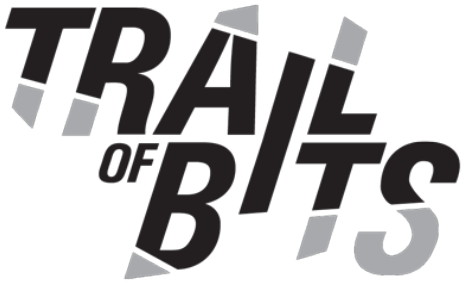


Kubernetes

Threat Model

June 28, 2019

Prepared For:
Kubernetes Security WG | *Kubernetes*

Prepared By:
Stefan Edwards | *Trail of Bits*
stefan.edwards@trailofbits.com

Introduction

The Cloud Native Computing Foundation (CNCF) tasked Trail of Bits to conduct a component-focused threat model of the Kubernetes system. This threat model reviewed Kubernetes' components across six control families:

- Networking
- Cryptography
- Authentication
- Authorization
- Secrets Management
- Multi-tenancy

Kubernetes itself is a large system, spanning from API gateways to container orchestration to networking and beyond. In order to contain scope creep and keep a reasonable timeline for threat modeling, the CNCF selected eight components within the larger Kubernetes ecosystem for evaluation:

- kube-apiserver
- etcd
- kube-scheduler
- kube-controller-manager
- cloud-controller-manager
- kubelet
- kube-proxy
- Container Runtime

In total, the assessment team found 17 issues across the various components, ranging in severity from Medium to Informational.

Key Findings

Kubernetes allows users to define policies at multiple levels of the cluster. This can impact items from network filtering through to how Pods interact with the underlying host. However, many of these policies are not applied without the correct components installed and enabled; this, unto itself, is fine. However, Kubernetes does not warn users that these policies are not applied due to missing components. This may lead to a situation wherein a user assumes they have applied a security control, when it is in fact missing. Warning users when security controls are missing or unapplied would allow them to respond by either enabling the correct components or mitigate the issue in another way. More generally, Kubernetes does not readily provide auditing information to users in a unified fashion. Components may or may not collect auditing information sufficient to track a user's path

through the system. Providing more feedback to users, administrators, and incident responders will help not only to increase the general security of Kubernetes, but also to give users a more consistent and friendly User Experience as well.

Kubernetes is also a highly networked system, with multiple communications protocols. Many of these protocols traffic in sensitive information, such as cluster secrets or credentials, and yet do not use the strongest TLS configurations possible or even use it at all. While it is difficult for attackers to intercept communications between components, it is possible in certain configurations. Therefore, always enforcing HTTPS, providing ingress and egress filtering for clusters, and using the strongest cryptographic controls possible will provide users with a secure baseline that must be modified to be insecure, rather than the other way around.

Furthermore, it must rely on traditional operating systems to do the heavy lifting of running containers and cluster services. However, it generally assumes those operating systems to be trustworthy, and that sensitive data may be exposed to the operating system freely. However, Hosts (called “Node” in the Kubernetes ontology if running a kubelet) should be treated as a separate security domain from the cluster itself: systems administrators for Hosts may not be the same systems administrators for the cluster. Sensitive data exposed to a Host could allow Internal Attackers or Malicious Internal Users to parlay their access to a single Host into wider cluster access, simply by viewing logs, processes, or environmental data. There are two general directions that Kubernetes can take:

1. Hosts must be “closed,” and all systems administrators for Hosts should be treated as privileged enough to view the data shared via IPC and environment variables.
2. Move away from exposing sensitive data to systems, and towards a model similar to a Vault or Key Management System/Hardware Security Module, wherein the system provides APIs that do not expose secrets to any other processes or persons under normal circumstances.

Either choice would be a reasonable direction for Kubernetes, however, a choice must be made. Choosing one direction or the other will ensure that controls and designs can be made that satisfy the chosen direction, and that all components understand and can adhere to this direction. A similar issue exists with Multi-tenancy: many components do not have a notion of Multi-tenancy, which is relatively loosely defined at the time of this assessment. Choosing a direction, and providing developers with guidance as to how to achieve this direction. This will strengthen the core of Kubernetes, and remove many of the current issues, wherein components do not have an answer for certain aspects of the system, because they are ill-defined.

Report Position

Kubernetes is a large, intricate system, with many security controls and design decisions having arisen from organic decisions that made sense in situ to diverse and distributed teams. This report attempts to catalog many of the discussions captured within the Rapid Risk Assessment processes. However, the raw Rapid Risk Assessment documents will be provided upon release of this report, so that the community as a whole may see the discussions and notes made during meetings.

The remainder of this report analyzes components, trust zones, data flows, threat actors, controls, and findings of the Kubernetes threat model. This was a point-in-time assessment, and reflects the state of Kubernetes, specifically version 1.13.4, at the time of the assessment, rather than any current or future state.

Methodology

This document is the result of several person-weeks worth of effort from members of the community, the Security Working Group, and the assessment team, across diagrams, documents, RRAs, Manning's Kubernetes in Action book, and Kubernetes' own documentation. It is a control-focused threat model, with a review of each component vis-a-vis the controls selected by the Security Working Group. The RRA template is provided in Appendix A: RRA Template.

