International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(18s) 2026 pp. 1083-1099
https://doi.org/10.70917/ijcisim-2026-4181
Article

Empirical Analysis and Mitigation Strategies for Kube-API server Vulnerabilities: Countering Security Posture Risks

Manikandan S.[1], Cecil Donald[2]

[1] Department of Computer Science, Christ University, Bengaluru, Karnataka, India.
Email: manikandan.s@res.christuniversity.in
[2] Department of Computer Science, Christ University, Bengaluru, Karnataka, India.
Email: cecil.donald@christuniversity.in

Abstract: For every Kubernetes cluster, Kube-API server acts as its heart, considered to be a single-entry point across cluster where all communication related to workload state & policies are enforced. When organization needs increases to address multi-tenant and production grade environments, it is must to focus on security risk involved in critical components like Kube-API. Considering API server importance, there are numerous opportunities to penetrate this component considering its architectural behaviour leading to security risks. This paper focuses on three such security posture risks, first one is related to API server deployments where mixed-versions during the component rolling upgrades causes the network traffic to route towards older cluster instances without proper validation checks/admission controls bypassing existing policies. The second concern is related to configuration set for API priority and fairness (APF), where shared priority queues across cluster can be compromised by a single noisy client causing other clients to hit denial-service issues. The third security risk is leakage of critical API server information via monitoring tool metrics; this information may be related to sensitive cluster workload details in a namespace/label to users with read-only access. For the mentioned vulnerabilities, this article performs complete analysis finding their underlying root cause, testing the issue and reproducing in internal environments and providing remediation strategies. The strategies are focused at workload configuration setup, analysing existing YAML structures, and suggesting code level fixes. The CVE analysis and findings highlight; API server requires focus and strategies to be implemented during upgrade flows, existing observability metric architecture and traffic related isolation.
Keywords: kube-API, Container Orchestration, CVE, Cluster lifecycle management, Security, Kubernetes

1. Introduction

With greater cluster related deployment tasks, comes greater security risks, especially in this modern software deployment and container orchestration process, Kubernetes plays a major role in cloud native architecture.

It is ignorant to not consider Kube-API server component security posture in a Kubernetes cluster, as this is not just a centre point in a cluster, but manages all the communication within cluster and takes important decisions related workload scheduling, config changes to be altered and policies to be enforced, making it cluster central nervous system. If this component is hampered, compromised or misleading it to incorrect information can greatly affect the cluster workload. This paper examines three vulnerabilities, these risks doesn't require an external attacker to compromise system or affecting coded credentials. Rather they expose, how the API server is designed to behave and how the component reacts under unexpected scenarios like performing rolling upgrades, involving multiple tenant traffic and their associated workload, and underlying observability infrastructure. The mentioned distinction is needed to be considered as these security risks won't be alarmed in vulnerability scanners, they require internal Kubernetes component understanding to discover such security risks.

Vulnerability 1: Kube-API Server Rolling Upgrade involving Mixed Cluster version.

For simpler cluster environments, control plan related upgrades are done instant with ease as the nodes involved are less. However for complex production setups, it is gradual process involving multiple nodes which needs to be uplifted one at a time, where the old and new uplifted version instances will be running on same load balanced traffic endpoint. For normal version upgrade scenarios these are manageable but when newer version introduces complex/strict validation logic which the older version doesn't hold is when the problem arises, as the load balancer wouldn't be aware of which version expects which related traffic to be routed and load balancer will route traffic equally across both, in such cases where the traffic request might flow in one version might be completely discarded in another version while the end user would wonder what caused the issue leading to security risk for clusters which require zero downtime requirements. This research focuses on changes that could be made at architecture level to address such upgrade scenarios.

Vulnerability 2: API Priority and Fairness related Violations.

The implementation reason for APF is to protect by ensuring traffic flows are allowed from all clients based on their respective priority. The goal is to analyse the requests sent in group, check their priority levels and provide the concurrent budget for each further letting the scheduler know which needs to be addressed first, the same feature of APF also leads to security risks. Lets assume when the traffic generated by user and APF share similar priority bucket due to configuration or product design, then the client with same priority generates heavy traffic will lead for that specific client to consume all the traffic budget crowding out the API related controller traffic.

The concerning factor is when a noisy client even without needed privileges can cause harm by just understanding the underlying traffic flow schema structure and its priority, which can make the client to send constant heavy traffic load according to this priority will lead to crash or would slow down the control plane node indirectly affecting the cluster workload for future authentic user requests. For the administrators, this would seem to be performance issue but the underlying cause is due to this APF priority level misjudgements.

Vulnerability 3: API Metric information leak via Observability systems.

Observability metrics such as from tools like Prometheus covering information related to Kube API server might need to be shared to administrators or across multiple teams as part of regular metric sharing to identify/address cluster related issues assuming the shared metrics are safe. The information shared usually covers any underlying system errors, latency counts, requests counts etc, but this assumption gets broken when labels pertaining that metric reveal critical information.

Information pertaining per-namespace or per-resource can speak a lot of information like whether a specific namespace is active or if any custom resource is being used, or if any specific operations are failing can be understood. If user with just read access to monitoring can access such information related to business can lead to security risk. With environments having strict restrictions to user regions such risks can be avoided , it is the metrics pathway holding critical information makes it a security risk.

Together, this paper discusses about the mentioned vulnerabilities with hide in plain sight making API server look secured but it is vulnerable in context. The idea of this paper is to not only catalogue/discuss about what is wrong but also analyse the root cause, reproduce and test the behaviour in internal environment with realistic production-like setup then provide tested remediations at workload configuration level, restructuring the existing YAML patterns and providing pseudo-code level fixes such that administrators/organizations can adapt post testing the suggestions in their lower environments. Analysing and understanding such vulnerabilities that exists as part of API-server are first step in securing and building production ready clusters that maintains its uptime and works efficiently.

2. Literature Review

The research concepts around Kubernetes world have evolved over the time, driven by newer inventions made and adopted by organizations using cloud native architecture. This paper discusses around eighteen papers about the

security posture of Kubernetes which is relevant by covering concepts related to enforcement pertaining cluster admission, integrity maintained at control plane node and metric exposure at observability layer together this paper provides a combined review.

[1] This paper provides a survey of various security challenges encountered at container orchestration level, where Kubernetes concepts play a crucial role in current market. The author details about how Kube-API server plays a crucial role as a gateway and maps out the security attack across various other components like ETCD, scheduler, node related agents further highlighting the risk of compromising API server will halt the overall cluster state.

