# Kubernetes Security: A Review of Threats, Best Practices, and Real World Hardening Techniques

Tisha Ghadiya[1], Harsh Trivedi[2]
[1]*Noble University, Junagadh, Gujarat*
[2]*None*

*Abstract*— **Kubernetes has become the de facto platform for container orchestration in modern cloud-native environments. However, its complex architecture introduces a broad attack surface that can be exploited through misconfigurations and weak security controls. This paper presents a comprehensive analysis of Kubernetes security vulnerabilities, including RBAC misconfigurations, ServiceAccount token exposure, insecure admission configurations, and runtime threats. The study combines theoretical analysis with practical experimentation conducted in a controlled Kubernetes environment. Multiple attack scenarios are simulated, and mitigation strategies are evaluated using mechanisms such as Role-Based Access Control (RBAC), Pod Security Admission (PSA), and runtime monitoring using Falco. A key contribution of this work is the integration of Kyverno for automated policy enforcement, ensuring secure configurations at deployment time. The proposed layered security model demonstrates that combining preventive, detective, and enforcement-based controls significantly enhances cluster security. The results highlight that a defense-in-depth approach is essential for securing Kubernetes environments against real-world threats.**

*Index Terms* — **Kubernetes Security, RBAC, Falco, Kyverno, Container Security, Cloud-Native Security, Policy Enforcement**

## I. INTRODUCTION

In recent years, containerization and microservice-based architectures have significantly transformed application development and deployment. Kubernetes has emerged as the leading orchestration platform due to its ability to automate deployment, scaling, and management of containerized workloads.

Despite its advantages, Kubernetes introduces multiple security challenges due to its distributed and configuration-driven architecture. A cluster consists of components such as the API server, etcd datastore, kubelet agents, and admission controllers. Misconfigurations in any of these components can lead to severe vulnerabilities.[3]

One of the primary concerns is improper RBAC configuration, which can allow excessive permissions and enable privilege escalation. Similarly, ServiceAccount tokens mounted inside pods introduce risks if compromised. Insecure workload configurations, such as running containers as root, further increase the attack surface.[5]

Traditional security approaches focusing only on preventive controls are insufficient. Runtime threats require continuous monitoring and detection mechanisms. This research proposes a layered security framework integrating RBAC, PSA, Falco, and Kyverno to address these challenges through a defense-in-depth strategy. [9]

## II. LITERATURE REVIEW

Existing research in Kubernetes security highlights vulnerabilities arising from misconfigurations, weak access control mechanisms, and lack of runtime visibility. Studies such as [2] and [3] emphasize that RBAC misconfigurations and insecure defaults are among the primary causes of security breaches in Kubernetes environments. These works demonstrate that failure to enforce the principle of least privilege leads to unauthorized access and privilege escalation.

Research in runtime security [7], focuses on detecting malicious behavior within containers. Tools such as Falco leverage kernel-level monitoring using eBPF to identify anomalies such as unauthorized file access and abnormal process execution. However, these approaches primarily provide detection rather than prevention.

Policy-based enforcement mechanisms have been proposed to address configuration inconsistencies.

These approaches ensure that only compliant configurations are deployed, reducing the risk of security violations.

Despite these contributions, most studies address isolated aspects of security. There is a lack of integrated frameworks that combine access control, runtime monitoring, and policy enforcement. This research addresses this gap by proposing a unified layered security model.

## III. METHODOLOGY

The methodology adopted in this research combines theoretical analysis with practical experimentation. Initially, key attack vectors such as RBAC misconfigurations, ServiceAccount token exposure, and insecure workloads are identified [2]

A controlled Kubernetes environment is configured to simulate real-world scenarios. Security mechanisms including RBAC, Pod Security Admission, Falco, and Kyverno are implemented in multiple layers [12].

RBAC is used to enforce access control, PSA ensures secure workload configurations, Falco provides runtime monitoring, and Kyverno [9] enforces policies at deployment time. Each mechanism is validated individually and then integrated into a unified framework.