Performing a threat model and architecture review of a system as large as Kubernetes proved challenging. First, we designed a dataflow for the selected components, and modified Mozilla's Rapid Risk Assessment (RRA) document to focus on the selected controls. Next, we pre-filled sections of the documents for each component, based on our understanding of the component and online documentation. Then, we polled the community for feedback, and held remote meetings with members of the community to correct any gaps in the RRA documents and to discuss the impact of each control within the selected component. Once the RRA had been filled out by a group of community members, a different group of community members was selected to peer review the document for accuracy.

We would like to thank all of the members of the community who came together to donate their time to us, in order to discuss and review areas of Kubernetes' design, and provide holistic information that can make Kubernetes as a whole better.

Components

The Kubernetes architecture is composed of multiple components, many of which are stand-alone binaries written in Go. The following table describes the eight selected Kubernetes components:

Component	Description
kube-apiserver	A RESTful web server that handles coördination of all aspects of a cluster. Specifically, it accepts client requests for updating all other components within a cluster. These requests are authenticated, authorized, processed, and then stored within etcd for further processing and use. Clients may subscribe to topics in order to be notified of changes that are relevant to their interests; for example, a kubelet may listen for Pod scheduling events that require the kubelet's action.
etcd	A key-value store that leverages gPRC and TLS (potentially with two-way, or client-authenticated TLS), used to store the most sensitive data within a cluster. Access to etcd should be restricted to as few users as possible. Generally, unrestrained access to etcd is considered "root" (or administrative) access to the cluster itself.
kube-scheduler	A component that reads from Pod descriptions and schedules the Pods to nodes based on a scheduling algorithm and resource constraints. In practice, this means the scheduler "listens" to new Pod creation on the API server, reviews the node list for potential resource allocation and rules, and updates the Pod to be assigned to a specific node within the API server. A further process, namely the kubelet, will actually handle the task of instructing the container runtime to run and execute the Pod.
kube-controller-manager	A daemon that listens for specific updates within the API server, taking action, and storing its own updates within the API server itself. The purpose of the daemon is to run "controllers" within an infinite loop, with each controller attempting to keep the state of the cluster consistent. This works by way of a call-back listener loop, and comparison of current cluster state with the desired state of the cluster as described by developers and administrators.
cloud-controller-manager	A daemon with similar purpose to kube-controller-manager, but instead of focusing on generic components within Kubernetes, it focuses on maintaining consistency with

	cloud-platform-specific components that back a Kubernetes cluster.
kubelet	A worker component tasked with all aspects of node operations within a worker. It interacts with the Container Runtime, listens for Pod scheduling and related events on the API server, and updates the API server as to Pod availability, resource usage, and general status. It is also the endpoint the API server reaches out to for logs and other updates from nodes and Pods within the cluster.
kube-proxy	A component to help, along with the Container Networking Interface (CNI), facilitate Kubernetes' transparent model of networking. Kubernetes requires that all Nodes and Pods be able to communicate without a (visible) Network Address Translation (NAT). kube-proxy utilizes items such as iptables and also serves to proxy or pass-thru traffic in order to ensure that all containers, Pods, and nodes are able to communicate with one another as if they were on a single network.
Container Runtime	A group of components that allow for the direct execution of containers within a cluster. This includes the necessary operating system integrations (such as control groups on Linux), configuration settings, and Kubernetes interfaces to a container system, e.g. Docker, cri-o, containerd, ...

Planes

Kubernetes itself is divided (roughly) into two “planes,” or groupings of components. The following table describes each plane, and groups the aforementioned components:

Plane	**Description**	**Components**
Control Plane	The Control Plane (CP) controls the state of the cluster and ensures that the desired components are running as specified by the end user. Generally, these are grouped as Masters (technically, API server, etcd, and related components) as well as the kubelet (which, whilst part of the CP, actually runs on every Node).	kube-apiserver, kube-scheduler, kube-controller-manager, cloud-controller-manager , kubelet, etcd
Data Plane	The worker, on the other hand, executes the actual Pods and containers that make up a client’s applications. These components focus on running and networking Pods and their associated containers.	Kube-proxy, CNI, CRI, Pods

Trust Zones

Systems include logical “trust boundaries” or “zones” in which components may have different criticality or sensitivity. Therefore, in order to further analyze the system, we decompose components into zones based on shared criticality, rather than physical placement in the system. Trust zones capture logical boundaries where controls should or could be enforced by the system, and allow designers to implement interstitial controls and policies between zones of components as needed.

Zone	Description	Included Components
Internet	The externally facing, wider internet zone.	kubectl, application clients
API Server	The central coordinator for the system, exposed only as much as needed for access from administrators with kubectl.	kube-apiserver
Master Components	Internal portions of the Master node that work via callbacks and subscriptions to the API server.	kube-controller-manager cloud-controller-manager
Master Data	The data layer of the API server and master server(s) themselves. This boundary contains items such as Consul or etcd, and is tantamount to “root” or administrative access to the cluster when accessed in an uncontrolled fashion.	etcd
Worker	The worker zone within a cluster includes all the components required to run and network containers.	kubelet, kubeproxy
Container	The container zone includes the actualLinux containers.	Container Runtime

Trust Zone Connections

Trust zones are only useful when we understand the data that flows between zones, and why.