[2] This paper primarily focuses upon Kubernetes related admission webhook control security. This work analyses about webhooks related to admission control can be configured, altered and compromised across various versions. The author by conducting internal experiments concluded that as long as there is version uniformity across admission webhook and API server the cluster tends to be reliable but vulnerability is explored when mixed versions are introduced, they further recommended stricter admission policies to ensure security posture of the cluster.

[3] This paper talks about various security risks involved while performing rolling upgrade in a distributed environment, they provided a theoretical framework although not related to Kubernetes, the authors talk about strict policy windows during partial rolling upgrades and further introduced a concept called “enforcement-drift”. This concept talks about time during which different validation logics are applied for different nodes for the same set of user requests – this closely resonates the vulnerability behaviour detailed in this paper.

[4] The authors in this paper talk about security risks during denial of service for cloud native gateways involving API and for rate-limit architectures, focusing on how priority queue mechanism can be violated. This paper highlights how an attacker with queue priority rule knowledge can send dummy requests to hamper the actual client requests sent to cluster leading to vulnerability risk. This provides an understanding upon how APF can be violated when configurations related to schema are not carefully administered allowing untrusted traffic flows.

[5] This study provides an in-depth understanding related to API priority and fairness both at security perspective and at performance level. The authors provided benchmark ratings for APF behaviour, by pushing high load concurrent traffic and isolate configurations due to which API traffic is violated by users having same priority levels. They provided data related to queue exhaustion points pertaining APF attack and provided recommendations to maintain ratio for traffic concurrency as remediation strategy.

[06] This paper inspects and analyses multi-tenant Kubernetes environment related security in-depth by exploring API server and admission webhooks as these two points in a cluster acts as gateway to cluster. The paper highlights the security needs to be enforced at the control plane layer rather than blindly relying on RBAC policies alone, also the security adapted for single tenant environment widely varies for multi-tenant setup.

[07] This paper focuses on identifying and cataloguing vulnerabilities across various Kubernetes releases following a CVE centric approach. The CVE’s focused involve cases related to API server request handling which co-relates with the context discussed in current research work. The author also highlighted how medium severity risks are majorly ignored during remediation phase.

[08] In this paper, This paper provided a theoretical analysis regarding observability tool related security risks. The author further analysed how there is a potential security risk involved due to metric leakage across shared environments. With internal testing environment, author highlights highly defined labels tend to leak critical information about the cluster state and related components to users with only read access in environment.

[09] In this article, critical relation between basic observability requirements and the need to provide minimal privilege across cloud native environment is discussed in detail. The author conducted deep survey highlighting upon API server metric are not properly sanitized during cluster deployments and the related unfiltered metrics are shared towards Prometheus observability tool. The author further provided recommendation to inculcate stricter tenant controls across API access to protect critical metric leakage to unintended users.

[10] This research article primarily focuses on configuration issues related to RBAC, further conducts deeper research related to existing cluster configuration and identifies how permission overlays provide easy access to API server leading to security risk. Alongside RBAC, API server related access controls as well are analysed providing information upon even on environments with strict RBAC can lead to API server vulnerabilities.

[11] This paper provides a methodology related to threat modelling pertaining control plane tools. The author uses STRIDE as base further extending upon the same, author introduced framework for threat modelling. The paper highlights that API server default hardening guidelines needs to be reconsidered while evaluating API server security. The framework corelates with API version mismatch and metric leakage highlighted in current work.

[12] This paper analyses security consequences that arises between Kubernetes and external load balancer, iterates about whenever there are changes made to traffic at infra level leading to unexpected consequences at application layer security. For the policies induced at infra level, makes the system let mixed version calls towards API server leading to security risk, showcasing the problem may not alone rely at Kubernetes level but also due to external sources regarding cluster initialization and deployment.

[13] This paper talks about inculcating finer strategies such as strict TLS, finely evaluated service identity mechanism and policy evaluation to be implemented until control plan node ensuring to have a zero trust mechanism. Existing security models were evaluated against the proposal and author identified issues at access provided for metric collection and admission control making this work align with the discussed vulnerabilities in this article.

[14] This paper discusses about existing compliance architecture such as CIS and NIST SP used by organizations by mapping their cluster configurations. The author identified major differences between the compliance framework recommendations and how the actual setup is made by organization especially around control plane configuration, related version management and access controls for observability tools. The discussed gaps provides a picture of security vulnerabilities existing in their production setup.

[15] In this paper a comprehensive analyses is performed across various attacker strategies through different entry points into a Kubernetes cluster leading to security risks. This paper identified that the most vulnerable components during attack scenarios are control plane and the API server being more critical as this acts as the entry point towards organizations workload cluster.

[16] This paper highlights how version mismatch introduced during upgrades in Kubernetes environments introduces security challenges and behavioural changes that becomes hard to track/audit by examining organization involved in supply chain. The authors further introduced strategies to overcome mixed version issues and identifying overlaps at admission control to address vulnerabilities as remediation action plan.

[17] In this paper, a detailed study is conducted regarding evolution of Kubernetes configuration over time in prolonged duration clusters, identify security risks when the cluster operations and complexities are increased over time. There is configuration drifts from initial cluster state specifically APF configurations are seen to take most changes with organizations introducing adjustments in traffic flow schemas leading to security risks. The paper iterates that from time to time APF configurations need to be reviewed and altered to protect the system state.

[18] The author in this paper evaluates environments by combining observability tooling used by cloud native organizations and data security/governance involved for the same. The author argues that these metric collection and sharing is not properly regulated during Kubernetes deployment leading to data leakage across teams. The paper discusses about real world issues where such metric leakage leads to severe security compromise scenarios and suggests to inculcate filtering for Kubernetes labels as a remediation approach to address these vulnerabilities.

3. Kube-API Server Vulnerabilities

For every communication that needs to be sent from an external source like cluster developers, administrators all need to pass via the gateway Kube-API server without an any secondary alternative making this a vulnerable target for security compromises. Any security risk that may arise in this component will affect the overall cluster state, the

three vulnerabilities discussed in this section covers three different angles upon which a security attack can be induced at the gateway.

A. Vulnerability 1: Mixed Version API servers and Weak Admission Checks

In critical production systems, the upgrade pattern is performed in a rolling manner, this way critical workload continues to run without disruption while the nodes gets upgraded one at a time maintaining the application workload. This issue arises when the latest upgraded API version introduces strict validation/admission rules that the previous version doesn't hold. As the load balancer is expected to route the traffic across both cluster instances, a traffic which could be rejected in newer admission logic will get cleared in older instance, making the cluster lead to unstable state.