The system is tested through simulated attack scenarios to evaluate its effectiveness in preventing, detecting, and mitigating security threats.

## IV. SYSTEM DESIGN

The system design of the proposed Kubernetes security framework is developed using a layered, control-plane-driven architectural approach that closely aligns with the internal operational model of Kubernetes. Unlike traditional application-centric designs, this architecture focuses on securing the Kubernetes request lifecycle by embedding security controls at multiple stages, including authentication, authorization, admission control, workload execution, and runtime monitoring.

At the core of the system lies the Kubernetes control plane, which consists of the API Server, etcd datastore, Scheduler, and Controller Manager. The API Server acts as the central entry point for all cluster interactions and is responsible for processing every request issued by users, service accounts, or automated systems. This makes it the most critical security boundary within the architecture.

The request lifecycle begins with authentication (AuthN), where the identity of the requester is verified using mechanisms such as X.509 certificates, bearer tokens, or external identity providers. Once authenticated, the request proceeds to the authorization (AuthZ) phase, where Role-Based Access Control (RBAC) policies are evaluated [10]. RBAC operates by matching the request attributes (user, verb, resource, namespace) against defined roles and role bindings. If a matching rule is found, the request is allowed; otherwise, it is denied. This ensures enforcement of the principle of least privilege at the API layer.

Following authorization, the request enters the admission control pipeline, which serves as a critical enforcement stage before the resource is persisted in the cluster state. Admission controllers are executed sequentially and can validate or mutate incoming requests. In this architecture, two primary admission mechanisms are integrated: Pod Security Admission (PSA) and Kyverno.

Pod Security Admission enforces predefined security standards at the namespace level, ensuring that workloads adhere to restricted security profiles [13]. It evaluates parameters such as privilege escalation, capability usage, and user permissions within containers. Kyverno extends this capability by providing dynamic, policy-driven validation and mutation of Kubernetes resources. Unlike PSA, which is rule-based, Kyverno allows custom policy definitions, enabling enforcement of organization-specific security requirements such as non-root execution and restricted volume mounts.

Once a request passes admission control, it is persisted in etcd, which serves as the single source of truth for the cluster state. The scheduler then assigns workloads (such as pods) to appropriate worker nodes based on resource availability and constraints. This scheduling decision marks the transition from control plane validation to data plane execution.

The data plane consists of worker nodes, each running kubelet, container runtime (e.g., containerd), and networking components. The kubelet is responsible for ensuring that the desired state defined in etcd is reflected in the actual state of the node. It communicates with the container runtime to pull images, create containers, and manage pod lifecycle.

Runtime security is integrated through Falco, which operates independently of the control plane. Falco is deployed as a DaemonSet, ensuring coverage across all nodes. It leverages eBPF or kernel modules to intercept system calls generated by containers [5]. These system calls are evaluated against predefined rules to detect anomalies such as unauthorized shell access, file modifications, or privilege escalation attempts.

From a network perspective, the system enforces clear separation between external access, internal service communication, and workload execution. External requests enter through controlled ingress points, while internal communication is governed by Kubernetes Services and network policies. This separation minimizes the attack surface and ensures controlled traffic flow within the cluster.

The overall architecture follows a defense-in-depth strategy, where each layer contributes to system security. RBAC restricts access, admission controllers prevent insecure configurations, runtime monitoring detects malicious behavior, and policy enforcement ensures continuous compliance. This multi-layered design ensures that even if one layer is bypassed, subsequent layers provide protection.

Additionally, the system incorporates feedback loops through Kubernetes controllers and monitoring mechanisms. Runtime alerts generated by Falco can be used to trigger incident response workflows, while controller loops ensure continuous reconciliation of system state. This dynamic and self-healing behavior aligns with Kubernetes' declarative model and enhances overall system resilience.

The proposed system design demonstrates that effective Kubernetes security requires integration across multiple layers rather than reliance on a single mechanism. By embedding security controls throughout the request lifecycle, the architecture provides comprehensive protection against both configuration-based and runtime threats.