Originating Zone	Destination Zone	Data Description	Connection Type	Authentication Type
Internet	API Server	kubectl administration functionality, which could be a VPN or other bastion host with direct access to the API server.	HTTPS	Verified and possibly two-way TLS
Internet	Container	Clients of the actual applications within the Kubernetes Worker Nodes.	Various (based on application)	N/A
API Server	Master Data	kube-apiserver retrieving and storing data from a key-value store such as etcd or Consul.	HTTPS	Verified
Master Components	API Server	Items such as the Scheduler, Controller Managers, Replication Manager, retrieving and updating items managed by the API server.	HTTPS/Internal Callbacks	Verified
API Server	Worker	Log and status retrieval from the API server to the individual Worker Node's kubelet.	HTTP	Unverified
Worker	API Server	Worker nodes' kublet must communicate with the API server for new Pod	HTTPS	Verified and possibly two-way TLS

		allocations, to update status Pods, and so on.		
Worker	Container	Scheduling containers for execution, handled by the kubelet, within a host node.	Interprocess Communications	N/A

Threat Actors

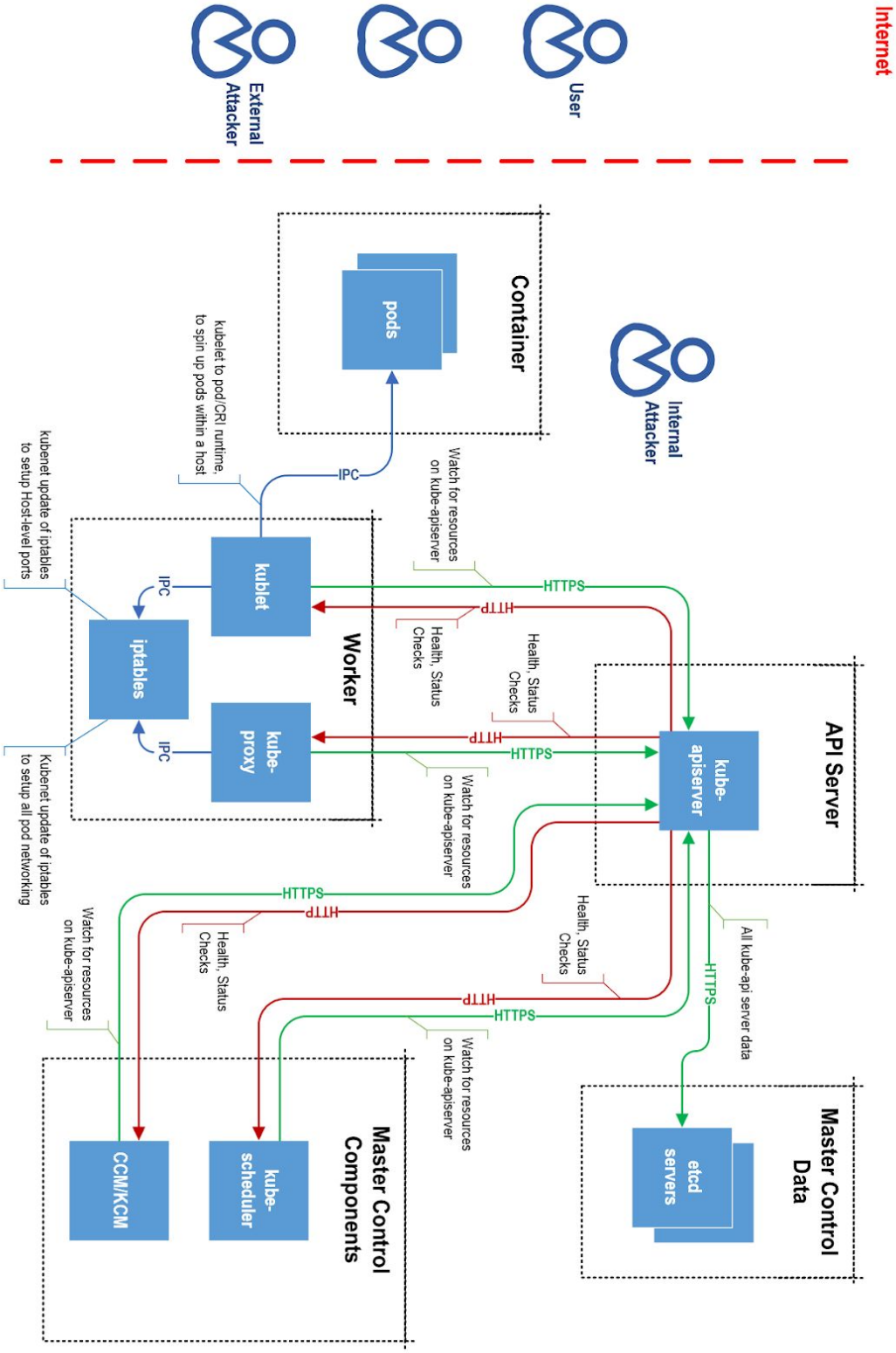
Similarly to Trust Zones, defining malicious actors ahead of time is useful in determining which protections, if any, are necessary to mitigate or remediate a vulnerability.
We will use these actors in all subsequent findings from the threat model. Additionally, we define other “users” of the system, who may be impacted by, or enticed to undertake, an attack. For example, a Confused Deputy attack such as Cross-Site Request Forgery would have a normal user as both the victim and the potential direct attacker, even though a secondary attacker enticed the user to undertake the action.

Actor	**Description**
Malicious Internal User	A user, such as an administrator or developer, who uses their privileged position maliciously against the system, or stolen credentials used for the same.
Internal Attacker	An attacker who had transited one or more trust boundaries, such as an attacker with container access.
External Attacker	An attacker who is external to the cluster and is unauthenticated.
Administrator	An actual administrator of the system, tasked with operating and maintaining the cluster as a whole.
Developer	An application developer who is deploying an application to a cluster, either directly or via another user (such as an Administrator).
End User	An external user of an application hosted by a cluster.