Root Cause

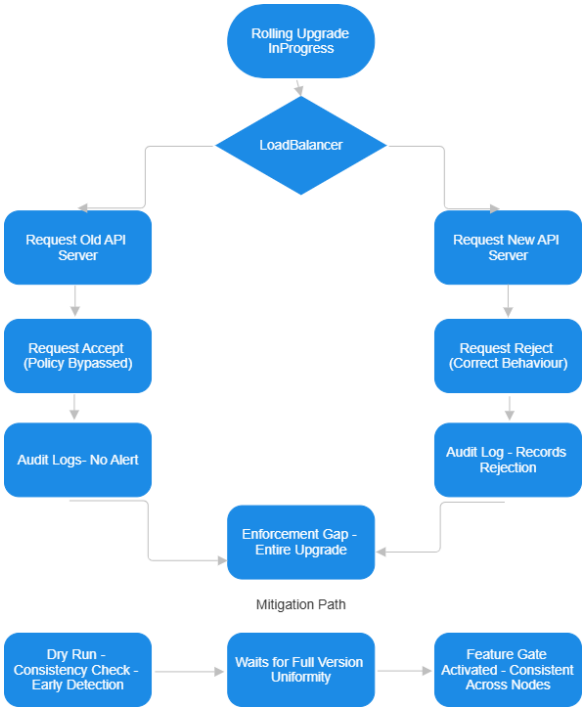
To understand the cause, we need to analyse the three design patterns relying on API server- rolling upgrades, admission rules and high cluster availability. Load balancers are expected to route traffic based on health signals received at the receiver node end without analysing version checks, as its functionality is to just route traffic. In the same way, each of these three design patterns have their own task to perform but when combined together during the mentioned rolling upgrade scenario is when the issue arises and goes unnoticed for the administrators. This issue becomes more complex when the administrators/deployment teams adds more admission rules and roll out the control plan upgrade causing security risk.

It is worth making a note that, these designs are working as expected as per their design structure and no configuration/specification are violated, the system as well works under expected conditions that it anticipates. The security risk arises when their operations are combines revolving around upgrade process and the fix cannot directly come just alone from code patch, the fix also needs to be planned by altering the procedures followed during upgrade, structing the same alongside making necessary changes at upstream code level.

Workaround Strategies

The strategies need to implemented for the entire flow sequence involved during upgrade – admission rules, validation at CRD schema level, and webhook configurations, these are expected to be activated only after confirming the all the nodes running API server in cluster instance are on the new version and shouldn't be introduced concurrently during upgrade process. The concept of Feature gates provide a clean mechanism to decouple which makes the new admission logics remain inactive within the binary and activate only after upgrade. Prior to upgrade, running a dry-run will provide a picture of new enforcement rules that will apply post upgrade, and setting clear affinity rules at load balancer will correct the routing behaviour per traffic route during mixed version upgrade window protecting from cluster exploitation.

Apart from these changes introduced at each level, from broader perspective administrators should treat control plan related upgrades as security events so there are also controlled entry and exit rules set rather than just a operational tasks in checklist. An upgrade shouldn't be considered complete without its consistency checked – whether there is automated check via scripts or manual run through done during dry-run, administrators should confirm that every API server control plane instance is enforcing same set of expected rules. This checklist should be incorporated in upgrade procedure runbook, these are the changes that organizations should adopt and inculcate to protect their application workload and make it reliable.



.

Figure 1. Rolling Upgrade leading to Mixed API Version

B. Vulnerability 2: Starving Control Plane Abusing API Priority and Fairness

The primary intention of API Priority-Fairness is to help administrators pass through critical traffic towards cluster even though there are interruptions via unintended client traffic. The process flow is such that each traffic is assigned with a priority, needed concurrency budget and queue request is passed when an allotted budget runs out. The issue is seen when client initiated unintended traffic and system level controller traffic end up having the same priority level. If the client ends up pulling all the API requests leading to consuming most of the budget will leave the controller with no requests getting passed through – the important requests end up waiting in queue or later getting dropped.

This concern looks low bar externally but it leaves the system in dangerous state. The external attacker wouldn't need any special privileges, or execute any code virus or need any special tools to crack this flaw. Once they have an authorized session and an understanding regarding the priority levels/configuration setup to evaluate their own request towards the APF will lead the mission critical system be in uneven state. The cluster symptoms would be – pods getting stuck in reconcile failing state, mid-way resource allotment stuck, cluster health checks failing but externally this would depict the cluster running in heavy and legitimate load.

Root Cause

The cause relies more on software defect rather than configuration failure. The underlying APF architecture provide ways to split actual system and user traffic by providing required priority levels, allotting reserved concurrency budget and strict traffic flow order but the issue arises when the enforcement isn't applied automatically as the cluster grows evolving its configuration setup. The breakage in traffic occurs when over a point of time these traffic flow schema are added one step at a time to manage the existing workload during which it creates a break path for bleeding unintended user traffic into APF setup. This is unintended issue where misconfiguration occurs over time.

The issue detection becomes difficult over time, the metrics generated by APF indicates increase in queue, related latency and concurrency overload in a busy cluster. But without appropriate APF alerts that specifically needs

to keep a watch on queue setup/growth in mission critical system while cluster workload increase over time, administrators will face difficult task in ruling out the underlying cause and assume issue to be at infrastructure end. As the natural resolution administrator would think it to scale existing resource without realising the actual symptom. The critical client request would end up being in queue for extended duration leaving only a small unintended client requests to exploit concurrency budget and saturate the queue.

Workaround Strategies

The primary fix is to isolate different system and user generated traffic at the APF level. This would help API controllers and mission critical control plane components to have dedicated APF levels with needed concurrency budget so external user generated traffic flow wouldn't match – this difference should be implemented at configuration level. The traffic flow setup needs to be put under scrutiny whenever critical system is carrying lower precedence number than a user generated schema so request is first-hand evaluated. If a single user traffic schema having an lower precedence count than the system one can lead to traffic misroutes even when the configuration setup is correct.

Each user flow checks needs to be setup at user end APF adding a second layer to scrutiny. This way user end traffic noise can only be able to exploit its over concurrency budget and not the entire system budget, making this a necessity security parameter check to avoid this vulnerability. Apart from configuration, metric collection at APF level is needed to capture queue depth over time in critical systems so it provide early signs of security risks.

In long run, APF related configuration changes need extreme evaluation alike RBAC policy changes. Each change to user traffic/APF level need to add additional check to differentiate intended and unintended user traffic. The administrators can track and maintain records of existing APF levels set for each namespace or user service account, reviewing them from time to time will help to prevent abrupt configuration drift over time leaving the cluster in vulnerable state. Most mission critical clusters which end up being security risk/vulnerable state are the one's which were hardened once without revisiting their configuration state from time to time.

