event from the previous event. From Figure 5.6, we observe that the change of state variables in every `pod_state` of the FSM allows transition to the subsequent event. We extract the events from the counterexample, the change in the value of state variables, and `pod_state`.In Table 5.4, in the leftmost column "Current State", we specify the state of FSM for a pod from before transition such as `pod_running` state. Then, in the next column, "Expected State," we list all the safe states of the FSM for a pod transitioning from "Current State" state. For instance, a pod should be in `service_exposed` or `pod_terminated` state as the next state of the `pod_running` state. In the third column "Unsafe state," we provide the state of FSM for a pod for transition from "Current State" due to a combination of misconfigurations. For instance, a transition from `pod_running` state to `pod_creation_request_initiated` state can happen due to combination of misconfigurations in transition conditions. In the rightmost column " Transition Conditions", we provide the combination of misconfigurations which lead a pod to unsafe state such as from `pod_running` state to `pod_creation_request_initiated` state.

We identify the following pod events that occur prior to insecure pod events from the counterexamples as follows:

Request authenticated (`request_authenticated`): This is the pod event where the Kubernetes API server authenticates the pod creation request from the practitioner.

Request authorized (`request_authorized`): This is the pod event where the Kubernetes API server authorizes the pod creation request.

Pod creation request initiated (`request_pod_creation_initiated:`): This is the pod event where the Kubernetes API server creates a pod object from the authenticated and authorized pod creation request.

Desired pod state stored in etcd (`desired_pod_state_stored_in_etcd`): This is the pod event where the Kubernetes API stores the pod object specification in the etcd database as a desired pod state.

Pod scheduling score phase(`pod_schedule_score_phase`): This is the pod event where the Kubernetes scheduler calculates scores for scheduling a pod.

Pod scheduling bind phase(`pod_schedule_bind_phase`): This is the pod event where the Kubernetes scheduler binds the pod to a specific worker node.

Kubelet receives pod specification (`kubelet_receives_podspec`): This is the pod event where the kubelet agent in the worker node receives the pod specification from the Kubernetes API server.

Image pull from registry (`image_pulled_from_registry`): This is the pod event where the kubelet agent in the worker node pulls image from the container image registry.

Container registry poisoned (`container_registry_poisoned`): This is the pod event where a malicious agent provides malicious image to kubelet and pushes malicious image to the container image registry with stolen credentials from kubelet agent.

Image provided to container runtime interface (`image_provided_to_cri`): This is the pod event where the kubelet agent provides the container image to the container runtime interface (CRI).

Volume mounted(`volume_mounted`): This is the state of pod event where the pod mounts volume for the container and allocates persistent storage for stateful pods.

Pod starting(`pod_starting`): This is the pod event where the pod has a network, has storage, and at least one container started running inside the pod.

Pod running(`pod_running`): This is the pod event where the container state is `RUNNING`.

Pod evicted(`pod_evicted`): This is the pod event where one misconfigured pod evicts other pods from the worker node due to high resource consumption causing a resource-related denial of service attack.

Service exposed(`service_exposed`): This is the pod event where the running pod creates an endpoint and exposes the IP address so that other services can access it.

Pod unrestricted communication (`pod_unrestricted_communication`): This is the pod event where the pod or exposed service has some misconfiguration that allows open network communication and lateral movement inside the Kubernetes cluster.

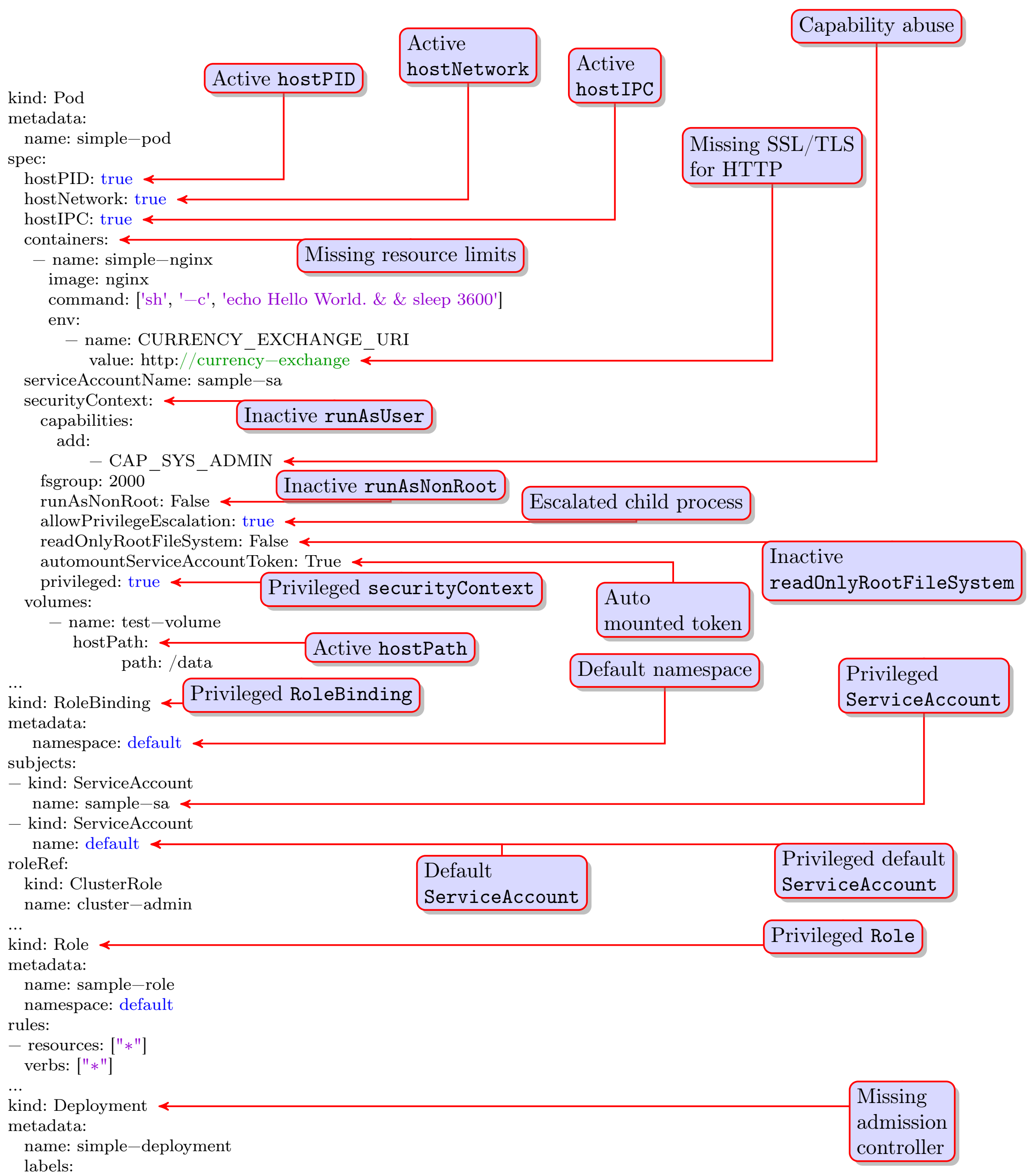
Remote service connected (`remote_service_connected`): This is the pod event where the pod or exposed service has exposed shell to an unauthorized remote machine.

Table 5.4: Misconfigurations that invoke insecure pod events

Current state	Expected State	Unsafe State	Transition Conditions
pod_running	service_exposed , pod_terminated	pod_creation_request_initiated	(request_authorized & !mutating_validating_admission_controller_available) --> (new_pod_creation_request & (container_breakout | malicious_container | pod_admission_PRIVILEGED) --> ((!admission_namespace_lifecycle & admission_control_bypass)) --> (remote_service_connected_from_cluster & malicious_container)
pod_running	service_exposed, pod_terminated	pod_evicted	(((!pod_cpu_memory_limit_enabled & !pod_cpu_memory_request_limit_enabled) & node_resource_quota_enabled) | (!readinessprobe_enabled & pod_eviction_pod_preemtion & node_resource_quota_enabled) | (!limit_node_PID |!limit_pod_PID | !pod_eviction_policy) | (namespace_resource_quota_enabled & (!pod_cpu_memory_request_limit_enabled & !pod_cpu_memory_request_limit_enabled)))
service_exposed	pod_terminated	pod_unrestricted_communication	(request_authorized & !mutating_validating_admission_controller_available) --> (new_pod_creation_request & (container_breakout | malicious_container | pod_admission_PRIVILEGED) --> ((!admission_namespace_lifecycle & admission_control_bypass)) --> (remote_service_connected_from_cluster & malicious_container) --> (CAP_NET_RAW | CAP_SYS_ADMIN | network_request_other_workload | security_context_privileged | security_context_run_as_user | lsm_SECCOMP_enabled | lsm_APPArmor_enabled | lsm_SELinux_enabled | hostprocess_enabled | hostNetwork_enabled | hostPID_enabled | hostIPC_enabled | host_path_enabled | !container_DropCapabilites | !running_as_NON_ROOT | pod_admission_BASELINE | !pod_admission_RESTRICTED | pod_admission_PRIVILEGED | !pod_admission_ENFORCE | pod_security_exemption_user | pod_security_exemption_namespace | pod_security_exemption_workload_pod | !network_policy_between_pods | docker_socket_enabled | default_namespace | service_account_privileged | service_account_automount_token)

```
-> State: 1.1 <-
  hostPID_enabled = TRUE
  service_account_automount_token = TRUE
  pod_security_exemption_workload_pod = TRUE
  security_context_privileged = TRUE
  admission_controller_image_scan = FALSE
  service_account_privileged = TRUE
-> State: 1.2 <-
  ClusterRole_cluster_admin = TRUE
  host_namespace_access = TRUE
  RCE_vulnerability = TRUE
  pod_state = request_authenticated
-> State: 1.3 <-
  API_request_verb_UPDATE = TRUE
  API_request_verb_PATCH = TRUE
  API_request_verb_DELETE = TRUE
  API_resource_pod = TRUE
-> State: 1.4 <-
  pod_object_created_by_resource_controller = TRUE
  remote_service_connected_from_cluster = TRUE
  pod_state = request_pod_creation_initiated
  ...
-> State: 1.5 <-
  pod_state = desired_pod_state_stored_in_etcd
  ...
-> State: 1.6 <-
  pod_state = pod_schedule_score_phase
  ...
-> State: 1.7 <-
  host_secret_exfiltration = TRUE
  host_file_system_access = TRUE
  pod_state = pod_schedule_bind_phase
  ...
-> State: 1.8 <-
  pod_state = kubelet_receives_podspec
  ...
-> State: 1.9 <-
  pod_state = image_pulled_from_registry
  ...
-> State: 1.10 <-
  pod_state = image_provided_to_cri
  ...
-> State: 1.11 <-
  pod_state = volume_mounted
  ...
-> State: 1.12 <-
  pod_state = pod_starting
  ...
-> State: 1.13 <-
  pod_state = pod_running
  ...
-- Loop starts here
-> State: 1.14 <-
  pod_state = request_pod_creation_initiated
  ...
-> State: 1.15 <-
  pod_state = desired_pod_state_stored_in_etcd
  ...
-> State: 1.16 <-
  pod_state = pod_schedule_score_phase
  ...
-> State: 1.17 <-
  pod_state = pod_schedule_bind_phase
-> State: 1.18 <-
  image_pull_policy = IFNOTPRESENT
  pod_state = kubelet_receives_podspec
  ...
-> State: 1.19 <-
  pod_state = image_provided_to_cri
  ...
-> State: 1.20 <-
  pod_state = volume_mounted
  ...
-> State: 1.21 <-
  pod_state = pod_starting
  ...
-> State: 1.22 <-
  pod_state = pod_running
  ...
-> State: 1.23 <-
  pod_state = request_pod_creation_initiated
  ...
```

Figure 5.6: A counter-example for over privileged pod



```
kind: Pod
metadata:
  name: simple-pod
spec:
  hostPID: true
  hostNetwork: true
  hostIPC: true
  containers:
   - name: simple-nginx
     image: nginx
     command: ['sh', '-c', 'echo Hello World. & & sleep 3600']
     env:
       - name: CURRENCY_EXCHANGE_URI
         value: http://currency-exchange
  serviceAccountName: sample-sa
  securityContext:
    capabilities:
      add:
          - CAP_SYS_ADMIN
    fsgroup: 2000
    runAsNonRoot: False
    allowPrivilegeEscalation: true
    readOnlyRootFileSystem: False
    automountServiceAccountToken: True
    privileged: true
  volumes:
     - name: test-volume
       hostPath:
             path: /data
...
kind: RoleBinding
metadata:
   namespace: default
subjects:
- kind: ServiceAccount
   name: sample-sa
- kind: ServiceAccount
   name: default
roleRef:
  kind: ClusterRole
  name: cluster-admin
...
kind: Role
metadata:
  name: sample-role
  namespace: default
rules:
- resources: ["*"]
  verbs: ["*"]
...
kind: Deployment
metadata:
  name: simple-deployment
  labels:
```

Figure 5.7: Example code snippet to demonstrate attack-akin configurations.