Additionally, defining attackers' paths through the various zones is useful when analyzing potential controls, remediations, and mitigations that exist within the current architecture:

Actor	Originating Zone	Destination Zone(s)	Description
Malicious Internal User	Any	Any	Malicious Internal Users are often privileged and have access to a wide range of resources. Therefore, controls must be in place to ensure users are authorized to undertake an action and log all actions within the system, for strong non-repudiation of actions.
Internal Attacker	Container	Containers in another namespace, Worker, API Server, Master Components, Master Data	Attackers who transit external boundaries and attain position on an internal container will seek to escalate privileges by accessing items in the API Server or Master Data zones, or parlay their access to other internal components of other Worker nodes.
External Attacker	Internet	Container, Worker, API Server	External Attackers will seek to transit network edges in order to become Internal Attackers, or use exposed API Server functionality to escalate privileges.

Dataflow


Internet
External Attacker
User
Internal Attacker
Container
pods
kubelet to pod/CRI runtime, to spin up pods within a host
IPC
kubenet update of iptables to setup Host-level ports
kublet
iptables
kube-proxy
IPC
IPC
Worker
Watch for resources on kube-apiserver
HTTPS
HTTP
Health, Status Checks
Health, Status Checks
HTTP
HTTPS
Watch for resources on kube-apiserver
API Server
kube-apiserver
HTTPS
HTTP
Watch for resources on kube-apiserver
Health, Status Checks
Kubenet update of iptables to setup all pod networking
HTTP
HTTPS
Health, Status Checks
All kube-api server data
HTTPS
HTTP
HTTPS
Watch for resources on kube-apiserver
Master Control Components
CCM/KCM
kube-scheduler
Master Control Data
etcd servers

Control Summary

Committee on National Security Systems (CNSS) Instruction (CNSSI 4009 defines "security control" as: `The management, operational, and technical controls (i.e., safeguards or countermeasures) prescribed for an information system to protect the confidentiality, integrity, and availability of the system and its information.` Controls are grouped by type or *family*, which collect controls along logical groupings, such as Authentication or Cryptography. This assessment will focus on six primary control families, per the request of the Security Working Group:

Family Name	Description
Authorization	Related to authorization of users and assessment of rights.
Authentication	Related to the identification of users.
Cryptography	Related to protecting the privacy or integrity of data.
Secrets Management	Related to the handling of sensitive application secrets such as passwords.
Networking	Related to the protocols and connections between cluster and application components.
Multi-tenancy	Related to the safe handling of two or more separate organizational groups within a cluster.

Additionally, we will keep the following families in mind throughout our review:

Family Name	Description
Auditing and Logging	Related to auditing of actions or logging of potential security events.
Configuration	Related to secure configurations of servers, devices or software.
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.
Undefined Behavior	Related to undefined behavior triggered by the program.

Our review assessed the controls along the following criteria:

- Strong: controls were well implemented, centrally located, non-bypassable, and robustly designed.
- Satisfactory: controls were well implemented, but may be weakened by vulnerabilities or are diffuse in location.
- Adequate: controls were implemented to industry-standard best practice guidelines.
- Weak: controls were either partially unimplemented, applied, or contained flaws in their design or location.
- Missing: an entire family of control was missing from a component.
- Not Applicable: this control family is not needed for protecting the current component.

kube-apiserver

Control Family	Strength	Description
Authorization	Satisfactory	kube-apiserver authorizes requests as a part of the normal request pipeline, in a centralized fashion. So long as outdated modes of authorization, namely ABAC, are not used, the authorization controls within this core component are relatively strong. One other central concern aside from ABAC's inclusion is the presence of webhooks, which may reach out to other, third-party components without much visibility to the kube-apiserver itself.
Authentication	Strong	kube-apiserver authenticates requests as part of the normal request pipeline, in a centralized fashion. So long as outdated modes of authentication are not used, the authentication controls within its core and critical subcomponents are relatively strong.
Cryptography	Adequate	kube-apiserver employs many cryptographic systems: from TLS connections all the way through to being the root Certificate Authority for at least two Public Key Infrastructures within a cluster. However, multiple connections within the system avoid using TLS for communication secrecy by default and some services do not authenticate their connections when they do so. For example, by default kubelet's connections do not utilize a certificate generated by the kube-apiserver, but rather an unvalidated self-signed one.
Secrets Management	Adequate	kube-apiserver is the heart of all information flow within a cluster: all information must be passed through and processed by the kube-apiserver. For the most part, secrets are safely handed by the kube-apiserver to etcd in a secure fashion. However, encryption of secrets is not a default action of the server, and must be configured by a command line switch, providing the perception of security, rather than actual security to the system.
Networking	Weak	kube-apiserver is a highly-networked component: all other components within the cluster communicate with it, and it must communicate with external services, like webhooks and components' own health

		reporters. However, there are no controls limiting egress and ingress to the kube-apiserver itself, and ancillary protections must be used in order to ensure that it is not communicating with external third parties in a fashion unintended by cluster administrators.
Multi-tenancy	Satisfactory	kube-apiserver is the arbiter of tenant interactions within a cluster. However, the general consensus of teams during the assessment was that isolation was strong enough for intra-organizational segmentation, but not for true multi-tenant situations. For example, minimal protections are in place for “noisy neighbor” and similar scenarios within a cluster.