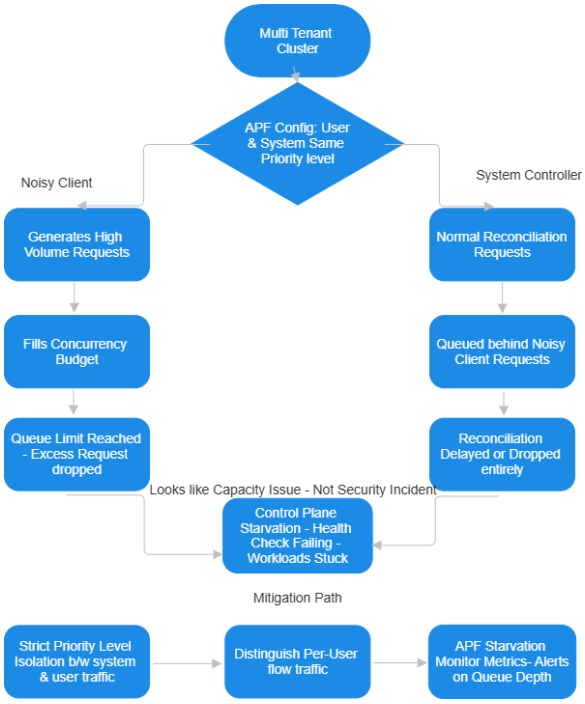


Figure 2. APF Starvation Flow Diagram

C. Vulnerability 3: API Server Metrics Information Leakage

The metrics exposed by API servers are detailed enough giving hints about API source group, resource type, namespace and related response from these sources. For administrators, these metric information serves good purpose to fuel their metric dashboard and resolve concerns quickly. From security/vulnerability standpoint it is a concern when such information gets shown to unintended teams within organization which can get surfaced via Grafana instances accessible to engineers who just may have authorization to check cluster health. This log not only is going to expose health checks but alongside cluster behaviour patterns making it vulnerable.

User with read-only access to a shared metric instance without authorization at API level can easily check cluster information like namespaces being critical, which resources are allotted, user triggered mission critical operations, if any unusual traffic has occurred within cluster. In organizational setup when each namespace is intended for specific team usage alone, the mentioned data leakage is critical vulnerability/security risk providing a pathway via metric monitoring stack.

Root Cause

The underlying cause relies in design without keeping up to the pace of evolution that occurred in Kubernetes metrics. During initial Kubernetes adoption, metrics carried low-sensitive operation related data and was reasonable. But as the architecture evolved and related content of Kubernetes labels grew, the metrics these components expose are critical. The gap relied was between the security model who governs the access metrics where separated from controls which governs direct API access.

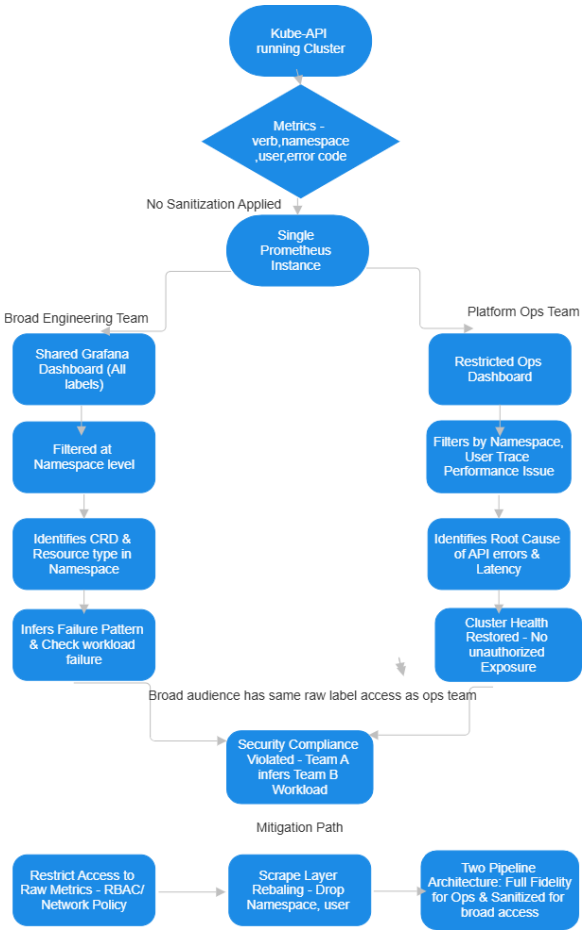
The Kubernetes label cardinality serves a reason to expose useful metric but as well as critical metrics leading to leakage. These labels at namespace level is what helps the metric dashboard to filter information and isolate performance related concerns or if any concerns exists at workload level. The same metric design also serves unauthorized user with no direct API access to cluster in viewing critical namespace information. Most organizations tend to run single metric instance to scrape cluster wide API server metrics with all related labels and namespaces. The same metric with no proper filtration/sanitization applied is further viewed by unintended internal users.

Workaround Strategies

The immediate action would be to restrict access at the metric endpoint level itself. The API server metric endpoint should have same level of security treatment as the API server itself – providing access only to service account related to metric collection and monitoring tools, enforcing strict network policies and RBAC controls combined so only intended information is shared and authorized user gets the expected metrics information. But to prevent the shared layer access at Prometheus end, need to inculcate remote-write boundaries which will filter/sanitize what is expected to reach the shared layer – information like namespace, related username, labels tied at sub resource level.

This two pipeline architecture is needed organization investment involving multiple tenants. A fully restricted pipeline will serve the intended security and platform related teams with needed labels intact. The filtered/sanitized pipeline will clear information carrying critical tenancy information also ensuring it serves intended audience with needed cluster health/performance related metrics. These pipelines can co-exists within the metric tooling ecosystem without needs to alter the API server, organizations need to have action plan for environment involving CRD and custom resource labels as these are organization specific variables that can be directly identified without a namespace tied with it.

Apart from this, audits need to be conducted for metric as part of regular security checks from time to time rather leaving them idle post initial configuration setup. Whenever the cluster grows in-terms of namespace, added resource types or additional workload the metric information shared changes alongside the Kubernetes label combination carrying sensitive information. The administrators should consider this shift as vulnerability in observability layer and check the security posture from time to time reviewing the user admission configurations and related RBAC policies to protect critical cluster information leakage.



.