## V. IMPLEMENTATION

The implementation of the proposed Kubernetes security framework is carried out in a structured, multi-phase manner, where each security mechanism is deployed, validated, and analyzed within a controlled environment. The implementation focuses not only on applying configurations but also on understanding the internal behavior of Kubernetes components when these security controls are enforced.

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: paper-sa
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-log-reader
  namespace: default
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-log-reader-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: paper-sa
  namespace: default
roleRef:
  kind: Role
  name: pod-log-reader
  apiGroup: rbac.authorization.k8s.io
```

Fig. 1 RBAC configuration

The first phase involves implementing Role-Based Access Control (RBAC) to regulate access to cluster resources. A ServiceAccount named "paper-sa" is created and associated with a restricted role that allows access only to specific resources such as pods. When a request is issued using this ServiceAccount, the API Server evaluates the request against RBAC policies stored in its authorization layer. Internally, this evaluation involves matching request attributes with role rules defined in the cluster. If the request matches an allowed rule, it is permitted; otherwise, it is denied. Experimental validation confirms that

access to pods is allowed, while access to sensitive resources such as secrets is blocked, demonstrating effective enforcement of least privilege.

The second phase focuses on analyzing ServiceAccount token behavior. By default, Kubernetes injects ServiceAccount tokens into pods through projected volumes managed by the kubelet. When a pod is created, the API Server generates a token associated with the ServiceAccount, and the kubelet mounts this token inside the container filesystem. This enables the pod to authenticate with the API Server. However, this behavior introduces a potential security risk, as compromised containers can use these tokens to interact with the API [11]. To mitigate this risk, automatic token mounting is disabled in the pod specification. As a result, the kubelet no longer injects the token into the container, effectively reducing the attack surface and preventing unauthorized API access.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sa-demo-no-token
  namespace: default
spec:
  automountServiceAccountToken: false
  serviceAccountName: paper-sa
  containers:
  - name: app
    image: busybox:1.36
    command: ["sh", "-c", "sleep 3600"]
```

Fig. 2 Pod configuration with ServiceAccount token automount disabled

The third phase implements Pod Security Admission (PSA) to enforce workload-level security. A namespace is configured with restricted security labels, ensuring that all pods deployed within it must comply with defined security standards [13]. When a pod creation request is submitted, the API Server forwards it to the PSA controller, which evaluates the security context of the pod. If the configuration violates policy constraints, such as running in privileged mode, the request is rejected before being stored in etcd. This demonstrates that PSA acts as a preventive control, blocking insecure workloads at the admission stage itself.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: psa-demo
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/enforce-version: latest
```

Fig. 3 Namespace configuration with (PSA) baseline enforcement enabled

The fourth phase introduces runtime monitoring using Falco. Falco is deployed as a DaemonSet across all nodes, ensuring continuous monitoring of container activities. Internally, Falco intercepts system calls generated by containers using eBPF. These system calls are analyzed against predefined rules to detect suspicious behavior. For example, when a container attempts to spawn an interactive shell, Falco identifies this as an anomaly and generates an alert. This capability provides real-time visibility into runtime activities, enabling detection of threats that bypass static configuration checks.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: privileged-test
  namespace: psa-demo
spec:
  containers:
  - name: c1
    image: nginx:1.27
    securityContext:
      privileged: true
```

Fig. 4 Example of a privileged pod configuration which will be rejected by PSA policies

The final phase integrates Kyverno as a dynamic policy enforcement engine. Kyverno operates as an admission controller and evaluates resource definitions against defined policies during the admission phase. Policies are created to enforce secure configurations, such as requiring containers to run as non-root users. When a non-compliant resource is submitted, Kyverno rejects the request before it is persisted in etcd. Internally, this process

involves webhook-based validation, where the API Server sends the resource definition to Kyverno for evaluation. This ensures that policy enforcement is automated and consistent across all deployments.