etcd

Control Family	Strength	Description
Authorization	Not Applicable	etcd, as deployed by Kubernetes, does not support authorization to separate out user actions. It should only be accessed by kube-apiserver and components of similar sensitivity.
Authentication	Satisfactory	etcd can be configured to use two-way (or client-side) TLS as an authentication mechanism. However, as it is intended to be run from its own cluster accessible only to kube-apiserver(s), minimal authentication requirements are needed other than network segmentation.
Cryptography	Not Applicable	etcd does not store data encrypted at rest, and instead relies on kube-apiserver to enforce cryptographic constraints. However, recommendations are made within the report to strengthen cryptographically-secure hashing operations within file system operations, such as the Write-Ahead Log (WAL).
Secrets Management	Not Applicable	While etcd stores secrets for the cluster, etcd only processes credentials sufficient to communicate with kube-apiserver. All other secrets are handled by kube-apiserver itself, and merely stored within etcd.
Networking	Not Applicable	Any etcd process is intended to be segmented by network and security controls from the rest of the Kubernetes cluster. These controls are external to etcd, and thus outside of the scope of this review.
Multi-tenancy	Not Applicable	etcd maintains no awareness of multiple tenants within the system, which is instead handled by kube-apiserver.

Kube-scheduler

Control Family	Strength	Description
Authorization	Not Applicable	kube-scheduler includes concepts that must be restricted by role or attribute to specific users. However, enforcement is the domain of the kube-apiserver, rather than kube-scheduler.
Authentication	Not Applicable	kube-scheduler does not handle authentication directly, but rather relies on kube-apiserver to be the arbiter of authentication. The sole item of authentication is the handling of client credentials, which are handled in the standard way: credentials are passed in via command line argument to the kube-scheduler binary on initial execution.
Cryptography	Not Applicable	kube-scheduler does not handle cryptography directly, but rather relies on kube-apiserver to be the arbiter of cryptography.
Secrets Management	Not Applicable	kube-scheduler does not handle secrets directly, but rather relies on kube-apiserver to be the arbiter for their correct storage. The sole exception is authentication credentials, which are passed in via command line argument to the binary on initial execution, and is a general finding in Kubernetes as a whole, rather than specific to kube-scheduler.
Networking	Adequate	kube-scheduler should mainly communicate with kube-apiserver, and thus minimal controls are required. However, ingress and egress controls are maintained externally to kube-scheduler. Attackers may be able to access health reporting and related information from internal positions within the cluster.
Multi-tenancy	Satisfactory	kube-scheduler makes heavy use of namespaces and other multi-tenant items (such as Pods for isolation boundaries, and RBAC for authorization controls) when scheduling cluster workloads. However, attackers may be able to abuse items such as anti-affinity within the cluster to control more than their fair share of cluster resources.

kube-controller-manager and cloud-controller-manager

Control Family	Strength	Description
Authorization	Adequate	KCM and CCM are actually a passel of components with various levels of authority. Items such as service account controller have permission to write to their policies, which may be used to escalate privileges. Additionally, further segmentation should be added to separate out sensitive functionality from functionality at a lower privilege level, so as to ensure that sensitive functions are not maliciously or mistakenly interacted with by other components.
Authentication	Not Applicable	KCM and CCM do not handle authentication directly, but rather rely on kube-apiserver to be the arbiter of authentication.
Cryptography	Not Applicable	KCM and CCM do not handle cryptography directly, but rather rely on kube-apiserver to be the arbiter of cryptography.
Secrets Management	Weak	KCM and CCM handle a wide number of secrets, which may be shared with the KCM and CCM systems by various means. These include environment variables, command-line arguments, and Kubernetes secrets. However, controls are diffuse, being written in situ for each KCM or CCM component.
Networking	Adequate	KCM and CCM must be able to talk to a large number of external-to-the-cluster components, such as cloud providers' infrastructure. However, controls themselves are diffuse, and implemented in situ within each KCM or CCM module.
Multi-tenancy	Not Applicable	KCM and CCM do not directly handle multi-tenant isolation. This could be problematic going forward, as KCM and CCM components could interact with namespaces that were not intended to have access to cloud or other provider boundaries.

kubelet

Control Family	Strength	Description
Authorization	Strong	Unless configured improperly, kubelet delegates all authorization requests to the kube-apiserver(s) of the cluster. This allows all requests that must access privileged information served by the kubelet's HTTP(S) servers to be authorized by the central authority within the system.
Authentication	Strong	Unless configured improperly, kubelet also delegates all authentication requests to the kube-apiserver(s) of the cluster. This allows all requests that must access privileged information served by the kubelet's HTTP(S) servers to be authenticated by the central authority within the system.
Cryptography	Not Applicable	kubelet does not handle cryptography directly, but rather relies on kube-apiserver to be the arbiter of cryptography.
Secrets Management	Adequate	kubelet is, like kube-apiserver, uniquely privileged within the system to see a large amount of secret information in an unencrypted form. For the most part, this information is handled safely with strong controls. However, bootstrap certificates are written unencrypted to the file system, and are not removed automatically.
Networking	Weak	kubelet includes a large number of services that are neither ingress-restricted nor controlled by delegated authentication. This is problematic as these services reveal information such as Pod specifications to any attacker capable of accessing the port.
Multi-tenancy	Not Applicable	kubelet does not handle multi-tenancy directly, but rather relies on kube-apiserver to be the arbiter of multi-tenant isolation.