Figure 3. Metric Leakage Flow Diagram

4. Security & Testing Parameters

The three concerns addressed in this paper arise during operational workload – such as upgrades, traffic routing, and observability conditions rather than core code level issue. So the test framework used for evaluation need to be designed accurately to evaluate API server under workload cluster rather than following traditional testing approaches.

A. Vulnerability 1: Mixed Version API Server Security & Testing Parameters

CVSS Scoring Assessment

This vulnerability is rated across the network attack layer – parameters to test just need an authenticated user client for Kubernetes API endpoint setup, no need of physical network access or any alter positioning needed at network. The exploitation complexity metric is set to High, as this would need attacker to understand when there is cluster upgrade/mid-upgrade, clear understanding on admission policy changes between version, and finally shape the trigger request to hit on old instance for the exploitation. From Privileges standpoint, it is set Low – attacker would just need a valid credential and for User Interaction, it is set None as the attack on whole is self-contained. For Scope, it is rated Changed as once the security attack is completed successfully bypassing admission rules set, it can affect multiple tenants/teams, related Kubernetes components and overall cluster security posture is hampered.

On the impact standpoint, Confidentiality is set to Low as the end result of attack point is on bypassing policy rather than targeting data exposure, though down the line effects such as vulnerable workload deployment can provide

a gap for attacker. Integrity score is High, it is primary harm here as the admission objects which shouldn't have existed in cluster are now present with unauthorized privileges, the expected design functionality of admission layer is hampered. Availability is scored Low, even though workloads deployed can be hampered at later point, from standalone availability perspective, score is set Low. The compiled score of all above parameters provides a CVSS score of 6.3 making the severity marked as Medium.

Kubernetes-Specific Testing Parameters

The single most security vulnerable standpoint risk is version skew window. The upgrade window is concern here for attack penetration, as an environment to upgrade all control plan nodes within fifteen minutes can differ for an environment that needs several hours due to manual upgrade procedures or intervention steps – the upgrade duration should always be tracked as for security assessment. The second parameter of concern is admission webhook policies – for cluster environment needing external webhooks like OPA gatekeeper or similar tools are more vulnerable than those organizations using built-in plugins, as the external webhook tools are not gated by versions in same way as built-in plugins. The testing framework used should evaluate all admission mechanism to flag accurate security vulnerable version-dependent behaviour.

Testing Methodology

The environment setup needs a minimum of two API server instances running on different minor versions behind any load balancer endpoint, also with CRD level validation or a fresh admission rule persisting within the new instance alone. To replicate the concern, the test should start by checking a request that new instance will reject and the older one accepts, then passing around a minimum of fifty test request iterations so accurate results are derived given variations existing at the load balancer level. Later the acceptance should be recorded against which the instance responded. The audit tracking logs should be always active on both old and new server instance alongside server headers or log identifiers active, so that we could correlate the traffic route and expected outcome accurately. An accurate setup exposing mixed version server instances should always return non-zero value for the test sample request sent via load balancer; a security hardened cluster where admission policy is active and deferred until post-upgrade was complete should return zero without concern as for which server handles the request.

The secondary testing mechanism should validate all cluster workload independently – setup done for feature gates, pre-validation checks for dry run, and session affinity set at load balancer level and evaluating them later in combined manner. First test run should be done at isolation for above setup to validate for baseline security risk later a combined setup will reveal gaps for configuration as a whole setup.

B. Vulnerability 2: APF Starvation at Control-Plane level traffic – Security & Testing Parameters

This vulnerability is rated at Network on the Attack vector parameter – there isn't need for any special network configuration modification, the entire attack can be induced at standard Kubernetes endpoint. The complexity set for Attack is Low, an authenticated user with access to understand traffic flow and assigned Priority level for cluster objects – a regular permission held by common cluster users can interact with existing APF configuration and induce a planned exhaustion on queue without need for any special tools. The Privilege required is set to Low, User mode Interaction set to None, and Scope is Unchanged as the external attacker/internal authorized user wouldn't change their existing privileges rather rely on what other cluster user depends upon.

The cluster impact dimension will provide clear picture on attack scenarios. Confidentiality score is None – there isn't data exposure at initial attack scenario. Integrity is Low, as once the queue starvation occurs at controller level resources within cluster leaving it in inconsistent state for longer duration affecting the cluster reliability without direct data impact. Availability remains primary concern, scored as High – once the queue is starved for longer duration, it will halt the controller getting reconciled, new workload processing gets queued/stuck, post which cluster health checks get into failure state. The mentioned factors combined provide a combined CVSS Base score of 6.5 setting it at Medium vulnerability band but considering the facts mentioned it has severe impact on production clusters.

Kubernetes-Specific Testing Parameters

The main configuration parameter here would be APF settings – before testing the starvation impact, the testing framework needs to be induced with complete flow of traffic schema and their existing Priority configurations and their associated Priority Levels for each cluster objects that tends to receive traffic signals from service accounts as well as user namespace workloads. The primary pre-requisite for replicating attack scenario is co-location existence within testing framework. Later concurrency budgets allotted for objects determine the starvation threshold limit – for the framework to judge and calculate how many users signals/requests it can process prior queuing at each shared priority level. This helps to understand the minimum resource requirement for reproducing successful attack also to adjust the parameters related to traffic load generation flow.

The third parameter would be request type selection as this carries more weightage. For long-lived requests – it requires higher resources with multiple operations triggered without any predetermined field selectors will hold more concurrency budget for longer span remaining effective starvation instruments. The testing framework should evaluate both aspects – short lived requests running at high count and long lived requests running at low count to compare which effectively saturates the target cluster, mostly the results are same for both methods.

Testing Methodology

The configuration setup for test environment should induce deliberate APF misconfiguration – either replication existing issue cluster scenario or to represent worst case condition in cluster workload. The baseline would be to have a deployment controller with redefined replica set count with known reconciliation cycle time – so difference can be identified for workload running with default configuration and when attack is induced targeting the APF shared priority level queue to understand and triage the issue further. Starvation is ideally concluded when the reconciliation time exceeds the expected baseline for a minimum of five repeated times to differentiate actual queue starvation from just normal cluster reconciliation noise.

The queue traffic load levels should be tested starting with below threshold value up until the saturation is reached so we get a clear curve picture to understand the issue and resources utilized to reach saturation. This curve provides understanding on two aspects – security risk the cluster can reach during unexpected loads and understanding regarding the concurrency budget to be allotted for cluster objects based on their tested usage scenarios beyond this defined budget would confirm a realistic attack on cluster workload. As part of mitigation check – isolation implemented at each priority level, per traffic flow concurrency distinguished, and need to adjust any gaps at concurrency budget – all these testing criteria should be tested individually before being evaluated in combined manner to identify the success pattern. The success criteria pattern is straight forward in a hardened cluster – the reconciliation time for API controller stays within expected baseline even during the span when attack is induced in a cluster running maximum load, which confirms that the individual user traffic is not interfering in the path of system traffic and its allotted concurrency budget under any unforeseen traffic conditions.