5.3 Answer to RQ 2.2

Prior survey among practitioners demonstrated the prevalence of the misconfigurations in Kubernetes [61]. Rahman et al. identified misconfigurations in Kubernetes manifests in open source software(OSS) repositories [57]. We extend the open source tool SLIKUBE [57] and build the tool (SLIKUBE+) to detect the presence of additional 12 security misconfigurations in Kubernetes OSS repositories. As an input the user will provide the directory path with Kubernetes manifests and SLIKUBE+ will output the count of misconfigurations for each of the Kubernetes manifests in the directory.

5.3.1 SLIKUBE+

We describe the rules of our extension SLIKUBE called SLIKUBE+ in Table 5.5. We describe the string pattern for the rules in Table 5.6. The results from our analysis is described in Table 5.8. In total, we identify 23 types of misconfigurations that are related pod security requirements described in Section 5.2

Table 5.5: Rules for SLIKUBE+ to detect security misconfigurations

Misconfiguration Name	**Rule**
Control plane node selector	$(\mathit{isNodeName}(x) \wedge \mathit{hasNameMaster}(x.\mathit{value}))$
Missing Admission Controller	$\mathit{isKind}(x) \wedge (\mathit{isApiVersion}(x) \wedge (\neg \mathit{isAdmissionConfiguration}(x.\mathit{value}) \vee \neg \mathit{isAdmissionReview}(x.\mathit{value}) \vee \neg \mathit{isValidationAdmission}(x.\mathit{value}) \vee \neg \mathit{isMutatingAdmission}) \vee \neg \mathit{isApiVersionAdmission}(x.\mathit{value}))$
Privileged Role	$(\mathit{isKind}(x) \wedge (\mathit{isRole}(x.\mathit{value}) \vee \mathit{isClusterRole}(x.\mathit{value}))) \wedge ((\mathit{isVerb}(x) \wedge \mathit{hasRiskyVerbPrivilege}(x)) \wedge ((\mathit{isResource}(x) \wedge \mathit{hasRiskyResourcePrivilege})) \wedge \mathit{isName}(x) \wedge$
Privileged `ServiceAccount`	$(\mathit{isKind}(x) \wedge (\mathit{isRole}(x.\mathit{value}) \vee \mathit{isClusterRole}(x.\mathit{value}))) \wedge ((\mathit{isVerb}(x) \wedge \mathit{hasRiskyVerbPrivilege}(x)) \wedge ((\mathit{isResource}(x) \wedge \mathit{hasRiskyResourcePrivilege}(x)) \wedge \mathit{isName}(x) \wedge (\mathit{isKind}(x) \wedge \mathit{isClusterRoleBinding}(x) \vee \mathit{isRoleBinding}(x) \wedge (\mathit{isRole}(x) \wedge \mathit{isKindClusterRole}(x) \wedge \mathit{isKindServiceAccount}(x)))$
Privileged `RoleBinding`	$(\mathit{isKind}(x) \wedge (\mathit{isClusterRoleBinding}(x) \vee \mathit{isRoleBinding}(x) \wedge (\mathit{isClusterAdmin}())))$
Inactive read-only root filesystem	$\mathit{isKind}(x) \wedge ((\mathit{isKindPod}(x) \vee \mathit{isKindDeployment}(x) \vee \mathit{isKindDaemonSet}(x) \vee \mathit{isReplicaSet}(x)) \wedge \mathit{isVolumeMount}(x) \wedge (\mathit{isReadOnlyRootFileSystem}(x) \vee \mathit{isReadOnlyFileSystem}(x) \wedge (\mathit{isEnabled}(x) \vee (\neg \mathit{isReadOnlyRootFileSystem}(x) \wedge \neg \mathit{isReadOnlyFileSystem}(x))$
Auto mount token	$\mathit{isKind}(x) \wedge (\mathit{isKindPod}(x) \vee \mathit{isKindDeployment}(x) \vee \mathit{isKindDaemonSet}(x) \vee \mathit{isReplicaSet}(x) \vee \mathit{isKindService}(x)) \wedge ((\mathit{isAutomountServiceToken}(x) \wedge (\mathit{isEnabled}(x))) \vee \neg(\mathit{isAutomountServiceToken}(x)))$
Default `ServiceAccount`	$\mathit{isKind}(x) \wedge (\mathit{isKindPod}(x) \vee \mathit{isKindDeployment}(x) \vee \mathit{isKindDaemonSet}(x) \vee \mathit{isReplicaSet}(x) \vee \mathit{isKindService}(x)) \wedge \neg \mathit{isKeyServiceAccountName}(x)$
Privileged default `ServiceAccount`	$\mathit{isKind}(x) \wedge (\mathit{isRole}(x.\mathit{value}) \vee \mathit{isClusterRole}(x.\mathit{value})) \wedge ((\mathit{isVerb}(x) \wedge \mathit{hasRiskyVerbPrivilege}(x)) \wedge (\mathit{isResource}(x) \wedge \mathit{hasRiskyResourcePrivilege}(x)) \wedge (\mathit{isName}(x) \wedge (\mathit{isDefault}(x)))$
Active `hostPath`	$\mathit{isKind}(x) \wedge ((\mathit{isKindPod}(x) \vee \mathit{isKindDeployment}(x) \vee \mathit{isKindDaemonSet}(x) \vee \mathit{isReplicaSet}(x)) \wedge \mathit{isHostPath}(x) \wedge (\mathit{isReadOnlyRootFileSystem}(x) \vee \mathit{isReadOnlyFileSystem}(x) \wedge (\mathit{isEnabled}(x) \vee (\neg \mathit{isReadOnlyRootFileSystem}(x) \wedge \neg \mathit{isReadOnlyFileSystem}(x))$
Inactive `runAsNonRoot`	$\mathit{isKind}(x) \wedge ((\mathit{isKindPod}(x) \vee \mathit{isKindDeployment}(x) \vee \mathit{isKindDaemonSet}(x) \vee \mathit{isReplicaSet}(x)) \wedge (\mathit{isRunAsNonRoot}(x) \wedge (\mathit{isEnabled}(x) \vee (\neg \mathit{isRunAsNonRoot}(x))$
Inactive `runAsUser`	$\mathit{isKind}(x) \wedge ((\mathit{isKindPod}(x) \vee \mathit{isKindDeployment}(x) \vee \mathit{isKindDaemonSet}(x) \vee \mathit{isReplicaSet}(x)) \wedge (\neg(\mathit{isRunAsUser}(x)) \vee (\neg \mathit{isRunAsGroup}(x)))$

Table 5.6: String Patterns Used for Functions in Rules

Function	**String Pattern**
isImagePullSecrets()	'imagePullSecrets'
isImagePullPolicy()	'imagePullPolicy'
isAlways()	'Always'
isNodeName()	'nodeName'
hasNameMaster()	'master', 'control-plane', 'controlplane'
isKind()	'kind'
isApiVersion()	'apiVersion'
isApiVersionAdmission()	'admissionregistration.k8s.io/v1'
isAdmissionConfiguration()	'AdmissionConfiguration'
isAdmissionReview()	'AdmissionReview'
isValidationAdmission()	'ValidatingWebhookConfiguration'
isMutatingAdmission()	'MutatingWebhookConfiguration'
isRole()	'Role'
isClusterRole()	'ClusterRole'
isVerb()	'verbs'
isResource()	'resources'
hasRiskyVerbPrivilege()	'list', 'watch', 'create', 'update', 'patch', 'delete', '*'
hasRiskyResourcePrivilege()	'pod', 'daemonset', 'secrets', '*'
isClusterRoleBinding	'ClusterRoleBinding'
isRoleBinding	'RoleBinding'
isServiceAccount	'ServiceAccount'
isClusterAdmin()	'cluster-admin'
isDefault()	'default'
isKindPod()	'Pod'
isKindDeployment()	'Deployment'
isKindService()	'Service'
isKindDaemonSet	'daemonset'
isKindReplicaSet()	'ReplicaSet'
isVolumeMount()	'volumeMounts'
isReadOnlyRootFileSystem()	'readOnlyRootFileSystem'
isReadOnlyFileSystem()	'readOnlyFileSystem'
isEnabled()	'True', 'true'
isAutomountServiceToken()	'automountServiceAccountToken'
isKeyServiceAccountName()	'serviceAccountName'
isVerb()	'verbs'
isResource()	'resources'
isName()	'name'
isRunAsNonRoot()	'runAsNonRoot'
isRunAsUser()	'runAsUser'
isRunAsGroup()	'runAsGroup'

5.3.2 Evaluation of SLIKUBE+

We evaluate SLIKUBE+ with the dataset used to evaluate SLIKUBE [57]. We describe the attributes of the dataset in Table 5.7. We use the following criteria to curate 51 GitHub and 14 GitLab repositories following the process of prior work [57].

- **Criterion-1:** The repository must have at least 10% files that are Kubernetes manifests.
- **Criterion-2:** The repository is available to download.
- **Criterion-3:** The repository is not a clone to another repository.
- **Criterion-4:** The repository has at least 2 commits per month. We set this criteria to filter out repository that has little activity.
- **Criterion-5:** The repository has at least five contributors. We set this criteria so that we can eliminate projects for personal use.
- **Criterion-6:** The repository does not contain projects that is used to demonstrate examples, conduct course works and used as book chapter.

Table 5.7: Misconfigurations in OSS: Dataset Descriptions

Attribute	**GitHub**	**GitLab**
Repositories	51	14
Kubernetes Manifests	1,590	449
Size (LOC)	146,361	39,236
Age (In days till 12/2021)	2252	2252

We describe the evaluation result of SLIKUBE+ on the dataset in Table 5.8.

Table 5.8: Kubernetes Security Misconfigurations in OSS

Misconfigurations	**GitHub**	**GitLab**
Active `hostIPC`	1	0
Active `hostPID`	5	0
Active `hostNetwork`	11	3
Escalated child process	3	0
Absent `securityContext`	90	4
Capability abuse	0	20
Privileged `securityContext`	3	9
Missing Resource Limit	69	10
Missing Network Policy	1,508	396
Default Namespace	95	2
Missing SSL/TLS for HTTP	395	217
Control plane node selector	0	0
Missing admission controller	1509	395
Privileged Role	47	35
Privileged `ServiceAccount`	73	51
Privileged `RoleBinding`	11	1
Inactive read-only root file system	227	43
Auto mount token	1	0
Privileged default `ServiceAccount`	0	1
Default `ServiceAccount`	784	125
Active `hostPath`	86	12
Inactive `runAsNonRoot`	560	72
Inactive `runAsUser`	556	72

5.3.3 Comparison of SLIKUBE+ with Existing Tools

We compare our SLIKUBE+ with SLIKUBE tool and existing static analysis tools for identifying Kubernetes security misconfigurations. SLIKUBE+ reports 12 more security misconfigurations than SLIKUBE [3]. Similar to SLIKUBE, compare our tool SLIKUBE+ with four state-of-the-art static analysis tool as follows: Checkov [9], KubeLinter [40], Datree [18], and Snyk [68]. We inspect the policy or rules of each the static analysis tool from their online documentation and idnetify which of the categories of our SLIKUBE+ tool are identified by each of these tools. We observe that only SLIKUBE+ detects all 23 category of security misconfigurations. Checkov, KubeLinter, Datree and Snyk do not identify 7, 3, 4 and 7 categories of Kubernetes security misconfigurations respectively.