kube-proxy

Control Family	Strength	Description
Authorization	Not Applicable	kube-proxy does not handle authorization directly, but rather relies on kube-apiserver to be the arbiter of authorization.
Authentication	Not Applicable	kube-proxy does not handle authentication directly, but rather relies on kube-apiserver to be the arbiter of authentication.
Cryptography	Not Applicable	kube-proxy does not handle cryptography directly, nor does it need access to sensitive information in general.
Secrets Management	Not Applicable	kube-proxy does not handle secrets directly, but rather relies on kube-apiserver to be the arbiter for their correct storage. The sole exception is authentication credentials, which are passed in via command-line arguments to the binary on initial execution. This is a general finding for Kubernetes as a whole, rather than specific to kube-proxy.
Networking	Strong	kube-proxy has strong, centralized controls that ensure correct interaction with both the Linux kernel (via iptables or ivps) and the networking configuration specified by cluster clients.
Multi-tenancy	Not Applicable	kube-proxy does not handle multi-tenancy directly, but rather relies on kube-apiserver to be the arbiter of multi-tenant isolation.

Container Runtime

Control Family	Strength	Description
Authorization	Not Applicable	Container Runtime does not handle authorization directly, but rather relies on kube-apiserver and kubelet to be the arbiters of authorization.
Authentication	Satisfactory	Container Runtime largely relies on kube-apiserver and kubelet to be the arbiter of authentication. However, there is a single control that relies on a slightly weakened authentication mechanism with a short lifetime, which slightly impacts the strength of the authentication control.
Cryptography	Not Applicable	Container Runtime does not handle cryptography directly, but rather relies on kube-apiserver and kubelet to be the arbiters of cryptography.
Secrets Management	Adequate	Container Runtime largely relies on kubelet to handle secrets. However, authenticated Pod repositories' credentials are exposed to the wider host, despite being secret within the Pod specification itself.
Networking	Adequate	Container Runtime is responsible for handling image retrieval within the cluster. However, it does not have egress filtering, which could be used to impact the cluster itself, especially when HTTP is used.
Multi-tenancy	Not Applicable	Container Runtime does not handle multi-tenancy directly, but rather relies on kube-apiserver and kubelet to be the arbiters of multi-tenant isolation.

Kubernetes-wide findings

Findings in this section impact Kubernetes as a whole. These are generally design issues that are shared amongst all components, despite not sharing code for the control or design. Broadly, these findings trend towards information disclosure, channel security (cryptography and networking), and the application of user configuration.

1. Policies may not be applied

Severity: Medium
Type: Configuration
Difficulty: Medium
Finding ID: TOB-K8S-TM01

Description
Kubernetes allows users to define policies, such as NetworkPolicy and PodSecurityPolicy, to restrict sensitive actions of specific components. For example, a NetworkPolicy is a YAML document intended to restrict the communications flow between Pods, in an effort to implement egress and/or ingress filtering. Users may implement such policies so as to enforce communication boundaries within a cluster, and ensure that Pods cannot communicate across sensitivity levels or tenants.

However, policies are not always enforced, and may fail silently. Kubernetes does not warn users when policies are applied which may not be enforced without further configuration changes or components. For example, PodSecurityPolicy requires an additional Admissions Controller to be configured for execution, and NetworkPolicy requires a Container Networking Interface (CNI) that can actually process and accept policies as configuration. Neither object will warn the user that the policy has not been applied, leading to a false sense of security.

Justification
The difficulty is medium for the following reasons:

- A user must intend to use one of the following policies without ancillary controls in place (such as external firewalls to restrict Pod communication).
- An attacker must have position sufficient to exploit the missing policy.

The severity is medium, for the following reasons:

- An attacker must have a secondary exploit in order to impact the cluster.
- Users are not alerted to the missing policy, meaning they cannot effectively monitor for situations that may arise from missing policies.

Recommendation
Short term, clearly document all locations within a Kubernetes cluster that may accept a policy or configuration that is not applied. This will impact at least kube-apiserver and CNI.

Long term, alert users to situations wherein policies may not be applied. Wherever possible, do not apply configurations, such as Pod specifications, when security-related policies cannot be applied. This will directly alert users to situations that fail to apply security controls they expect, and allow them to take the appropriate configuration action as needed to apply the desired control.

References

- NetworkPolicy Prerequisites
- PodSecurityPolicy “Enabling Pod Security Policies” and “Authorizing Policies” sections

2. Insecure TLS by default

Severity: Medium
Type: Cryptography
Difficulty: Medium
Finding ID: TOB-K8S-TM02

Description
Kubernetes allows components to communicate via Transport Layer Security (TLS), with kube-apiserver serving as the root Certificate Authority (CA) for multiple Public Key Infrastructures (PKIs) within a cluster. However, multiple components within the system do not use secure TLS configurations and instead disable certificate verification and use self-signed certificates. Attackers may utilize these insecure defaults to man-in-the-middle (MITM) connections between cluster components, without the knowledge of either cluster components or administrators.

Justification
The difficulty is high for the following reasons:

- An attacker must have sufficient position to MITM connections.
- Clusters will generally have other ancillary controls, such as cloud-hosting providers or edge firewalls, to prevent the most eager attackers from affecting this attack.
- MITM attacks tend to be noisy, or cause other failures within a system, limiting the ease with which an attacker may keep this position.