C. Vulnerability 3: API Metrics Information Leakage – Secuirty & Testing Parameters

From the attack vector standpoint, this vulnerability scored at Network layer – there isn't need for any user local cluster here, attack can occur at any metric tool endpoint such as Prometheus, Thanor or Grafana endpoint layer. The complexity for attack is set at Low, as attacker wouldn't need special skills or tools to interpret the Prometheus metrics – the user/attacker with knowledge of PromQL without any need for code exploit can query the cluster labels and build meaningful understanding about the existing cluster status. As part of observability onboarding for users, the privileges are added by default to metric tools so Privilege complexity is set to Low. The user interaction complexity is set to Low – as the information is easily/readily available without any interference needed from cluster administrator to trigger the information leakage, attacker/unintended user would just query the metric tool to fetch critical information.

Scope score is set Changed – The attacker/unintended user is crossing the default set tenancy boundary to understand cluster behaviour and related namespaces and resources for which the user doesn't hold any API

authorization crossing the set tenancy boundary. Confidentiality scores Medium – the data exposed isn't raw rather structured data through which attacker needs to understand behavioural patterns, workload data, error rates. But within environments holding strict authentication mechanism related to user tenancy or obligation pertaining confidentiality this score is set High. Scores for Integrity and Availability is set to Low, as the leaked information is read-only. All these combined provides a CVSS base score of 5.4 setting this vulnerability at Medium band – although Medium band, if the mentioned factors affect critical production setup, it can cause weightier impact on cluster workload.

Kubernetes-Specific Testing Parameters

The starting point for this vulnerability is at metric label inventory level. Testing needs to start at label dimension scope present on Kube-Api server as this covers label information of related namespace, resource type used, API groups it is tied with and error code related information – all these sit on a shared layer. All these metrics needs to be individually evaluated to understand attacker inference potential. CRD's set at metric flow needs to scanned, as these are usually organization specific holding customer resource group names that directly helps to understand critical workload without need of namespace context understanding, so this metric needs to be scanned before reaching shared layer.

An critical inspection target is configuration set for metric scraping and relabelling pipeline set between the raw metric data endpoint and shared backend endpoint. The entire path set for metric need to traced as well – from the API server endpoint related rules set for relabelling, federation layers, and configurations set for read-write needs to scanned to understand which metrics lands in shared and which gets filtered/aggregated during the transit. By examining above, organization will get to reveal unintended labels that are reaching shared layer.

Testing Methodology

The test framework requires need two set of access profiles – one with read access to shared layer of Prometheus or Grafana for metric account without API server permission just basic user authentication and another test reference account with full API server access and related namespace holding critical workload information which generates cluster behaviour patterns. The reference account can trigger controlled rate of failed requests towards any resource type that could create a pattern in metric flow series, then the metric account with read access alone can attempt to detect this pattern using standalone metric filter queries without needing any direct API access.

The test success rate can be measured via three inference tasks – to confirm critical workload namespace exists and active, further identifying most used resource type within it, and understanding an estimated error/failure rate that occurs within the same namespace. By successfully acknowledging two out of three tasks will confirm the vulnerability – the real world attacker/unintended users crossing the intended threshold set at organizational level will confirm the security attack. As for the remediation strategy, the same three tasks are individually ran in a controlled sequence – namespace existence pattern tasks will fail once the critical label stripping is performed, overall sanitization is validated once when metric revealing exact values are now replaced with bucketed values and access-control mechanisms are validated when metric read only account cannot reach endpoint revealing critical workload information with RBAC/network policies implemented. A Kubernetes metric system is confirmed to be fully remediated once its hardened configuration provides no space for metric reader account to infer in all three mentioned tasks – so critical cluster behaviour patterns are preserved protecting mission critical workload information leakage.

5. Results & Discussions

This sections details vulnerability in its unmitigated state and its related behaviour, post which shows the results achieved post implementing proposed workaround strategies and code fixes.

A. Vulnerability 1: Mixed Version API Servers – Results & Discussions

Existing State

When the configuration is unmitigated – two of API servers running on two different versions routed by a shared load balancer endpoint – the request content passed or privilege set at user level has nothing to do with enforcement outcome. It entirely depends on node picked by load balancer to route traffic. During the test, out of fifty iterations conducted approximately forty-nine percent of requests were accepted even with updated admission policies that were set to reject, the route tracking was almost sitting with fifty-fifty distribution. Which means, during upgrade window which can vary depending on load and cluster resource count from several minutes to hours – there was always a bypass available upon every request that can land in older instance version. For attacker knowing the upgrade duration/cadence alongside admission rules, this upgrade span is reliable path.

Another finding that concerns – the existing gap wasn't traced in audit log explicitly until both instances logs were combined and correlated to understand the behaviour. If single node logs are alone reviewed, the output would just show expected old version rejection while the other node was quietly accepting the same set of requests. This bypass with no audit signals can remain undetected and sit during entire upgrade window, making this vulnerability sit at Medium band with high cluster operational risk.

Post-Mitigation Results

After applying the suggested version guard pattern the unintended traffic acceptance rate came down to zero across all iterations – the older instance as expected stopped accepting requests and returned version mismatch error as intended in audit and as well as client logs. The suggested admission policy activation controller provided an add-on improvement: by isolating policy activation during upgrade initialization and automating the checks for version uniformity across all control plane nodes, so the upgrade duration window through which the gap existed for attacker to induced mixed-version requests is reduced to zero now. Under testing environment with two-hour upgrade window, there was reduction of ninety-six percent in policy exposure time frame compared to traditional approach of triggering admission rules at the beginning of upgrade frame rather at the end. An additional verification layer of running dry-run to check consistency helped to identify gaps in enforcement divergence within the monitoring span of test simulated upgrade operations. This helps administrators in providing actionable alerts before encountering version inconsistency behaviour in critical application workload.

Discussion

The primary takeaway from these tested results is even if an admission policy is rightly configured but only partially enforced across clusters is equivalent for not having policy configured at first place as the attacker is eventually going to induce version mismatch requests finding the loophole. The suggested activation controller and version guard helps to close gap identified at operational enforcement by implementing consistency at code level itself so manual human judgements are now removed and the cluster will now only implement additional policies that will get enforced across every cluster instance. For platform administrators, must have implication is to consider control plane upgrades as security events with predefined entry and expected exit criteria and not just another operational procedure to be carried out.