The integration of these mechanisms forms a complete security pipeline that spans the entire Kubernetes lifecycle. RBAC enforces access control at the authorization stage, PSA and Kyverno enforce policies during admission, and Falco provides runtime monitoring at the node level. Each component operates independently yet contributes to a unified security framework [6].

```
customRules:
  rules-custom.yaml: |-
    - rule: Shell In Container
      desc: Detect shell execution inside a container
      condition: evt.type=execve and (proc.name in (ba
      output: Shell spawned in container (user=%user.n
      priority: WARNING
      tags: [container, execution]
```

Fig. 5 Custom Falco rule for detecting shell execution inside containers at runtime

This layered implementation demonstrates that effective Kubernetes security requires coordination between control plane validation and runtime observation. By combining preventive and detective mechanisms, the system ensures comprehensive protection against a wide range of attack vectors.

## VI. RESULTS

The results obtained from the implementation of the proposed Kubernetes security framework provide a comprehensive understanding of how different security mechanisms operate across various stages of the Kubernetes lifecycle. Unlike traditional evaluation approaches that focus solely on success or failure of configurations, this study emphasizes behavioral analysis of the system under controlled attack scenarios.

The evaluation begins with Role-Based Access Control (RBAC), where controlled access requests are issued using a restricted ServiceAccount. Internally, each request is processed by the Kubernetes API Server, which performs authorization checks by evaluating the request attributes against RBAC policies stored in its authorization module. The results demonstrate that requests aligned with defined permissions are successfully executed, while unauthorized requests are denied with explicit error responses. This confirms that RBAC enforces strict access boundaries and effectively prevents privilege escalation.

A deeper observation reveals that RBAC operates deterministically, meaning that every request is evaluated against predefined rules without ambiguity. This ensures predictability in access control behavior. However, the results also highlight that RBAC alone is insufficient, as it does not prevent misconfigurations at the workload level or detect runtime threats. This reinforces the need for additional security layers.

ServiceAccount token analysis provides critical insights into credential exposure risks. By default, Kubernetes automatically injects tokens into pods through projected volumes managed by the kubelet. Experimental results show that these tokens are accessible within the container filesystem, making them vulnerable to exploitation if the container is compromised. When token mounting is disabled, the absence of token directories confirms that credentials are no longer exposed. This demonstrates that simple configuration changes can significantly reduce the attack surface.

Further analysis indicates that token exposure is not inherently a vulnerability but becomes a risk when combined with other attack vectors such as container compromise. This highlights the importance of contextual security, where multiple factors contribute to overall risk.

Pod Security Admission (PSA) results demonstrate strong enforcement of workload-level security policies. When insecure pod configurations are submitted, the API Server forwards the request to the admission controller, which evaluates it against predefined security standards. The results show that pods requesting privileged access or violating security constraints are consistently rejected before being persisted in etcd. This confirms that PSA acts as an effective preventive control mechanism.

An important observation is that PSA operates at the namespace level, enabling consistent enforcement across multiple workloads. This reduces the risk of configuration drift and ensures uniform security standards within the cluster.

Runtime monitoring results obtained from Falco provide valuable insights into system behavior during execution. Falco continuously monitors system calls generated by containers and evaluates them against predefined rules. During testing, various suspicious activities such as shell execution and file access are simulated. Falco successfully detects these activities and generates real-time alerts.

The results highlight that runtime monitoring complements preventive controls by providing visibility into actual system behavior. Unlike RBAC and PSA, which operate before execution, Falco operates during execution, enabling detection of threats that bypass initial security checks.

Kyverno enforcement results demonstrate the effectiveness of policy-based security controls. Kyverno operates as an admission controller and evaluates resource definitions against defined policies. Experimental results show that non-compliant workloads are automatically rejected, while compliant configurations are successfully deployed. This ensures consistent enforcement of security standards across the cluster.