The severity is medium, for the following reasons:

- Attackers can only MITM connections that do not verify certificates.
- By default, all communications with kube-apiserver are verified, leaving only communications between components (such as kube-apiserver to kubelet) open for MITM.

Recommendation
Short term, change the default for kubelet to use verified TLS for all communications, as opposed to the default use of self-signed certificates.

Long term, ensure that all components within the cluster use kube-apiserver-generated certificates, and verify all connections between components. This will ensure that all data shared throughout the cluster is shared only with intended parties, and connections will fail should any MITM be detected.

References

- Kubelet configuration options which mention the self-signed certificate bootstrap.

3. Most components accept inbound HTTP

Severity: Medium
Type: Cryptography
Difficulty: Medium
Finding ID: TOB-K8S-TM03

Description
Many components in Kubernetes may have changes in state, health, or general status information that should be reported. In order to handle this, Kubernetes uses HTTP as a universal mechanism for data collection: components serve standard HTTP routes which serve data in standard formats consumable by kube-apiserver. However, sensitive information is also presented by at least one component (kubelet), and all others present information over unencrypted, unauthenticated interfaces. An Internal Attacker with sufficient position could sniff traffic or more easily man-in-the-middle requests between components.

Justification
The difficulty is medium for the following reasons:

- Attackers must either have sufficient position or have discovered missing secondary controls to affect the attack.
- Internal attackers, or External Attackers in a misconfigured cluster, can easily request data.

The severity is medium for the following reasons:

- Sensitive data is rarely stored unencrypted within podspecs.
- Data is read only, even if it exposes secrets or allows attackers to transit trust boundaries (such as an Internal Attacker suddenly being able to see Pod specs).

Recommendation
Short term, move all components to HTTPS, and disable HTTP unless specifically requested by cluster administrators.

Long term, do not leak sensitive information without delegated authentication from kube-apiserver. This would include Pod specs and even certain types of statistics, such as number of Pods running on a given node. Minimizing the information given out to unauthenticated requests can reduce information available to attackers for lateral movement.

4. Most components do not enforce outbound HTTPS

Severity: Medium
Type: Networking
Difficulty: Medium
Finding ID: TOB-K8S-TM04

Description
Kubernetes interacts with external resources such as Webhooks and container repositories for multiple resource types. Components such as kubectl, kube-apiserver, and the Container Runtime fetch resources all have reasonable use cases to request data from external resources.

However, no component requires secured connections for external resources; an attacker with sufficient position could man-in-the-middle these connections and change the response, or issue requests to third-party websites which are not under the control of the organization.

Justification
The difficulty is medium for the following reasons:

- Attackers must have sufficient position or privilege in order to affect this attack.
- Minimal tooling is required, especially for Malicious Internal Users, to launch this attack, lowering difficulty.

The severity is medium for the following reasons:

- Attackers can launch attacks on external resources.
- Attackers can modify resources to be executed by the cluster.
- Attackers with this position can launch other, more fruitful attacks, lowering severity.

Recommendation
Short term, enforce the use of HTTPS throughout the lifecycle of a retrieval request, and ensure that no resources, regardless of source, are fetched from insecure communications channels.

Long term, design egress controls that can be applied to components such as Container Runtime and kube-apiserver, to ensure that these resources communicate with external resources intended by administrators. This should include validation of remote URLs, hosts, and other metadata, to ensure that cluster users can only access intended resources.

5. Credentials exposed in environment variables and command-line arguments

Severity: Low
Type: Data Exposure
Difficulty: High
Finding ID: TOB-K8S-TM05

Description
Kubernetes uses credentials and secrets throughout the cluster. These may be simple bootstrap credentials for components such as Container Runtime, or more complex tokens passed in via environment variables. Regardless, credentials are exposed to a wider audience than intended when supplied to components in this fashion: Internal Attackers with Host position or Malicious Internal Users may have position sufficient to see command line arguments (such as via `ps` or `top`) or environment variables. Additionally, secrets passed as command-line arguments are much more likely to end up in world-readable system logs on the host, aggregated logs for the organization and sent to third party log service providers. This would allow an attacker with sufficient position to parlay their access to a Host into wider cluster access.

Justification
The difficulty is high for the following reason:

- An Internal Attacker or Malicious Internal User must have privileged access to hosts in order to see all processes' arguments and environments.

The severity is low for the following reasons:

- Attackers could parlay access to the Host into wider cluster access.
- Attackers with this level of access could likely impact the cluster in other ways, lowering severity.

Recommendation
Short term, document all the ways that items are passed via command-line arguments and environment variables, so that end users understand their exposure. This should include Inter-Process Communication (IPC) mechanisms that spawn new processes, such as kubelet interacting with Container Runtime.

Long term, move away from exposing sensitive credentials via mechanisms that may be revealed to unintended third parties.

6. Names of secrets are leaked in logs

Severity: Low
Type: Logging

Difficulty: High
Finding ID: TOB-K8S-TM06

Description
Kubernetes includes logging throughout components. However, multiple components log the names of secrets provided to them. An attacker with access to cluster logs, an Internal Attacker with access to host logs, or a Malicious Internal User with access to aggregated logs in a Security Information and Event Management (SIEM), could see the names of secrets within a cluster, which could themselves be sensitive, depending on the user and application. Furthermore, anywhere that command line arguments are logged may include secrets or names of secrets, depending upon the component.