Table 5.9: Comparison of SLIKUBE+ with Existing Tools

Misconfiguration Name	**SLIKUBE+**	**Checkov**	**KubeLinter**	**Datree**	**Snyk**
Active `hostIPC`	✓	✓	✓	✓	✓
Active `hostNetwork`	✓	✓	✓	✓	✓
Active `hostPID`	✓	✓	✓	✓	✓
Capability Abuse	✓	✓	✓	✓	✓
Escalated child process	✓	✓	✓	✓	✓
Privileged `securityContext`	✓	✓	✓	✓	✓
Missing Resource Limit	✓	✓	✓	✓	✓
Absent `securityContext`	✓	✓	×	✓	×
Missing SSL/TLS for HTTP	✓	×	×	×	×
Default namespace	✓	✓	✓	✓	✓
Missing network policy	✓	×	✓	×	✓
Control plane node selector	✓	✓	✓	×	×
Missing admission controller	✓	×	×	×	×
Privileged role	✓	✓	✓	✓	✓
Privileged `ServiceAccount`	✓	✓	✓	✓	✓
Privileged `RoleBinding`	✓	×	✓	✓	×
Inactive read-only for root filesystem	✓	✓	✓	✓	✓
Auto mounted token	✓	✓	×	✓	×
Privileged default `ServiceAccount`	✓	✓	✓	✓	✓
Active `hostPath`	✓	×	✓	✓	×
Default `ServiceAccount`	✓	✓	✓	✓	✓
Inactive `runAsNonRoot`	✓	×	✓	✓	✓
Inactive `runAsUser`	✓	×	✓	✓	✓

Table 5.10: Comparison between SLIKUBE+ and SLIKUBE

Misconfiguration Name	**SLIKUBE+**	**SLIKUBE**
Active `hostIPC`	✓	✓
Active `hostPID`	✓	✓
Active `hostNetwork`	✓	✓
Escalated child process	✓	✓
Absent `securityContext`	✓	✓
Capability Abuse	✓	✓
Privileged `securityContext`	✓	✓
Missing Resource Limit	✓	✓
Missing SSL/TLS for HTTP	✓	✓
Missing network policy	✓	✓
Default namespace	✓	✓
Control plane node selector	✓	×
Missing admission controller	✓	×
Privileged role	✓	×
Privileged `ServiceAccount`	✓	×
Privileged `RoleBinding`	✓	×
Inactive read-only root filesystem	✓	×
Auto mount token	✓	×
Privileged default `ServiceAccount`	✓	×
Active `hostPath`	✓	×
Default `ServiceAccount`	✓	×
Inactive `runAsNonRoot`	✓	×
Inactive `runAsUser`	✓	×

5.4 Validation of Attacks for Identified Configurations

We validate the attacks for misconfigurations using `kubeadm` installation of Kubernetes cluster. The `kubeadm` installation provides multi-node vanilla Kubernetes cluster. We use vagrant to set up a three node Kubernetes cluster with one control plane and two worker nodes using virtual box VMs for nodes in our local workstation. The configuration of the local machine is Mac OS Intel Core i7, 8 core CPU with 16GB of memory. We assign the control plane 4GB of memory and 2 virtual central processing unit (vCPU) and the worker nodes have 3GB of memory and 2 virtual central processing unit each. The virtual central processing unit(vCPU) represents the central processing unit (CPU) of a virtual machine. We deploy the microservice-demo [27] application to demonstrate our attack which has 11 microservices. We use this application as Google uses this application to demonstrate the use of Kubernetes [27]. We edit the Kubernetes manifests of `cartservice` and `checkoutservice` of the microservice-demo by providing root privilege, privilege escalation, host namespaces and container privileges so that we can exploit the misconfigurations to conduct an attack. We assume that the attacker has an "initial access" into the Kubernetes cluster [31]. In each of our attack validation set up, the attacker has an access to a valid Kubeconfig file and uses the kubectl CLI interface to communicate with the Kubernetes API server.

We verify the pod security requirements using NuXMV model checker with our FSM model for a pod. We validate attacks against each of the counter examples generated by NuXMV. We demonstrate 6 attacks which map to the OWASP top 10 Kubernetes security vulnerabilities. We describe the attacks as follows:

$\boxed{\textbf{Attack \#1}}$ **Workload privilege escalation attack:** In this attack, an attacker exploits the unnecessary privilege of a container running inside a pod.

Detection and attack description:

We verify the pod security requirement *"Any container running inside a pod with unnecessary privilege will not be in an unsafe state at runtime"*. We observe a counterexample for this pod security requirement and detect an attack where an attacker can generate a new pod creation request from a running pod.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: The attacker gets access to a privileged pod that starts with missing admission controller misconfiguration [33] [52]. The pod has the privileges: `hostNetwork: true`, `hostIPC: true, hostPID: true, allowPrivilegeEscalation: true`, active hostPath , Capability abuse, privileged `securityContext`, `automountServiceAccountToken: true, readOnlyRootFileSystem: False, runAsNonRoot:False, runAsUser: False`. The pod is associated with the `serviceAccount: default`. The privileges of the pod allows the attacker to escalate pod isolation boundary in the host [66].

Step-2: The attacker can access the underlying host with pod misconfiguration `hostNetwork: true, hostIPC: true, hostPID: true`, privileged `securityContext`, active hostPath and `automountServiceAccountToken: true` to get the service account token of `serviceAccount: default` [66].

Step-3: The attacker has access to the service account token of `serviceAccount: default` as privileged `ServiceAccount` that has privileged role. The attacker uses the privileged `ServiceAccount` which has over-privileged permission as a `cluster-admin` role.

Step-4: The attacker installs `kubectl` tool from the container for misconfiguration `readOnlyRootFileSystem: False`. The attacker can send a request to the Kubernetes API server with the service account token of privileged `ServiceAccount`.

Step-5 The attacker leverages the misconfigurations and creates a new malicious privileged pod with the service account token of privileged `ServiceAccount`. The malicious pod can run as cryptominer to disrupt other running pods. The attacker can also read Kubernetes cluster secrets, modify/delete running pods in the Kubernetes cluster from the malicious pod.

Implication: An attacker can run malicious applications such as a cryptominer and consume excessive resource to disrupt running pods, get secrets from the Kubernetes API server or modify/delete any running pods causing service disruption to the victim organization. Hence, with this "workload privilege escalation attack", an attacker can violate the confidentiality, integrity and availability of the Kubernetes cluster.

Mitigation: We mitigate this attack by eliminating our identified privileged pod misconfigurations: `hostNetwork:true` , `hostIPC:true`, `hostPID:true`, privileged `securityContext`, `allowPrivilegeEscalation:true`, `readOnlyRootFileSystem:False`, `runAsNonRoot:False`, `runAsUser: False`, active hostPath, Capability abuse, so that the attacker can not access the pod to get the root privilege with write permission and get unnecessary access to the host machine. We eliminate any privileged role and role binding to default service account and disabled token mounting in the pod by eliminating the following misconfigurations if available: privileged role, privileged `ServiceAccount`, `automountServiceAccountToken: true`, `serviceAccount: default`.

$\boxed{\textbf{Attack \#2}}$ **Remote code execution attack:** In this attack, an attacker exploits any vulnerabilities in the library, dependencies, container images of the pod deployment manifest.

Detection and attack description:

We verify the pod security requirement *"Any unscanned or misconfigured container image will not lead a pod to an unsafe state"*. We observe a counterexample for this pod security requirement and detect an attack where an attacker can remotely connect to a running pod and create a new pod creation request.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: An attacker creates a malicious pod with an unscanned container image with missing admission controller misconfiguration [33] [52]. The manifest contains a remote code execution vulnerability that can communicate with a remote attacker machine. The pod has the misconfigurations: `hostNetwork: true`, `hostIPC: true`, `hostPID: true`, `allowPrivilegeEscalation: true`, `automountServiceAccountToken: true`, privileged `securityContext`, Capability abuse, `readOnlyRootFileSystem: False, runAsNonRoot:False, runAsUser: False`.

Step-2: The attacker uses the ncat [51] tool and listens to the malicious pod IP and port to connect with the malicious pod to get a remote shell from the remote attacker machine [66].

Step-3: The attacker can access the underlying host with pod misconfiguration `hostNetwork: true, hostIPC: true, hostPID: true, automountServiceAccountToken: true`, privileged `securityContext`, and active hostPath to get the service account token of `serviceAccount: default` [66].

Step-4: The attacker has access to the service account token of `serviceAccount: default` as privileged `ServiceAccount` that has privileged role. The attacker uses the privileged `ServiceAccount` which has over-privileged permission as a `cluster-admin` role.

Step-5: The attacker installs `kubectl` tool from the container for misconfiguration `readOnlyRootFileSystem: False`. The attacker can send a request to the Kubernetes API server with the service account token of privileged `ServiceAccount`.

Step-6 The attacker leverages the misconfiguration missing admission controller and creates a new malicious privileged pod with the service account token of privileged `ServiceAccount`. The malicious pod can run as cryptominer to disrupt other running pods. The attacker can also read Kubernetes cluster secrets, modify/delete running pods in the Kubernetes cluster from the malicious pod.

Implication: An attacker can exploit the remote code execution vulnerability to get access to the Kubernetes cluster and leverage the misconfigurations in pod manifests to create malicious applications such as cryptominers. The attacker can also leak secrets, capture the network communication, and modify/delete pods without getting noticed by the cluster administrator. Hence, this “remote code execution attack” can violate confidentiality, integrity and availability of the Kubernetes cluster.

Mitigation: We mitigate this attack by applying "ValidatingAdmissionPolicy" [41] that checks container image origin and eliminates missing admission controller misconfiguration so that any malicious container with remote code execution vulnerability never gets into Kubernetes cluster. After that, we eliminate our identified privileged pod misconfigurations: `hostNetwork:true` , `hostIPC:true`, `hostPID:true`, privileged `securityContext`, active hostPath, capability abuse, `allowPrivilegeEscalation:true`, `readOnlyRootFileSystem:False`, `runAsNonRoot:False`, `runAsUser: False` so that the attacker can not access the pod to get the root privilege with write permission and get unnecessary access to the host machine. We eliminate any privileged role and role binding to default service account and disabled token mounting in the pod by eliminating the following misconfigurations if available: privileged role, privileged `ServiceAccount`, `automountServiceAccountToken: true`, `serviceAccount: default`.

$\boxed{\textbf{Attack \#3}}$ **RBAC privilege maneuver attack:** In this attack, an attacker can exploit over-privileged RBAC permission in Kubernetes cluster.

Detection and attack description:

We verify the pod security requirement *"An over-privileged RBAC permission in Kubernetes cluster will not lead a pod to an unsafe state."*. We observe a counterexample for this pod security requirement and detect an attack where an attacker can create a new pod creation request from the Kubernetes dashboard using a default service account with over-privileged RBAC permission.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: The attacker has access to the kubernetes dashboard with the service account token of `serviceAccount: default` as privileged default `ServiceAccount` that has `cluster-admin` role to access the Kubernetes dashboard.

Step-2: The attacker can access the Kubernetes dashboard with the `cluster-admin` and privileged default `ServiceAccount`. The attacker leverages the misconfigurations missing admission controller to deploy malicious pod. The attacker can also modify/delete nodes, expose secrets and sensitive applications with the `cluster-admin` privilege.

Implication: An attacker with over-privileged RBAC permission, such as `cluster-admin` privilege can provide an attacker a complete control over the Kubernetes cluster of an organization and violate confidentiality, integrity, availability.

Mitigation: To mitigate the attack, We eliminate any privileged role and role binding to default service account and disabled token mounting in the pod by eliminating the following misconfigurations if available: privileged role, privileged `ServiceAccount`, privileged Default `ServiceAccount, serviceAccount: default, cluster-admin`.

$\boxed{\textbf{Attack \#4}}$ **Denial of service attack:** In this attack, an attacker leverages the lack of centralized policy and missing resource limit specification for pods in Kubernetes cluster. In the attack, attacker deploys the pod without specifying resource limits to create denial of service attack.