A key insight from the results is that Kyverno bridges the gap between configuration validation and policy enforcement. Unlike PSA, which enforces predefined rules, Kyverno allows dynamic policy definitions, enabling organizations to implement customized security controls.

When analyzing the combined behavior of all mechanisms, it becomes evident that each layer addresses a specific category of vulnerabilities. RBAC controls access, PSA and Kyverno enforce configuration security, and Falco provides runtime visibility. The integration of these mechanisms creates a comprehensive security pipeline that spans the entire Kubernetes lifecycle.

Comparative evaluation shows that single-layer security approaches are insufficient in dynamic environments. For example, RBAC cannot detect runtime anomalies, and Falco cannot prevent misconfigurations. The layered approach ensures that weaknesses in one layer are compensated by other layers.

Another important observation is the reduction in attack surface achieved through layered security. By restricting access, preventing insecure configurations, and monitoring runtime behavior, the system significantly reduces the number of exploitable entry points.

The results also demonstrate improved system resilience. Even when certain security controls are bypassed, other layers continue to provide protection. This redundancy is a key characteristic of defense-in-depth strategies.

Overall, the results confirm that the proposed layered security framework provides comprehensive protection against a wide range of Kubernetes security threats. The combination of preventive, detective, and enforcement-based controls ensures that vulnerabilities are addressed at multiple stages, significantly improving the overall security posture of the cluster.

## VII. TESTING

Testing and verification of the proposed Kubernetes security framework are conducted through a series of controlled experiments designed to simulate real-world attack scenarios [3]. The objective of testing is not only to validate the functionality of individual security mechanisms but also to evaluate their effectiveness when integrated into a unified system.

The testing process begins with RBAC validation, where unauthorized access attempts are simulated using restricted Service Accounts. These requests are processed by the API Server, which evaluates them against RBAC policies. The results consistently show that unauthorized actions are denied, confirming that RBAC effectively enforces access control. Further analysis indicates that RBAC decisions are made in real time, ensuring immediate response to access requests.

Service Account token testing focuses on evaluating credential exposure risks. Pods are deployed with default configurations, and the presence of tokens within the container filesystem is verified. This confirms that tokens are accessible and can be potentially exploited. When token mounting is disabled, the absence of tokens confirms successful mitigation. This test demonstrates that configuration-level changes can significantly reduce security risks.

Pod Security Admission testing involves deploying workloads with insecure configurations, such as privileged containers. The API Server forwards these requests to the admission controller, which evaluates them against security policies. The results show that insecure workloads are consistently rejected,

preventing them from being executed within the cluster. This confirms the effectiveness of preventive controls in reducing attack surface.

Kyverno testing extends this validation by enforcing custom policies. Both compliant and non-compliant workloads are deployed to evaluate policy enforcement. Non-compliant workloads are rejected, while compliant ones are successfully deployed. This demonstrates that Kyverno provides flexible and automated policy enforcement, ensuring consistency across deployments.

Runtime testing is conducted using Falco, where suspicious activities are simulated within running containers. These activities include shell execution, file access, and privilege escalation attempts. Falco successfully detects these anomalies and generates alerts in real time. This confirms that runtime monitoring is essential for identifying threats that cannot be detected through static analysis [7].

In addition to individual testing, integrated testing is performed to evaluate the combined effectiveness of all security mechanisms. Attack scenarios are designed to bypass one layer of security and test whether other layers can detect or mitigate the threat. The results show that even when one mechanism is bypassed, other layers continue to provide protection, demonstrating the robustness of the layered security approach.

Failure scenario testing is also conducted to evaluate system behavior under adverse conditions. For example, attempts are made to deploy insecure workloads with modified configurations to bypass admission controls. While some attempts may bypass initial validation, runtime monitoring detects suspicious behavior, ensuring that threats are identified even after deployment.

Another important aspect of testing is performance evaluation. The impact of security mechanisms on system performance is analyzed to ensure that security does not compromise system efficiency. The results indicate that while there is a slight overhead due to monitoring and policy enforcement, it is within acceptable limits and does not significantly affect system performance [8].