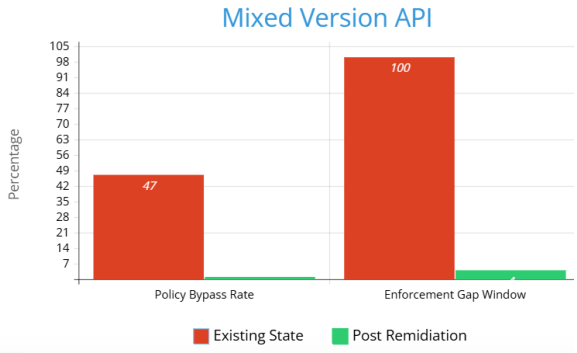


Figure 4. Mixed Version API Results

B. Vulnerability 2: APF Priority Starvation – Results & Discussions

Existing State

For a default APF configuration setup – where the user namespace and Kube-controller shared the same priority level during traffic flow – the degradation for API controller occurred within ninety seconds of starvation with clients starting requests. Under normal cluster operation scenarios, the test controller reconciliation cycle took around four-five seconds but during noisy client requests scenarios targeting the shared priority level, this number went up to thirty-two seconds under first four minutes – so this degradation was around seventy-five percent of its set default baseline. During maximum saturation the API controller went non-functional leading to new pod scheduling requests timing out as the traffic queue hit its defined configured limit, desired state was diverged from its expected state leaving the fresh requests being dropped out.

The attack traffic could easily bypass the default APF settings, the target starvation client usually maintained fifteen concurrent requests towards a resource group – this pattern cannot be distinguished by API server that usually gets similar legitimate requests from a busy application. The symptoms observed at operational layer – increased queue depth, concurrency budget exhausted & an increased latency across cluster – these were similarities observed on a high load application cluster. It is hard to conclude on root cause with just aggregated metrics fetched until and unless the APF level metric data specifically monitoring each priority level is checked and evaluated for root cause identification. An administrator just monitoring aggregated metrics would conclude the issue as resource shortage and scale the same rather focusing on actual root cause related to APF client requests/traffic flow schema.

Post-Mitigation Results

After hardening the APF configuration – ensured that no unintended user traffic reached Kube-API controller manager on its dedicated priority level and pre-reserved concurrency budget under any traffic flow conditions set – the testing was done using the starvation client for a minimum of three times of its original request rate, noticed there weren't noticeable impact during API controller reconciliation. The observed cycle time was around three-point-nine and four-point-three throughout the conducted test load; there was bare minimum variation of five percent from the pre-set baseline. A secondary result was derived upon testing per user traffic flow distinguisher bounded with priority-level: with default rule, a single noisy client could consume the entire allotted shared concurrency budget, but with distinguisher implemented each user were allowed to flow only within their assigned queue, ultimately reducing the blast radius by eighty two percent. The monitoring set for APF starvation detected the simulated attempt made to trigger client starvation within thirty seconds when queue depth crossed its alert threshold later no alert triggers were observed throughout the load test post-remediation being implemented.

Discussion

The derived results exposed the gap within denial-of-service and how the same modeled as part of Kubernetes security checks. Most constructed threat modeling focuses on workload resource exhaustion such as disk, memory, or more on the network level related traffic. But APF starvation falls under unmodeled category: to trigger an exhaustion at API controller doesn't need the attacker to have special user privileges, no unusual traffic patterns or any deviation apart from the normal API related usage patterns. As administrators rarely audit the APF misconfigurations which accumulates and changes quietly as cluster grows, this traffic flow schema misconfiguration goes unnoticed leaving the APF in a vulnerable state to be exploited. Post mitigation test results proves that remediation applied helps to strictly isolate user and system traffic at the APF level reducing the starvation attack towards API server to zero. From a broader aspect, configuration related to APF should be provided highest security governance similar to RBAC and admission policies.

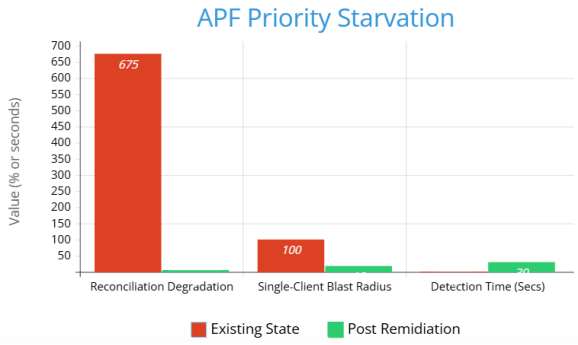


Figure 5. APF Priority Starvation Results

C. Vulnerability 3: API Metrics Information Leakage – Results & Discussion

Existing State

For existing configuration setup – detailed flow of metrics Kube-API server including namespace, response codes and errors are directly routed to shared Prometheus instance making to accessible to unintended users. The test account for metrics account showed a score of ninety nine percent completing all inference tasks in combination during repeated test runs. By simply filtering API server request, namespace existence could be determined by within sixty seconds into test run using namespace label which reveals values leading to a non-zero series. To identify resource type inference, it took around four minutes and for the pattern related to failures like error codes tied to a particular namespace took around eight minutes, none of these needing any specific query or tools. With basic filtering in Grafana dashboard we could derive API metrics, which clears that an attacker/unintended user with just dashboard access could extract critical workload metrics without even reaching to Kube-API server.

The critical Custom resource group (CRD) related metrics which are usually specific at organizational level had their critical metrics revealed via resource label part under apiserver_request_total – with the label value the attacker can easily understand the workload category. This vulnerability concern is too risky for critical organizations which are having regulated environments consisting of sensitive information under each namespace which are now revealed under the monitoring stack.

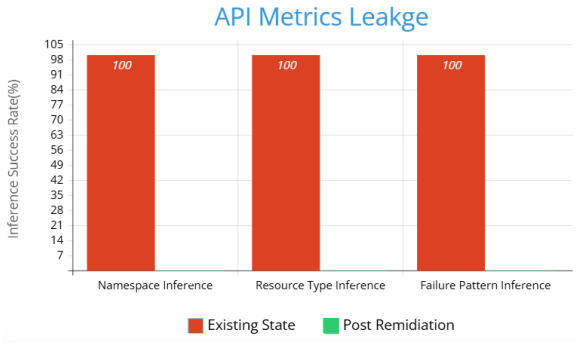
Post-Mitigation Results

Once the relabeling rules where applied at Prometheus by filtering namespace, error code, and sub resource labels, replacing CRD with a generic bucket label name before the actual metrics reached the shared layer – the test metrics reader account this time could not inference any of the three tasks. Namespace metric leakage failed as now the label dimension was removed at the shared layer, resource type leakage is now nil as now the CRD resource label are provided as generic bucket labels and the pattern to recognize failures provided no signals as now the label information needed to tie error codes with critical workload no longer exists at the shared layer, so the mitigation achieved one hundred percent of inference pathways being closed. However, the needed aggregated metric information consisting of request rates, error codes, latency related information and needed system information remained intact at the shared layer, protecting the needed health monitoring capabilities but tenancy related details were eliminated without any exception.