Detection and attack description:

We verify pod security requirement *"Lack of centralized policy such as missing admission controller will not lead a pod to an unsafe state."* We observe a counterexample for this pod security requirement and detect an attack where an attacker can cause a denial of service attack from a running pod.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: An attacker creates a malicious pod with an unscanned container image with missing admission controller and missing resource limit misconfigurations [33] [52]. The manifest contains a remote code execution vulnerability that can communicate with a remote attacker machine [66]. The pod has the misconfigurations: `runAsNonRoot:False, runAsUser: False`

Step-2: The attacker uses the ncat [51] tool and listens to the malicious pod IP and port to connect with the malicious pod to get a remote shell from the remote attacker machine [66]

Step-3: The attacker runs a malicious program inside the running container of the pod with `readOnlyRootFileSystem: False`, absent resource limit misconfigurations. The pod consumes the entire CPU and memory limit of the node and disrupts the availability of running pods in the Kubernetes cluster.

Implications: Lack of centralized policy can allow outdated, vulnerable images, dependencies to run as containers. As a result, any attacker can leverage the underlying image vulnerability to cause critical service disruption, such as a resource-related denial of service attack in the Kubernetes cluster. This attack violates the availability of the Kubernetes cluster.

Mitigation: To mitigate the attack, We eliminate missing resource limit, disable root user and enable read only file system in pod configuration. We also recommend using "ValidatingAdmissionPolicy" to eliminate missing admission controller so that any pod without resource limit never gets deployed at the Kubernetes cluster. We eliminate the following misconfigurations if available: missing admission controller, absent resource limit, `runAsUser:False, readOnlyFileSystem:True`.

$\boxed{\textbf{Attack \#5}}$ **Network boundary exploitation:** In this attack, an attacker leverages misconfigurations related to network segmentation and exploits unrestricted sensitive applications.

Detection and attack description:

We verify pod security requirement *"Misconfigurations in network segmentation will not lead a pod to an unsafe state"*. We observe a counterexample for this pod security requirement and detect an attack where an attacker can access a sensitive pod, such as a database, from a malicious pod.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: An attacker creates a malicious pod with an unscanned container image with missing admission controller misconfiguration [33] [52]. The manifest contains a remote code execution vulnerability that can communicate with a remote attacker machine. The pod has the misconfigurations: `hostNetwork: true, readOnlyRootFileSystem: False, namespace: default, runAsNonRoot:False, runAsUser: False`

Step-2: The attacker uses the ncat [51] tool and listens to the malicious pod IP and port to connect with the malicious pod to get a remote shell from the remote attacker machine [66].

Step-3: The attacker again uses the ncat tool using privilege `hostNetwork: true`, missing network policy in `namespace: default` to establish a connection with the Redis database in port 6379.

Step-4: The attacker can update the data in the Redis database as Redis does not require any authentication by default [63], [77]. The attacker can modify or delete sensitive information from the redis database. As a result the pods may get tampered data when required. This attack violates confidentiality, integrity and availability of Kubernetes cluster.

Implications: Any attacker can leverage the network misconfigurations to connect with sensitive applications such as database and get sensitive data or modify existing data. The attacker can also cause a network-related denial of service attack to business-critical applications.

Mitigation: To mitigate this attack, we apply network policy for the sensitive applications such as redis database and avoid using default namespace. We apply non root user with read only permission and eliminate access to host network privilege from the pods as well. We eliminate the following misconfigurations if available: missing network policy, `namespace: default, hostNetwork: True, readOnlyRootFileSystem: False, runAsUser: False`.

Attack #6 **Secret exfiltration attack:** In this attack, an attacker can leverage secret management-related misconfigurations to steal secrets from Kubernetes cluster. By default, the etcd database in the control plane is not encrypted, and access to an unencrypted etcd database can leak sensitive cluster information.

Detection and attack description:

We verify pod security requirement, *"If any secrets are accessible and stored without encryption in Kubernetes cluster, it can lead a pod to an unsafe state"*. We observe a counterexample for this pod security requirement and detect an attack where an attacker can access a pod with a privileged service account, deploy a pod in the control plane node, and extract the unencrypted etcd database contents.

We validate the capability of an attacker to perform the attack with the following steps:

Step-1: The attacker gets access to a privileged pod that starts with missing admission controller misconfiguration [33] [52]. The pod has the privileges: `hostNetwork:true` , `hostIPC:true, hostPID: true, allowPrivilegeEscalation: true`, privileged `securityContext`, `automountServiceAccountToken: true, readOnlyRootFileSystem: False`, active hostPath, Capability abuse, `runAsNonRoot:False, runAsUser: False`. The pod is associated with the `serviceAccount: default`. The privileges of the pod allows the attacker to escalate pod isolation boundary in the host [66].

Step-2: The attacker uses the ncat [51] tool and listens to the malicious pod IP and port to connect with the malicious pod to get a remote shell from the remote attacker machine [66].

Step-3: The attacker can access the underlying host with pod misconfiguration `hostNetwork: true, hostIPC: true, hostPID: true`, active hostPath, privileged `securityContext`, `automountServiceAccountToken: true` to get the service account token of `serviceAccount: default` [66].

Step-4: The attacker installs `kubectl` tool from the container for misconfiguration `readOnlyRootFileSystem: False`. The attacker can send a request to the Kubernetes API server with the service account token of privileged `ServiceAccount`.

Step-4: The attacker creates a privileged pod in the control plane node with misconfigurations missing admission controller, `nodeName:master`. The privileged pod has the permissions: `hostNetwork: true , hostIPC: true, hostPID: true, allowPrivilegeEscalation: true, runAsNonRoot: false`, privileged `securityContext, readOnlyRootFileSystem: False`, Capability abuse and the pod runs in the control-plane node.

Step-5: The attacker installs etcd client and can access the key and certificate for etcd using `hostNetwork: true , hostIPC: true, hostPID: true, allowPrivilegeEscalation: true, runAsNonRoot: false` privileged `securityContext, readOnlyRootFileSystem: False`, and Capability abuse misconfigurations.

Step-6: The attacker can use the key and certificate to access the unencrypted etcd database that contains all cluster secrets and information. This action from the attacker violates the confidentiality of the Kubernetes cluster.

Implication: Any attacker who has access to Kubernetes API server can extract unencrypted cluster secrets such as database credentials and cluster information and violate confidentiality of the Kubernetes cluster.

Mitigation: To mitigate the attack, we avoid the configuration `nodeName:master` so that the pods are not deployed in the control-plane node. We also eliminate the following misconfiguration if present in the pod configurations: `hostNetwork:true , hostIPC:true, hostPID:true, privileged:true, allowPrivilegeEscalation:true`, active hostPath, Capability abuse, `readOnlyRootFileSystem:False, runAsNonRoot:False, runAsUser: False` privileged role, privileged `ServiceAccount` and missing admission controller.

In Table 5.11, we list our attacks, configuration sequences and dependencies for insecure pod provisioning.

Table 5.11: Misconfigurations that invoke insecure provisioning

Attack Name	Configuration Sequence	Dependencies

Workload privilege escalation attack	`missing admission controller --> (hostIPC:True, hostPID:True, hostNetwork:True, active hostPath, allowPrivilegeEscalation:True, Capability abuse, privileged securityContext) --> (serviceAccount: default,runAsNonRoot:False, runAsUser: False, readOnlyRootFileSystem: False, automountServiceAccountToken: True, privileged role) --> privileged ServiceAccount, missing admission controller`	`kubectl exec checkoutservice /bin/sh`
		`serviceaccount -> ca.crt namespace token`
		`wget`
		`kubectl`

Remote code execution attack	`missing admission controller --> (hostIPC: True, hostPID:True, hostNetwork:True, allowPrivilegeEscalation: True, Capability abuse, privileged securityContext) --> readOnlyRootFileSystem: False, runAsNonRoot:False, runAsUser: False, serviceAccount: default, automountServiceAccountToken: True, privileged role --> privileged ServiceAccount, missing admission controller`	`ncat -vlp <PORT>`
		`revshell-pod.yaml`
		`serviceaccount -> ca.crt namespace token`
		`apt-get`
		`curl`
		`kubectl`

RBAC privilege maneuver Attack	`serviceAccount: default, automountServiceAccountToken: True, runAsNonRoot:False, runAsUser: False, --> privileged default ServiceAccount, cluster-admin, missing admission controller`	`kubernetes dashboard access`
		`serviceaccount -> ca.crt namespace token`
		`dashboard service account`

Denial of Service Attack	`missing admission controller, absent resource limit --> readOnlyRootFileSystem: False, runAsNonRoot: False`	`ncat -vlp <PORT>`
		`revshell-pod.yaml`
		`DoS-Daemonset.yaml`

Network boundary exploitation	`missing network policy, missing admission controller --> (hostNetwork:True, runAsNonRoot:False, runAsUser:False) --> namespace: default`	`revshell.yaml`
		`ncat -vlp <PORT>`
		`ncat -vn redis:6379`

Secret exfiltration attack	`missing admission controller -->(hostIPC: True, hostPID:True, hostNetwork:True, allowPrivilegeEscalation: True, Capability abuse, privileged securityContext) --> readOnlyRootFileSystem: False, automountServiceAccountToken: False, runAsNonRoot:False, runAsUser: False, --> (serviceAccount: default, privileged ServiceAccount)`	`revshell.yaml`
		`ncat -vlp <PORT>`
		`apt-get`
		`curl`
		`kubectl`
		`serviceaccount -> ca.crt namespace token`
		`nodeselector-controlplane`
		`hack-control plane.yaml`
		`wget etcd`
		`/etc/kubernetes/pki/etcd/ca.crt,`
		`healthcheck-client.crt`
		`etcdctl IP:2379 ca.crt healthcheck-client.crt`

Chapter 6

Authentic Learning for Learning Kubernetes Security Misconfiguration Analysis

As per the 2021 CNCF annual survey, 96% of the 19,000 practitioners surveyed are either using or evaluating Kubernetes for their respective organizations [34]. Furthermore, the survey highlights that approximately 5.6 million developers globally are utilizing Kubernetes [34]. However, practitioners also acknowledge that Kubernetes has evolved into a complex software platform with a steep learning curve, emphasizing the need for a skilled workforce proficient in Kubernetes. [35] [30]. According to the Redhat survey conducted in 2021, 94% of practitioners reported experiencing incidents related to Kubernetes security misconfigurations [62]. To address this issue and cultivate a more skilled cybersecurity workforce in the industry with expertise in Kubernetes, one potential solution is to educate students on Kubernetes security misconfigurations

Previous research has demonstrated that authentic learning exercises have proven effective in enhancing students' understanding of various subjects, such as mobile application security [54] and infrastructure-as-code (IaC) [58]. Building upon this, we formulate the hypothesis that an authentic learning-based exercise will help students in comprehending Kubernetes security misconfigurations.

We answer the following research questions:

- RQ 5.1 How to design authentic learning based exercise to help students for secure development of Kubernetes Manifests?
- RQ 5.2 How does authentic learning help students to learn about secure development of Kubernetes Manifests?

6.1 Background on Authentic Learning

Authentic learning is recognized as an instructional approach that prioritizes the engagement of students in problem-based activities that reflect real-world contexts [44]. When implementing authentic learning, curriculum exercises exhibit distinct characteristics that contribute to its effectiveness. These characteristics include [45]: (i) it focuses on hands-on exercises relevant to the real-world problems, (ii) it encourages students to have a diverse set of perspectives for the same exercise, and (iii) it utilizes available resources to solve exercises.

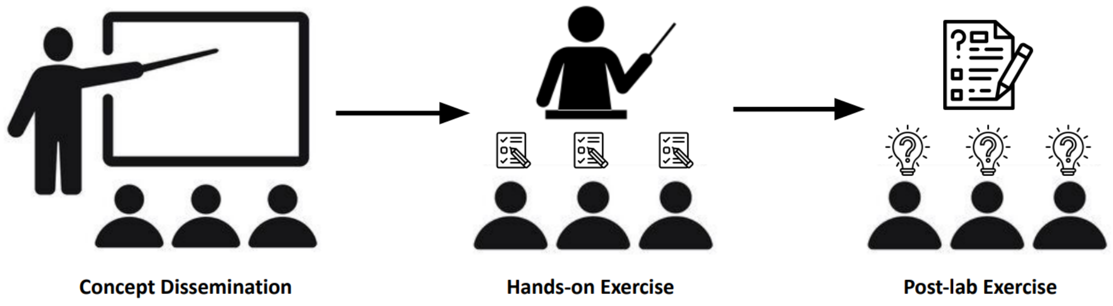


Figure 6.1: The diagram illustrates the three distinct steps of the authentic learning-based exercise, encompassing pre-lab content dissemination, hands-on exercise and active learning, and post-lab exercise with real-world scenarios.