The testing results also highlight the importance of continuous monitoring and policy updates. As new threats emerge, security policies and detection rules must be updated to ensure ongoing protection. This emphasizes the need for adaptive security mechanisms in dynamic environments.

Overall, the testing and verification process confirms that the proposed Kubernetes security framework effectively addresses both preventive and reactive aspects of security. The combination of RBAC, PSA, Kyverno, and Falco provides comprehensive coverage across different stages of the system lifecycle.

The results validate that the system meets the defined security objectives and provides a robust solution for securing Kubernetes environments against real-world threats.

## ACKNOWLEDGMENT

I would like to express my sincere gratitude to my teachers for their invaluable guidance and insights into cloud-native security throughout this project. Special thanks go to my HOD for their constructive feedback on the YAML security policies, and to my family for their unwavering support. Finally, I acknowledge the open-source Kubernetes community, whose extensive documentation and security benchmarks were foundational to this research.

## REFERENCES

[1] S. I. Shamim, R. G. Gibson, and M. W. Sensuse, "Benefits, Challenges, and Research Topics: A Multi-vocal Literature Review of Kubernetes," arXiv preprint arXiv:2211.07032arXiv preprint arXiv:2211.07032, 2022.

[2] M. B. Skowron et al.et al., "XI Commandments of Kubernetes Security: A Systematization of Knowledge," arXiv preprint arXiv:2006.15275arXiv preprint arXiv:2006.15275, 2020.

[3] A. Alqarni et al.et al., "Configuration Defects in Kubernetes," arXiv preprint arXiv:2512.05062arXiv preprint arXiv:2512.05062, 2025.

[4] "The Kubernetes Security Landscape: AI-Driven Insights from Real-World Discussions," arXiv preprint arXiv:2409.04647arXiv preprint arXiv:2409.04647, 2024.

[5] M. A. Syairozi and F. Arizal, "Container runtime security detection and prevention techniques," ZenodoZenodo, Apr. 2025. [Online]. Available: https://zenodo.org/records/15236086

[6] "Integrating Trivy, Falco, and OPA for Container Security in Kubernetes," J. Secur. Adv. Eng. Res.J. Secur. Adv. Eng. Res., vol. 6, no. 2, pp. 216-220, 2019.
[7] D. Kumar, "Evaluating Falco's ability to detect lateral movement in Kubernetes," Bachelor's thesis, Radboud Univ., Nijmegen, Netherlands, 2025.
[8] A. A. Syairozi and F. Arizal, "Comparative Analysis of eBPF-Based Runtime Security Monitoring Tools," in Proc. Int. Conf. Inf. Syst. Technol.Proc. Int. Conf. Inf. Syst. Technol., 2025, pp. 136-141.
[9] S. Yadav, "Optimizing Kubernetes Security through automated Policy Enforcement," Rep., Norma NCIRL, 2025.
[10] Kubernetes Documentation, "Security," Kubernetes.io. [Online]. Available: https://kubernetes.io/docs/concepts/security/
[11] NSA Cybersecurity Technical Report, "Kubernetes Hardening Guidance v1.2," Nat. Sec. Agency, Fort Meade, MD, USA, Aug. 2022. [Online]. Available: https://media.defense.gov /2022/Aug/29/200306362/-1/-1/0/CTR_ KUBERNETES_HARDENING_GUIDANCE_1 .2_20220829.PDF
[12] CNCF Security TAG, "Cloud Native Security Whitepaper v2," Cloud Native Computing Found., May 2022. [Online]. Available: https:// github.com/cncf/tag-security/blob/main /community /resources/security-whitepaper /v2/ CNCF_cloud-native-security-whitepaper- May2022-v2.pdf
[13] CIS Kubernetes Benchmark, "Center for Internet Security Kubernetes Benchmark v1.9.0," CISecurity.org, 2025. [Online]. Available: https: //www.cisecurity.org/benchmark/kubernetes