Now the fully detailed metric pipeline retaining all the label dimensions are made available only for intended administrators/platform security teams so their operations of detailed investigation/debugging capabilities are retained. The two-pipeline architecture required zero changes at the API server level and caused no operation overhead at the Prometheus level. The audit logs now identified the metric leakage for all critical label dimensions at the pre-remediation state itself so a clean slate was provided before relabeling rules were applied.

Discussion

The critical metric leakage results are considered to be of high importance for organizational security teams to consider and evaluate. During initial phase of Kubernetes growth, considering metrics as low priority could be reasonable but the current version pace of Kubernetes reveals detailed critical information at the Kube-API server level. It has become a must for organizations to adopt and extend security governance at observability layer considering the fact that user with metrics level access without a need for API server access can still view mission critical information which was intended for them. By applying post mitigation steps, the results revealed that after filtering the needed metrics, still no observability related values were lost, the dashboards retained their operational efficiency and all the needed teams having to view their intended cluster health signals.



6. Conclusion

The conversations pertaining security around Kubernetes always leaned towards the workload layer such as network policies at pod level, secret management, hardening at container/pod level. The mentioned workload layer scrutiny is not misplaced but this paper built a case upon the most fundamental component and how secured/vulnerable this component is, which cluster workloads relies upon – Kube API server. The critical three vulnerabilities evaluated here – version gaps identified during rolling upgrades, abuse at APF level to starve control plane, and critical metric leakage at shared metric layer. These vulnerabilities arise gradually during operational workload and conditions which seemed hardened during cluster initialization but later to realize they were under security risk when critical workload is growing and eventually related configuration setup had to be evolved over time, administrators overlooked if actually the API server handling all these growing requests was hardened as per the latest load to meet the growing operation needs.

The findings listed in this paper are more than fixes. The strategies suggested to overcome security risk, code fixes, and methodologies suggested to test such vulnerabilities serve as starting points for organizations to adopt, further extend and build a stronger repeatable security governance standard from time to time, rather than apply once and forget. The derived results back this up: bypassing policies dropped to zero, API controller exploitation/degradation went from six hundred percent to seventy-five percent, all known inference paths towards critical workloads were blocked completely. These were achieved by thoroughly reviewing and hardening API configurations from time to time, evaluating traffic flows and setting right APF levels for each client and treating observability and upgrade layers as real security risks/concerns rather that just another operational tasks.

The procedures needed to address these vulnerabilities pre-exist inside Kubernetes itself without need for any special tooling – Isolating APF mechanisms, implementing feature gates, reviewing and hardening admission policies, creation relabeling rules at observability layer. The part missing at organizational level was an awareness gap about these security risks and tested strategies to address them.

REFERENCES

1. Chen, L. et al. "Security challenges in container orchestration platforms: A systematic survey." Journal of Cloud Computing Security, 2024.
2. Patel, R., and Nguyen, S. "Admission control consistency in versioned Kubernetes deployments: Risks and enforcement strategies." IEEE Transactions on Dependable and Secure Computing, 2023.

3. Morrison, A., Kapoor, D., and Wu, F. "Enforcement drift during rolling upgrades in distributed control planes." ACM Symposium on Cloud Computing (SoCC), 2022.
4. Ferrara, G., and Blum, T. "Adversarial exploitation of priority queuing in cloud-native API gateways." International Journal of Information Security, 2023.
5. Zhang, H., Liu, X., and Okonkwo, E. "Performance and security analysis of API Priority and Fairness in large-scale Kubernetes clusters." USENIX Annual Technical Conference (ATC), 2023.
6. Ramirez, C., Park, J., and Lindström, M. "Multi-tenant Kubernetes security: Isolation boundaries and shared control-plane risks." Journal of Systems and Software, 2024.
7. Gupta, A., and Hoffman, B. "A CVE-driven security analysis of the Kubernetes API server across major release versions." Computers & Security, 2023.
8. Nakamura, Y., Flores, D., and Elahi, T. "Metric label cardinality as a side-channel in shared Prometheus monitoring environments." Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2023.
9. Bergström, K., and Anand, P. "Observability versus least privilege: Governing Prometheus access in cloud-native multi-tenant deployments." IEEE Security & Privacy, 2024.
10. Osei, K., Turner, M., and Yamaguchi, R. "Empirical study of RBAC misconfiguration patterns and privilege escalation pathways in production Kubernetes clusters." Network and Distributed System Security Symposium (NDSS), 2023.
11. Vasquez, E., and Singh, H. "STRIDE-extended threat modeling for Kubernetes control-plane security." ACM Transactions on Privacy and Security, 2022.
12. Keller, J., Dubois, A., and Srinivasan, V. "Load balancer-induced policy inconsistency in horizontally scaled API server deployments." IEEE International Conference on Cloud Engineering (IC2E), 2023.
13. Tran, Q., Mehta, S., and Adeyemi, O. "Zero-trust architecture for Kubernetes internals: Gaps, primitives, and a path forward." Journal of Network and Computer Applications, 2024.
14. Whitfield, C., and Cho, B. "Mapping Kubernetes control-plane hardening to regulatory compliance frameworks: An empirical gap analysis." Computers & Security, 2024.
15. Reyes, M., Schneider, L., and Okafor, N. "Adversarial modeling of Kubernetes cluster entry points with control-plane impact assessment." International Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2022.
16. Hasan, F., Kowalski, P., and Meier, T. "Version heterogeneity and security-relevant behavioral divergence in Kubernetes upgrade pipelines." IEEE Transactions on Cloud Computing, 2023.
17. Abramowitz, D., and Lee, C. "Longitudinal study of Kubernetes security configuration drift in production environments." ACM Conference on Data and Application Security and Privacy (CODASPY), 2024.
18. Johansson, E., Bello, A., and Kim, D. "Data governance in cloud-native observability pipelines: Metrics exposure risks in enterprise Kubernetes deployments." Journal of Cloud Computing: Advances, Systems and Applications, 2024.