The implementation of an authentic learning-based exercise typically involves three distinct steps.These steps are outlined as follows:
In this initial step, the instructor introduces the students to the fundamental concepts related to the topic at hand. Through various teaching methods, such as lectures or presentations, the instructor imparts the necessary theoretical knowledge and background information to the students. This pre-lab content dissemination phase sets the foundation for the subsequent hands-on exercises. The second step of the authentic learning-based exercise involves providing students with hands-on exercises that are directly relevant to the real-world application of the subject matter. Through active learning strategies, students engage directly with the material and apply their theoretical knowledge in practical scenarios. The instructor guides and supports the students during this hands-on exercise phase, facilitating their

learning and understanding of the subject matter through active participation. Following the completion of the hands-on exercise, the authentic learning-based exercise progresses to the post-lab exercise stage. In this phase, the instructor presents the students with exercises based on real-world scenarios that reinforce and deepen their understanding of the subject matter. These exercises challenge students to apply their acquired knowledge and skills to solve complex problems or address practical challenges. By working through these real-world scenarios, students develop a more comprehensive understanding of the subject matter and enhance their problem-solving abilities in authentic contexts. In Figure 6.1, we have demonstrated three steps of authentic learning steps. The inclusion of these three steps in the authentic learning-based exercise ensures a holistic and practical learning experience for students, promoting deeper engagement and mastery of the subject matter. The three steps are as follows:

- Step 1: Pre-Lab Content Dissemination
- Step 2: Hands-On Exercise
- Step 3: Post-Lab Exercise on Real-World Scenarios

Authentic learning and experiential learning are two distinct instructional approaches that are often compared in the context of education. While authentic learning emphasizes real-world problem-solving activities, experiential learning follows a four-phase model consisting of design, conduct, evaluation, and feedback [25].

In experiential learning, the instructor plays a pivotal role in creating a structured and supportive environment for students throughout the design and conduct phases [25]. The learning experience is carefully designed to facilitate active engagement and experiential opportunities [25]. Subsequently, in the evaluation phase, the instructor assesses the specific learning outcomes achieved through the experience, followed by providing feedback to the students [25].

In contrast, authentic learning focuses on exposing students to real-world problem-solving scenarios based on their in-class experiences. By engaging in authentic tasks, students have the opportunity to develop and refine both soft and hard employable skills that are aligned with market demands [53]. Prior research has successfully integrated authentic learning-based exercises into various domains, such as secure software development in mobile computing [54], resulting in improved self-efficacy and confidence among students. Authentic learning has also been applied to enhance learning in secure infrastructure-as-code (IaC) development, enabling students to gain insights into secure IaC practices [58].

In our research, we choose to adopt the authentic learning approach instead of experiential learning to prepare a highly employable cybersecurity workforce with expertise in Kubernetes security misconfiguration analysis. By utilizing authentic learning, we aim to provide students with practical, hands-on experiences in addressing real-world challenges in Kubernetes security. This approach aligns with our objective of equipping students with the necessary skills and knowledge to meet the demands of the industry in the field of Kubernetes security.

6.2 Authentic Learning-based Exercise Design

We designed our authentic learning-based exercise for the Kubernetes security misconfiguration and deployed it in "Software Quality Assurance" course in the fall 2022 semester at Auburn University. After collecting the feedback from the students, we re-deploy our authentic learning-based exercise into two different classes at Auburn University and Tuskegee University in the spring 2023 semester, respectively. We construct the three steps of our authentic learning-based exercise as follows:

6.2.1 Concept Dissemination

In the class, we introduce students to containers and tools to automate the management of containers. We specifically focus on one container management and orchestration tool, Kubernetes. Practitioners use configuration files known as manifests to deploy containers into the Kubernetes cluster. We introduce the students to security misconfigurations and the use of static analysis tools to identify Kubernetes security misconfigurations.

6.2.2 Hands-on Exercise

In the hands-on exercise, we instructed students to install Docker on their computers. We introduce the students to an open-source tool called SLIKUBE [3]. We conduct a live demonstration for the students on using SLIKUBE to detect security misconfigurations in Kubernetes. We demonstrate how to download the tool from DockerHub [20] and instructed them to run it inside the Docker container. We explain to the students the detailed output of the SLIKUBE presented as a CSV file, such as the directory column, the path of the manifests column, specific misconfiguration columns, and the total column that reports the total occurrences of misconfiguration in Kubernetes manifest.

6.2.3 Post-lab Exercise

In the post-lab exercise, we provide students Kubernetes manifests from open-source repositories(OSS) such as GitHub and GitLab. We ask the students to run the SLIKUBE tool on the provided Kubernetes manifests. After running the SLIKUBE on provided manifests, we ask the students to analyze the output of SLIKUBE and report the top 3 most frequent Kubernetes misconfigurations with a description. Moreover, we ask students to complete a survey on the authentic learning-based exercise.

6.2.4 Survey Design

We deploy the survey using the online Qualtrics platom. We design our survey questions related to the background of the participating students and the experience on the authentic learning-based exercise.

Question Related to Student Background

As part of our study, we administered a questionnaire consisting of three questions aimed at assessing the background of the students in the class. These questions aimed to gather information regarding the students' prior experience in cybersecurity, software quality assurance activities, and program analysis tools before participating in the workshop exercise. The specific questions were as follows:

(i) How would you rate your experience in cybersecurity prior to the workshop? (ii) How would you rate your experience in software quality assurance activities prior to the workshop? (iii) How would you rate your experience with program analysis tools prior to the workshop?

We follow the recommendations of Kitchenham [38] and create a five-item Likert scale as follows: 'Expert', 'Somewhat Expert', 'Knowledgable', 'Little knowledge' and 'No knowledge'.

Question Related to Usefulness of Authentic Learning-based Exercise:

As part of our evaluation process, we administered a questionnaire to the students, consisting of six questions that aimed to assess the perceived usefulness of our authentic learning-based exercise. Additionally, we included one question specifically targeting the usefulness of the workshop in facilitating learning about Kubernetes misconfigurations. The questions and response options were as follows:

(i) Which part of the authentic learning experience was useful for you? - Pre-stage (ii) Which part of the authentic learning experience was useful for you? - In-class experience (iii) Which part of the authentic learning experience was useful for you? - Post-stage (iv) Which part of the authentic learning experience was useful for you? - Pre-stage and in-class experience (v) Which part of the authentic learning experience was useful for you? - Pre-stage and post-stage (vi) Which part of the authentic learning experience was useful for you? - All three steps

We ask the participating students to rate the usefulness of each aspect of the authentic learning experience using a five-item Likert scale, with response options ranging from 'Extremely useful', 'Useful', 'Moderately useful', 'Little useful', to 'Not at all useful'.

Furthermore, we included a question specifically addressing the impact of the workshop on learning about Kubernetes misconfigurations. The question was as follows:

Did the workshop help you to learn about Kubernetes misconfigurations?

To assess the perceived helpfulness of the workshop, we employed a five-item Likert scale with response options including 'Very helpful', 'Helpful', 'Somewhat helpful', 'Little helpful', and 'Not at all helpful'.

6.3 Results

During the fall 2022 semester, we conducted data collection by administering a survey to students enrolled in the "Software Quality Assurance" course at Auburn University. A total of 51 responses were collected from the students, providing valuable feedback on the authentic learning-based exercise. Based on the feedback received in the fall 2022 semester, we proceeded to redeploy the authentic learning-based exercise in subsequent semesters at both Auburn University and Tuskegee University during the spring of 2023. In the spring 2023 semester, a total of 76 responses were collected. Among these, 66 responses were from students at Auburn University, while the remaining 10 responses were from students at Tuskegee University.

To provide further insights into the survey participants, Figure 6.2 showcases the distribution of students based on their educational background during their participation in the survey. Notably, we observed that 83% of the students identified themselves as undergraduate seniors, while 15% were graduate master's students. Furthermore, we noticed

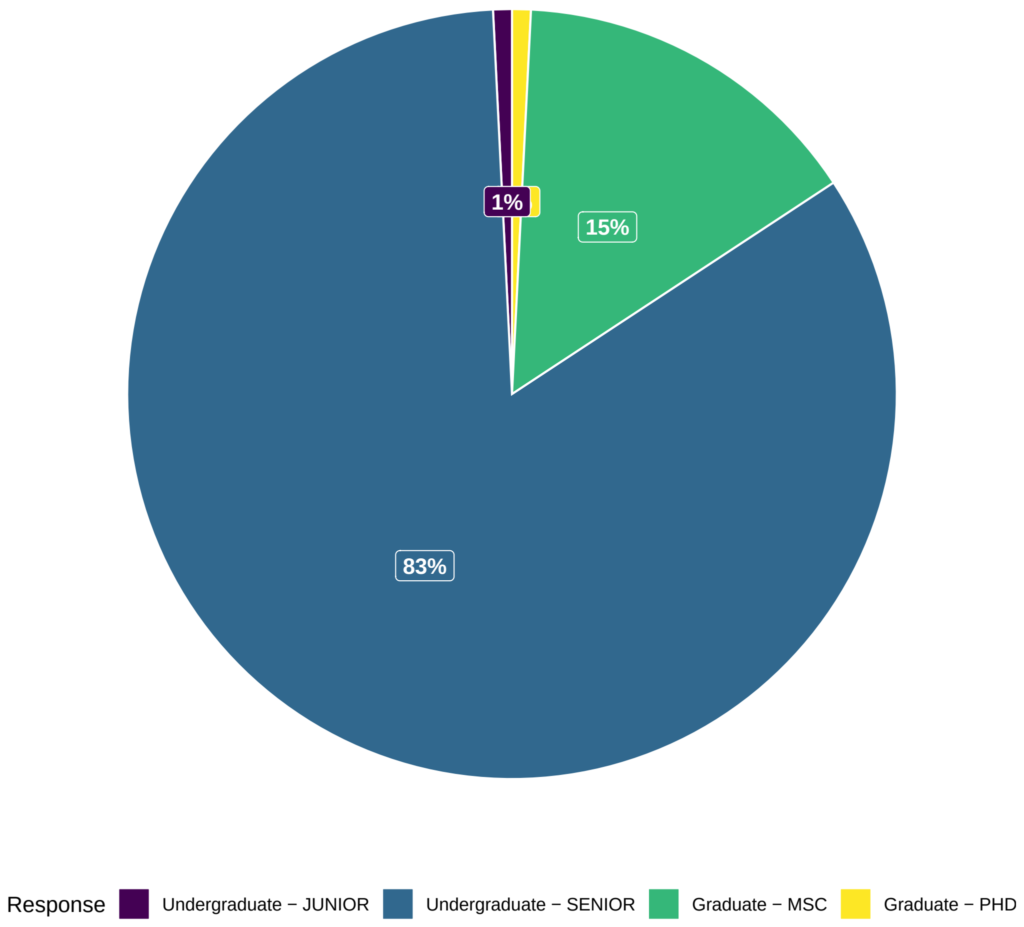


Figure 6.2: Educational Background of Students Participating in the Authentic Learning-based Exercise

that a minority of the students, comprising only 2% of the total, consisted of graduate PhD students and junior undergraduates.

6.3.1 RQ 5.1 How to design authentic learning based exercise to help students for secure development of Kubernetes Manifests?

We present the findings regarding the perception of students based on their diverse backgrounds, including their educational level, expertise in software quality assurance, expertise in software security, and expertise in static analysis tools. We summarize the findings in Figure 6.3, Figure 6.4, Figure 6.5 and Figure 6.6, respectively.

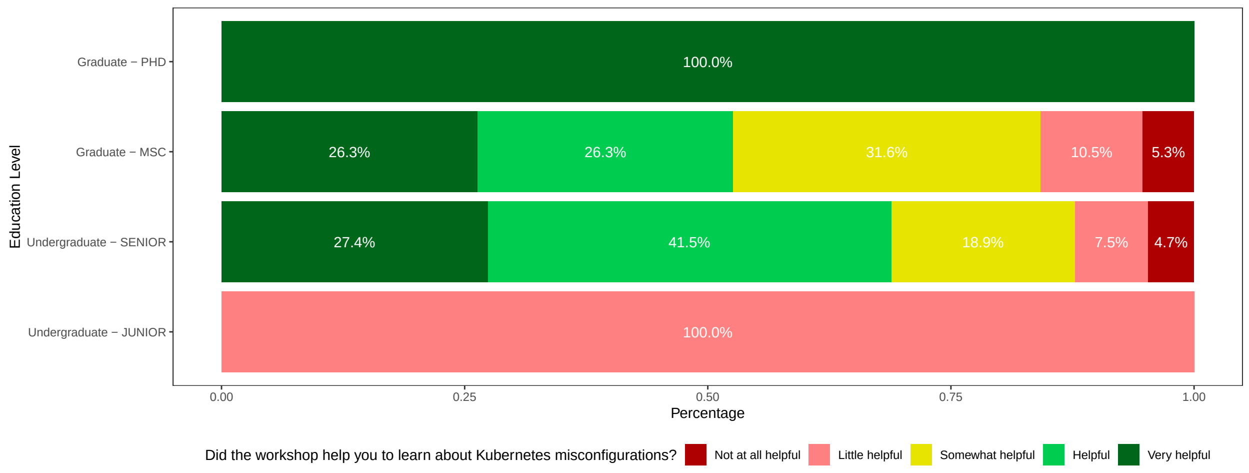


Figure 6.3: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Misconfigurations Based on Their Educational Background

Upon analysis, we observed that a significant majority of undergraduate senior students (87.8%), graduate master's students (86.2%), and graduate PhD students (100%) found our designed authentic learning-based exercise to be helpful in learning about Kubernetes misconfigurations. However, it is noteworthy that all undergraduate students reported finding the exercise to be of little help. In Figure 6.3, we provide an overview of the students perception on our authentic learning-based exercise based on their education level.

In the fall 2022 semester, we collected feedback from the students and carefully addressed their suggestions and concerns. We made necessary modifications to our exercise to enhance its effectiveness before redeploying it in the spring 2023 semester.

These findings demonstrate the importance of considering students' diverse backgrounds and educational levels when designing and implementing authentic learning-based exercises. The results highlight the positive impact of our authentic learning-based exercise on students with higher educational levels and expertise in software quality assurance, software security, and static analysis tools.

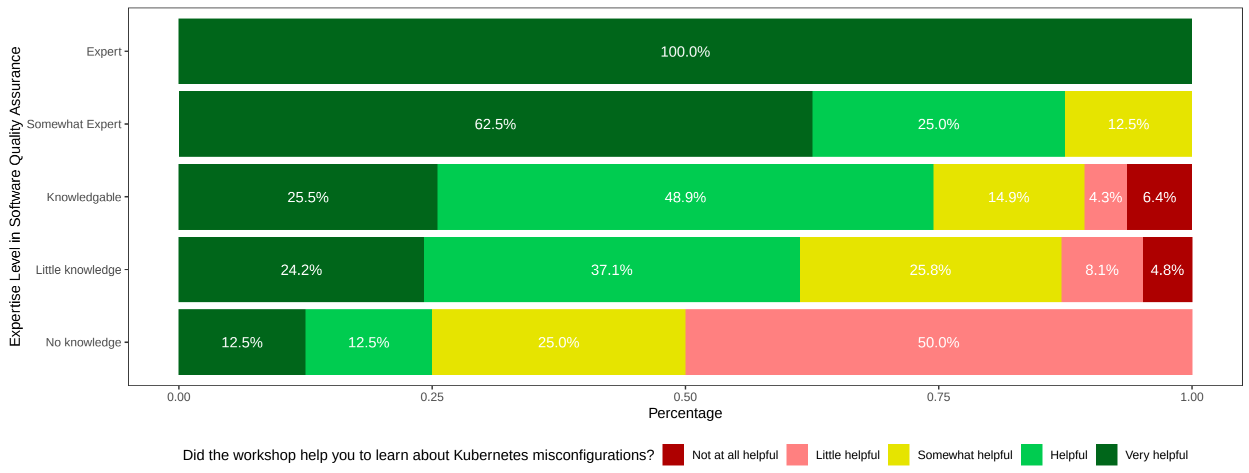


Figure 6.4: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Misconfigurations Based on Their Expertise in Software Quality Assurance

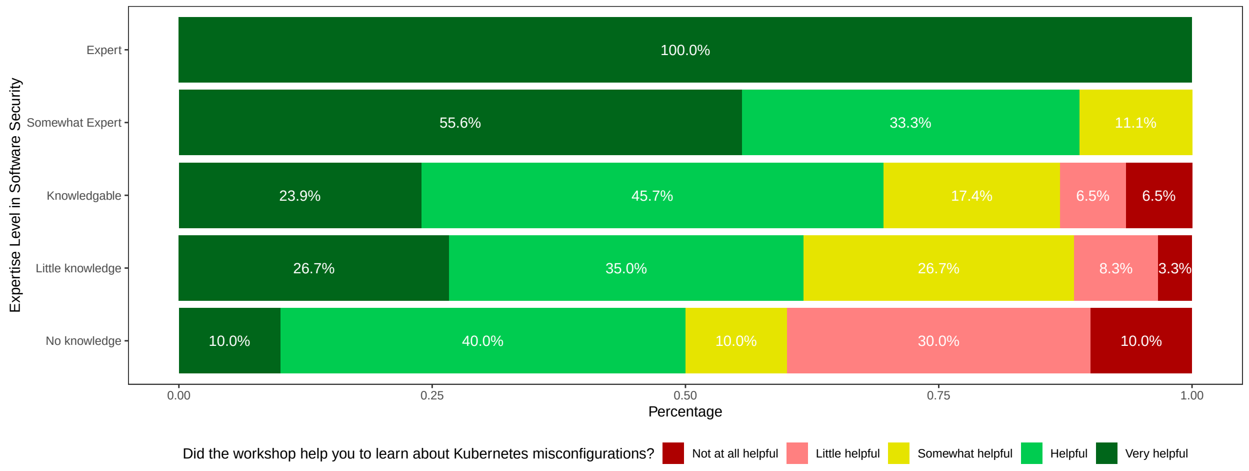


Figure 6.5: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Misconfigurations Based on Their Expertise in Security

We observe that the students who has prior expertise in software quality analysis, software security and static analysis report the exercises are helpful for them to learn about Kubernetes security misconfigurations. We find that the students who evaluates themselves as "Expert" and "Somewhat Expert", have learned better compared to other students in the class. One potential reason can be the lack of adequate technical background to follow through the hands on exercise and perform post lab exercise. For instance, one student reports that *"I could not get docker to install correctly. I don't know why, but with many*

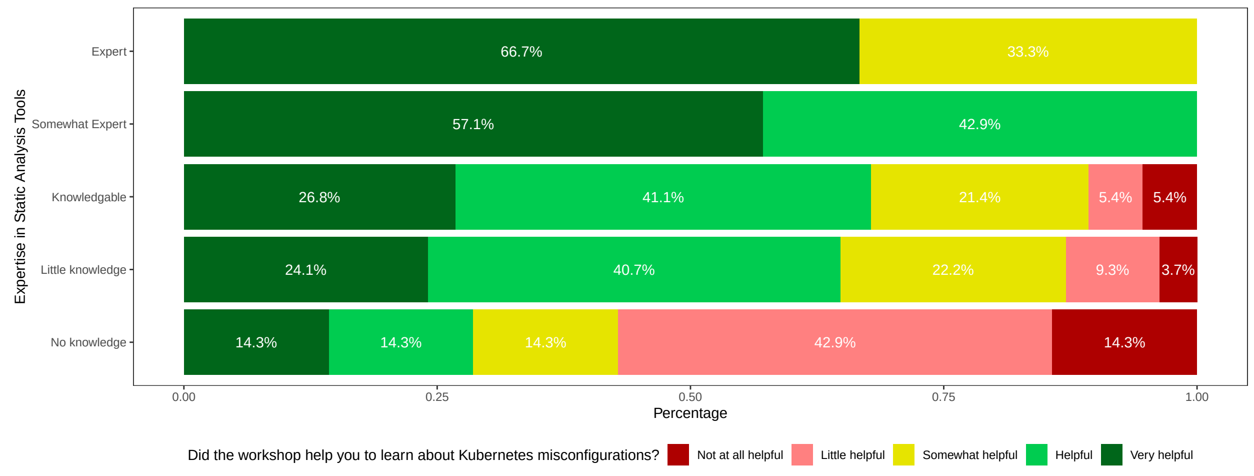


Figure 6.6: Reported Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Misconfigurations Based on Their Expertise in Static Analysis Tools

of the recent workshops, even if I follow the instructions one-to-one, my laptop just doesn't agree with the software. It is pretty frustrating to not even be able to start these workshops." Another student reports that, *"This workshop was very informative, and using docker was a plus because it is useful in industry."*. Our results suggest that our designed workshop exercise is more suitable for students with prior relevant technical background. We integrate prerequisite technical background concept dissemination, more detailed installation instruction in the workshop to make our authentic learning-based exercise more usable for students with little to no background on security, static analysis and software quality assurance.

6.3.2 RQ 5.2 How does authentic learning help students to learn about secure development of Kubernetes Manifests?

In Figure 6.7, we report the students response on the usefulness of the authentic learning-based exercise steps. We observe that 27.8%, and 45.7% of the students find all the three steps “Extremely useful” and “Useful” respectively. We notice that only 3.1% of the students report that the all 3 steps of our designed authentic learning-based exercise is “Not useful at all”.

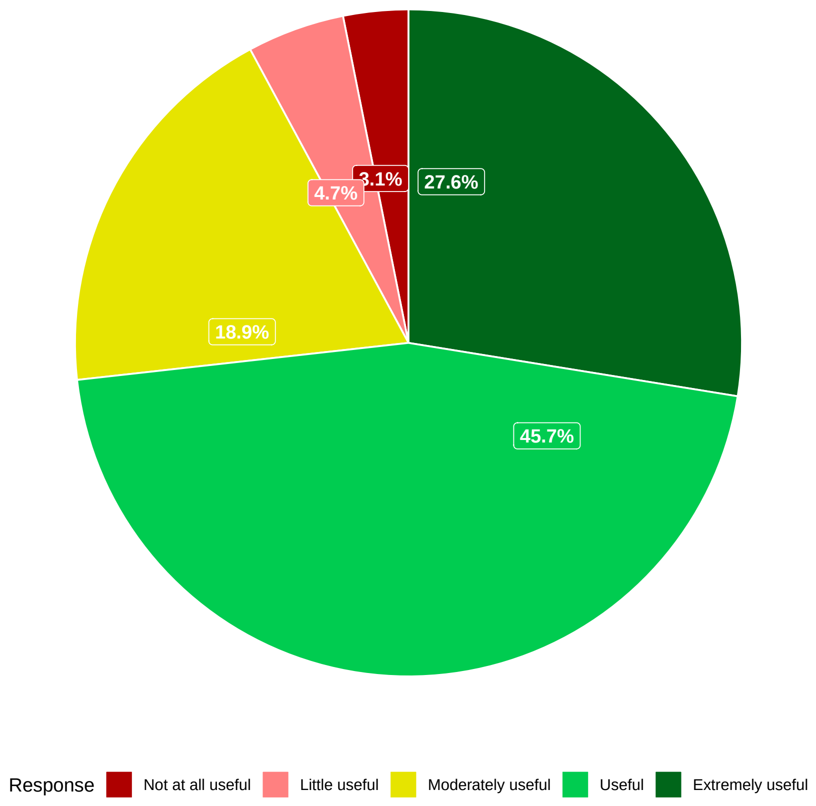


Figure 6.7: Reported Perception of Students on usefulness of all three steps in the authentic learning-based exercise

We also report the overall perception among all the students in the class on how this authentic learning-based exercise help them understand Kubernetes security misconfiguration. We find that 27.6%, 38.6% , 20.5% of the students report that they find the authentic learning-based exercise “Very helpful”, “Helpful” and “Somewhat helpful” respectively. We find 8.7% and 4.7% students report the exercise is “Little helpful” and “Not at all helpful”. One student reports that, “I really liked learning about Kubernetes; it’s something that the industry uses a lot but isn’t taught in school.” Based on the overall feedback and response from the students suggest that our designed authentic learning-based exercise helps students in understanding the Kubernetes security misconfigurations.

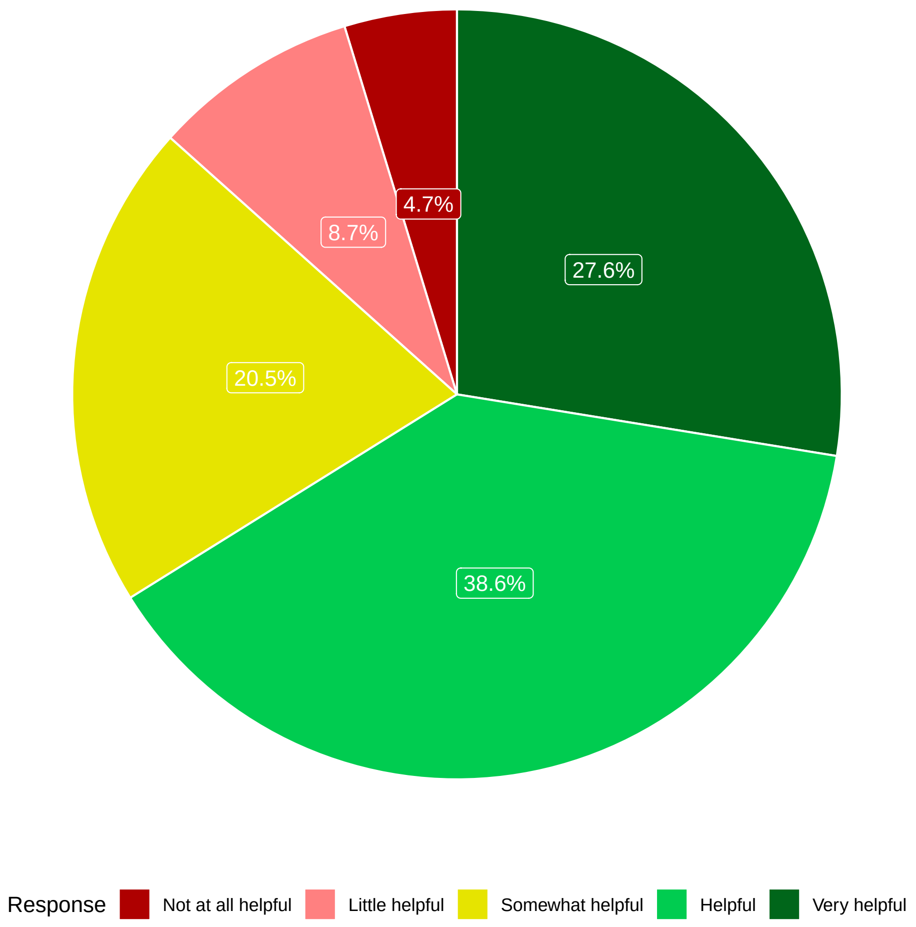


Figure 6.8: Overall Perception of Students on the Authentic Learning-based Exercise to Learn Kubernetes Misconfigurations

Chapter 7

Discussion

7.1 Implication for Practitioners

7.1.1 Application of Kubernetes Security Best Practices

Kubernetes provides utilities for users to manage containers at scale. However, our description of the 11 practices in Section 3.3 shows that effective and secure usage of Kubernetes requires the implementation of security practices applicable for multiple components within the Kubernetes installations: containers, pods, 'etcd' database etc. Applying the 11 practices mentioned above in Section 3.3 also needs a deep understanding of Kubernetes components and configurations. Our discussion in Section 3.3 can be helpful in two ways: *first*, understand the components where security practices are applicable. *Second*, practitioners who already have Kubernetes in place can use our identified practices as a benchmark and compare their usage of practices.

7.1.2 Application of Security Static Analysis

SLIKUBE+ extends SLIKUBE with 12 additional security misconfigurations. We recommend practitioners use our security static analysis tool SLIKUBE+ to perform regular scanning to avoid the propagation of misconfigurations.

7.1.3 Better Understanding of Misconfiguration Consequences

We verify the pod security requirements using the NuXMV model checker and identify known attacks from the counterexamples. The practitioner can use our research to have a better understanding of the consequences of misconfigurations and how the misconfigurations can pose a threat to the overall security posture of an organization. We also provide mitigation strategies to prevent known attacks by fixing the misconfigurations detected by our SLIKUBE+ tool.

7.2 Implication for Researchers

7.2.1 Baseline for Future Research

Our discussion in Section 2.2 shows that Kubernetes security to be an under-explored research area. Our derived list of security practices can provide the groundwork for Kubernetes security research.

7.2.2 Automated Framework for Generating Attack-akin Configurations

We describe the existing challenges in constructing finite state machines for Kubernetes and discuss the possible ways for future researchers to address the challenges.

Advancing Context-Aware Models

Our approach involves creating a finite state machine for a pod in Kubernetes by harnessing knowledge extracted from various Internet artifacts and the official Kubernetes documentation. As a result, the state of the finite state machine in our model is influenced by specific pod security requirements and assumptions of the model designer. In the past, researchers have extracted finite state machines from component interactions by instrumenting conformance tests to detect logical vulnerabilities in cellular network protocols [37].

Presently, Kubernetes requires approximately 380 conformance tests for all its distributions from various vendors, making them essential components of the system [43]. Researchers can now employ source code level instrumentation through annotations in Kubernetes conformance tests to generate information-rich logs. These logs can help identify the state of a Kubernetes object, corresponding actions, and the values of function parameters, thus enabling the detection of state transitions.

Utilizing instrumentation in conformance tests will help researchers in generate information-rich logs for all the conformance tests. From the logs, researchers can identify the interaction between the components in Kubernetes and build semantically meaningful models. Such models can play a crucial role in identifying misconfiguration-related vulnerabilities and logical vulnerabilities present in Kubernetes.

Enhancing Pod Security Specification

In our research, we utilize OWASP's top 10 Kubernetes security risks to establish pod security requirements and translate them into propositional logic formulas for the NuXMV model checker. However, we encounter a limitation due to the lack of comprehensive pod security-related specifications. Currently, we rely on 10 pod security requirements derived from OWASP Kubernetes top 10 security risks. To improve this, there is a need for additional Kubernetes pod security specifications from external vendors or official Kubernetes security special interest groups. While resources like CIS benchmarks, NSA Kubernetes hardening guide, and PCI-DSS container orchestration guidelines provide valuable guidelines, they cannot directly serve as pod security requirements for translation into propositional logic formulas. Future researchers have an opportunity to create a comprehensive Kubernetes security specification, enabling the verification of these specifications against the Kubernetes pod finite state machine model and exploration of more known attacks.

Extending Verification for Enhancing Completeness

In our research, we have developed a finite state model based on existing knowledge extracted from the official Kubernetes documentation and various Internet artifacts. Our model abstracts a running Kubernetes cluster in the context of its individual components. It represents the abstraction of the pod life cycle, its phases, the containers within each pod, and their states. Prior research has demonstrated that achieving soundness and completeness for parameterized verification problems is generally undecidable [4] [36].

In constructing the pod finite state model, we primarily focused on soundness rather than completeness. We have ensured that our FSM model is sound and does not generate false positives. Hence, whenever our model checker identifies a pod security requirement violation, it is indeed valid. However, our approach is not complete, as it cannot detect all possible violations. We have only extracted the necessary information from the documentation and internet artifacts to construct state machines. To improve the completeness of our approach, researchers can incorporate Kubernetes conformance testing to cover more interaction among the Kubernetes components. The coverage of tests will enhance the completeness of our approach, as it can detect more possible violations.

Utilizing Isolation Boundaries for Kubernetes Entities

Kubernetes is a complex software system with various isolation boundaries. The isolation boundary can be defined as the separation between the entities in a system environment that protects each entity from threats from other entities in the system. Isolation boundary can be specified as machine-level, process or component-level, and trust boundary level. For instance, a pod running with a misconfiguration `hostIPC: true` may put the other pods at a security risk if they run in the same worker node rather than a different worker node. If an attacker get an access to the misconfigured pod with remote code execution then the attack path to compromise the pods in the same worker node will be different than the pods running in the other working nodes in the cluster. In our research, we did not explicitly define the isolation boundary to establish the threat model against a motivated attacker and extracting the attack context from the counterexamples.

To address this, future researchers can design the model to specify machine-level isolations, such as those for the scheduler, controller, and API server residing in control plane nodes, as well as Kubelet, Kube-proxy, and user-specified pods running in worker nodes.

Additionally, researchers can include the trust boundary of components within their model design consideration. For example, a pod and the containers inside it are within the same trust boundary. By defining isolation boundaries, researchers can better identify the attack context from the counterexamples.

7.3 Implication for Educators

Our result in Section 6.3.1 suggests that students with prior background in security, software quality assurance and static analysis tools find the exercise helpful for them to learn about the Kubernetes security misconfigurations. We also observe that the undergraduate junior students find the exercise as “Little helpful” whereas the PhD students find the exercise “Very helpful”. Kubernetes is a complex software and requires a prior technical background in relevant technologies such as Docker. Future cyber security educators or trainers should design the exercise so that students with little background in this domain can participate in the hands-on exercise comfortably.

7.4 Threats to Validity

In this Section, We describe the limitations of our research work.

7.4.1 Conclusion Validity

The identified security best practices described in Section 3.3 can be susceptible to biases of the rater who identified the practices by applying open coding. We mitigate this limitation by allocating another rater who applied closed coding. The 12 additional misconfiguration categories of SLIKUBE+ described in Section 5.3 may return false positives if it is evaluated on the proprietary dataset. Our construction of a finite state machine to abstract Kubernetes cluster is dependent on pod phases, container states, pod status and their relationships described in the Internet artifacts. Hence the transition relations may result in an unrealistic attack path with false positive counter-examples. We mitigated this limitation by validating each of the counter-examples with attack associated OWASP security vulnerability. Our evaluation result in Section 6.3.2 of the survey consists of 127 members and the background of the students may create a bias. We mitigate this bias by deploying the authentic learning module into 3 courses in 2 universities in fall 2022 and spring 2023 semester.

7.4.2 Construct Validity

Our identified categories in Section 3.3 are susceptible to experimenter bias in which author's professional experience can influence the category results. We list the pod properties in Table 5.1. To collect the pod properties, we systematically curating Internet artifacts to identify pod properties related to pod security requirements in Section 5.1.2. The list of pod properties in Table 5.1 can be susceptible to author's bias.

7.4.3 External Validity

Our identified security best practices in Section 3.3 might not be generalizable as we might have excluded practices unique to the proprietary domains, and not discussed publicly in Internet artifacts. The evaluation of SLIKUBE+ is limited to our dataset described in Table 5.7. The validation of attacks in the `kubeadm` cluster may not be generalizable and replicable in all other distribution of Kubernetes as the Kubernetes version and relevant components change very frequently. Results in Section 6.3.2, may need to be more generalizable as we did not survey the students who are enrolled in other courses where Kubernetes security is taught.

7.4.4 Internal Validity

We acknowledge that the our Internet artifact search process describe in Section 3.2 and Section 5.1 are not comprehensive. We also acknowledge that the limited number of manifests we consider for constructing SLIKUBE+ rules can impact the credibility of our tool.

Chapter 8

Conclusion

Kubernetes has become the go-to tool for implementing the practice of automated container orchestration. While Kubernetes has yielded benefits for IT organizations, security misconfigurations can make Kubernetes-based software deployments susceptible to security attacks.

To help practitioners secure their Kubernetes cluster, systematization of knowledge related to practitioner-reported practices helps practitioners to secure Kubernetes installations. We conduct a qualitative analysis of 104 Internet artifacts, such as blog posts, to identify 11 security best practices for Kubernetes. To help the practitioners to have a proper understanding of the consequences of security misconfigurations, we build a finite state model based on the interaction of pod life cycle and container states. We extract pod properties by conducting the semi-structured interview and systematically analyzing Internet artifacts related to pod requirements. We use a NuXMV model checker to verify the 10 pod security requirements constructed from OWASP's top 10 vulnerabilities and validate 6 sequences of actions that lead to an attack. We extend an open-source tool to construct a new static analysis tool called SLIKUBE+ and identify 23 security misconfigurations.

We construct an authentic learning module for Kubernetes security misconfigurations and a survey to collect feedback. We deploy the authentic learning module into 2 different universities in 3 different courses in 2 semesters. We evaluate the feedback from the students and observe 73.3% of students find the exercise module to be "Extremely useful" and "Useful". Overall 66.2% of students report that authentic learning is 'Very helpful" and "Helpful" to learning about Kubernetes security misconfigurations.

We discuss the limitations of the dissertation in Section 7.4. Furthermore, we provide the implication for practitioners, opportunities for future researchers to build upon our work and implication for educators in Section 7.1, Section 7.2 and Section 7.3, respectively. Our work will help practitioners in adopting security best practices. Moreover, practitioners will understand how the misconfigurations in Kubernetes manifests can make the Kubernetes environment susceptible to known attacks. We expect our research will help the researchers further advance the science of secure Kubernetes manifest development.

Bibliography

[1] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 970–973, 2018.

[2] M. Ahmadvand, A. Pretschner, K. Ball, and D. Eyring. Integrity protection against insiders in microservice-based infrastructures: From threats to a security framework. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 573–588. Springer, 2018.

[3] akondrahman. akondrahman/sli-kube, 2022.

[4] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[5] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008.

[6] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef. Leveraging the serverless architecture for securing linux containers. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 401–404. IEEE, 2017.

[7] D. B. Bose, A. Rahman, and S. I. Shamim. 'under-reported' security defects in kubernetes manifests. In *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, pages 9–12. IEEE, 2021.

[8] E. A. Brewer. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 167, New York, NY, USA, 2015. Association for Computing Machinery.

[9] bridgecrew. checkov. `https://www.checkov.io/4.Integrations/Kubernetes.html`, 2022. [Online; accessed 12-May-2022].

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, Jan. 2016.

[11] Canonical. Kubernetes and cloud native operations report 2021, 2021.

[12] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuxmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.

[13] C. Chang, S. Yang, E. Yeh, P. Lin, and J. Jeng. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, 2017.

[14] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, et al. *Handbook of model checking*, volume 10. Springer, 2018.

[15] CNCF. With kubernetes, the u.s. department of defense is enabling devsecops on f-16s and battleships, 2020.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT press, 2022.

[17] B. F. Crabtree and W. L. Miller. *Doing qualitative research.* sage publications, 1999.

[18] datree. datree. `https://hub.datree.io/built-in-rules#containers`, 2022. [Online; accessed 14-May-2022].

[19] dghubble. dghubble/go-twitter. `https://github.com/dghubble/go-twitter`, 2022. [Online; accessed 12-Jan-2022].

[20] Docker. Daemon socket option. `https://docs.docker.com/engine/reference/commandline/dockerd/`, 2022. [Online; accessed 19-Jan-2022].

[21] J. Flanigan. Zero trust network model. *Tufts University: Medford, MA, USA*, 2018.

[22] V. Garousi, M. Felderer, and T. Hacaloğlu. Software test maturity assessment and test process improvement: A multivocal literature review. *Information and Software Technology*, 85:16 – 42, 2017.

[23] V. Garousi, M. Felderer, and M. V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121, 2019.

[24] V. Garousi and B. Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, 2018.

[25] J. W. Gentry. What is experiential learning. *Guide to business gaming and experiential learning*, 9:20, 1990.

[26] R. L. Glass. *Software Creativity 2.0.* developer.* Books, 2006.

[27] Google. microservice-demo. `https://github.com/GoogleCloudPlatform/microservices-demo`, 2023. [Online; accessed 10-Apr-2023].

[28] S. Hariri and M. C. Kind. Batch and online anomaly detection for scientific applications in a kubernetes environment. In *Proceedings of the 9th Workshop on Scientific Cloud Computing*, pages 1–7, 2018.

[29] S. Hopewell, M. Clarke, and S. Mallett. Grey literature and systematic reviews. *Publication bias in meta-analysis: Prevention, assessment and adjustments*, pages 49–72, 2005.

[30] https://cloudnativenow.com. The Brutal Learning Curve of a New Kubernetes Cluster . `https://cloudnativenow.com/features/the-brutal-learning-curve-of-a-new-kubernetes-cluster/`, 2023. [Online; accessed 20-June-2023].

[31] https://microsoft.github.io/. Microsoft Threat Matrix. `https://microsoft.github.io/Threat-Matrix-for-Kubernetes/tactics/InitialAccess/`, 2023. [Online; accessed 28-April-2023].

[32] https://owasp.org/. OWASP Kubernetes Top Ten. `https://owasp.org/www-project-kubernetes-top-ten/`, 2023. [Online; accessed 28-April-2023].

[33] https://www.armosec.io. Definitive Guide to Kubernetes Admission Controller. `https://www.armosec.io/blog/kubernetes-admission-controller/`, 2023. [Online; accessed 28-April-2023].

[34] https://www.cncf.io. CNCF ANNUAL SURVEY 2021 . `https://www.cncf.io/wp-content/uploads/2022/02/CNCF-Annual-Survey-2021.pdf`, 2022. [Online; accessed 20-June-2023].

[35] https://www.forbes.com. Addressing The Kubernetes Skills Gap . `https://www.forbes.com/sites/forbestechcouncil/2023/05/10/addressing-the-kubernetes-skills-gap/?sh=6f5bc84e23f4`, 2023. [Online; accessed 20-June-2023].

[36] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino. Lteinspector: A systematic approach for adversarial testing of 4g lte. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[37] I. Karim, S. R. Hussain, and E. Bertino. Prochecker: An automated security and privacy analysis framework for 4g lte protocol implementations. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 773–785. IEEE, 2021.

[38] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[39] D. Kortepeter. U.S. lawmakers eye AWS role in Capital One data breach, 2019.

[40] kubelinter. kubelinter. `https://docs.kubelinter.io/#/generated/checks`, 2022. [Online; accessed 13-May-2022].

[41] Kubernetes. Production-grade container orchestration.

[42] Kubernetes User Case Studies, May 2020.

[43] Kubernetes. Kubernetes Conformance Tests . `https://github.com/kubernetes/kubernetes/blob/master/test/conformance/testdata/conformance.yaml`, 2024. [Online; accessed 26-March-2024].

[44] M. M. Lombardi and D. G. Oblinger. Authentic learning for the 21st century: An overview. *Educause learning initiative*, 1(2007):1–12, 2007.

[45] F. W. Maina. Authentic learning: Perspectives from contemporary educators. 2004.

[46] A. Martin and M. Hausenblas. *Hacking Kubernetes: Threat-Driven Analysis and Defense.* O'Reilly Media, 2021.

[47] V. Medel, O. Rana, J. a. Banares, and U. Arronategui. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, page 257–262, New York, NY, USA, 2016. Association for Computing Machinery.

[48] S. Miles. *Kubernetes: A Step-By-Step Guide For Beginners To Build, Manage, Develop, and Intelligently Deploy Applications By Using Kubernetes (2020 Edition).* Independently Published, 2020.

[49] Mirantis. What are the primary reasons your organization is using Kubernetes?, 2021.

[50] S. Muralidharan, G. Song, and H. Ko. Monitoring and managing iot applications in smart cities using kubernetes. *CLOUD COMPUTING*, 11, 2019.

[51] Nmap.org. netcat tool. `https://nmap.org/ncat/`, 2023. [Online; accessed 28-April-2023].

[52] NSA. Kubernetes Hardening Guidance. `https://media.defense.gov/2021/Aug/03/2002820425/-1/-1/1/CTR_KUBERNETESHARDENINGGUIDANCE.PDF`, 2021. [Online; accessed 10-Jan-2022].

[53] A. Ornellas, K. Falkner, and E. Edman Stålbrandt. Enhancing graduates' employability skills through authentic learning approaches. *Higher education, skills and work-based learning*, 9(1):107–120, 2019.

[54] K. Qian, D. Lo, R. Parizi, F. Wu, E. Agu, and B.-T. Chu. Authentic learning secure software development (ssd) in computing education. In *2018 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2018.

[55] A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE, 2019.

[56] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams. Security smells in ansible and chef scripts: A replication study. *ACM Trans. Softw. Eng. Methodol.*, 30(1), Jan. 2021.

[57] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita. Security misconfigurations in open source kubernetes manifests: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, dec 2022. Just Accepted.

[58] A. Rahman, S. I. Shamim, H. Shahriar, and F. Wu. Can we use authentic learning to educate students about secure infrastructure as code development? In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 2*, pages 631–631, 2022.

[59] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin. Synthesizing continuous deployment practices used in software development. In *Proceedings of the 2015 Agile Conference*, AGILE '15, page 1–10, USA, 2015. IEEE Computer Society.

[60] RedHat. Kubernetes adoption, security, and market trends report, 2021.

[61] RedHat. State of Kubernetes Security Report, 2021.

[62] RedHat. State of Kubernetes Security Report, 2021.

[63] redis.io. Redis Security. `https://redis.io/docs/management/security/`, 2023. [Online; accessed 28-April-2023].

[64] J. Saldaña. *The coding manual for qualitative researchers.* Sage, 2015.

[65] J. Saldana. *The Coding Manual for Qualitative Researchers.* SAGE, 2015.

[66] Seth Art, Principal Security Consultant. Bad Pods: Kubernetes Pod Privilege Escalation. `https://bishopfox.com/blog/kubernetes-pod-privilege-escalation`, 2023. [Online; accessed 28-April-2023].

[67] J. Shah and D. Dubaria. Building modern clouds: Using docker, kubernetes google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189, 2019.

[68] Snyk. snyk. `https://snyk.io/security-rules/kubernetes/`, 2022. [Online; accessed 15-May-2022].

[69] M. Song, C. Zhang, and E. Haihong. An auto scaling system for api gateway based on kubernetes. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, pages 109–112, 2018.

[70] Stackrox. Kubernetes and container security and adoption trends, 2021.

[71] stefanprodan. stefanprodan/podinfo. `https://github.com/stefanprodan/podinfo`, 2022. [Online; accessed 12-Jan-2022].

[72] N. Surantha and F. Ivan. Secure kubernetes networking design based on zero trust model: A case study of financial service enterprise in indonesia. In *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 348–361. Springer, 2019.

[73] T4. Container Platform Market Share, Market Size and Industry Growth Drivers, 2018 - 2023, 2020.

[74] K. Takahashi, K. Aida, T. Tanjo, and J. Sun. A portable load balancer for kubernetes cluster. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, page 222–231, New York, NY, USA, 2018. Association for Computing Machinery.

[75] T. Taylor. 5 Kubernetes security incidents and what we can learn from them, 2020.

[76] C.-W. Tien, T.-Y. Huang, C.-W. Tien, T.-C. Huang, and S.-Y. Kuo. Kubanomaly: anomaly detection for the docker orchestration platform with neural network approaches. *Engineering reports*, 1(5):e12080, 2019.

[77] tutorialspoint.com. Redis - Security. `https://www.tutorialspoint.com/redis/redis_security.htm`, 2023. [Online; accessed 28-April-2023].

[78] A. A. Ur Rahman and L. Williams. Software security in devops: Synthesizing practitioners' perceptions and practices. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, CSED '16, pages 70–76, New York, NY, USA, 2016. ACM.

[79] Z. Wei-guo, M. Xi-lin, and Z. Jin-zhong. Research on kubernetes' resource scheduling scheme. In *Proceedings of the 8th International Conference on Communication and Network Security*, ICCNS 2018, page 144–148, New York, NY, USA, 2018. Association for Computing Machinery.

[80] wikipedia. OWASP. `https://en.wikipedia.org/wiki/OWASP`, 2023. [Online; accessed 28-April-2023].