Justification
The difficulty is high for the following reasons:

- Attackers must have sufficient privilege or position to access logs.
- Sensitive names (such as internal projects) must be used by Developers or Administrators.

The severity is low for the following reasons:

- In and of themselves, the names of secrets may only reveal certain aspects of a program or application, rather than actionable details.
- Attackers with this level of access may be able to access other details of an application that are more impactful, such as Pod configurations.

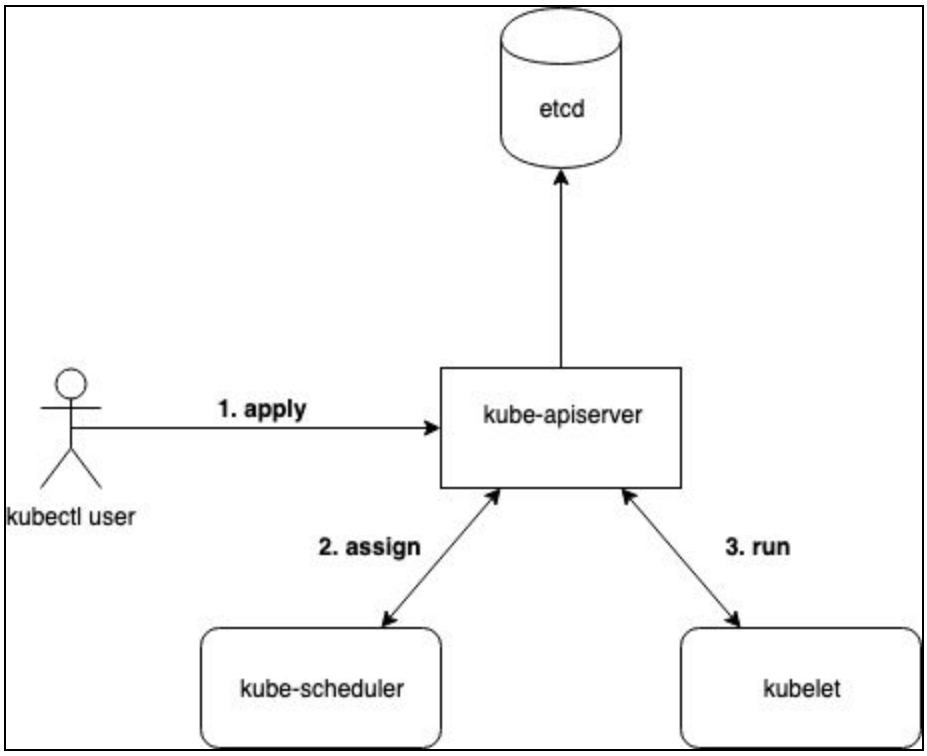
Recommendation
Short term, document all the ways in which data may be leaked within the cluster's logs, so that users are at least aware of these locations and may filter manually.

Long term, implement filtering akin to logging filters in Java frameworks, such as Simple Logging Faces For Java (SLF4J). This will allow users to apply custom filters that can filter logs in ways that the Kubernetes team cannot anticipate, without interrupting the normal operation of logging within the cluster.

kube-apiserver findings

kube-apiserver is the heart of the cluster: it provides the final say on the cluster's state, and all updates are coordinated through watching for new resources and updating them directly.

It works by providing a central ReST server that all other components within the system access via HTTPS. These requests are authenticated by various Authenticators (components run within the kube-apiserver to authenticate requests), authorized by Authorizers, and processed by other elements, such as Admissions Controllers and Resource Validators. Once a request has been processed by all subcomponents, it is stored in etcd, where it can be later retrieved by other cluster components "watching" for updates. For example, a simple flow is as follows:



1. A client updates a Pod definition via `kubectl`, which is itself a `POST` request to the kube-apiserver.
2. The scheduler watches for Pod updates via an HTTP request to retrieve new Pods.
3. The scheduler then updates the Pod list via a `POST` to the kube-apiserver.
4. The node's `kubelet` retrieves a list of Pods assigned to it via an HTTP request.
5. The node's `kubelet` then updates the running Pod list on the kube-apiserver.

7. No non-repudiation or audit of user actions by default

Severity: Medium
Type: Audit and Logging
Difficulty: Low
Finding ID: TOB-K8S-TM07

Description
The kube-apiserver is the heart of the cluster: all transactions must pass through its handlers and be served again to other cluster components. In this way, kube-apiserver ensures consistent cluster state across all components: creation, modification, and deletion are all coördinated via this central service. However, kube-apiserver does not keep a log of users' actions without debug mode being enabled, meaning that reconstructing an attacker's path through the cluster is extremely difficult.

Justification
The difficulty is low for the following reasons:

- Attackers do not require special tools or privileges to interact with the kube-apiserver.
- Internal Attackers or Malicious Internal Users already have sufficient privileges to interact with kube-apiserver to some degree.

The severity is medium for following reasons:

- Attackers must have sufficient privileges to undertake a sensitive action, or have a secondary exploit.
- In general, this is not vulnerability unto itself, but rather represents a location where incident responders would not have sufficient information to properly respond to an attack.

Recommendation
Short term, document that secondary logging mechanisms must be used in cases that need strong non-repudiation and audit controls. This will ensure that at least users who require this functionality will not be surprised that it is missing.

Long term, add logging sufficient to track a user's action across the cluster. This could be as simple as tracking events solely within kube-apiserver, or could coördinate across the cluster as a whole. We recommend that at least all authenticated events, including delegated authentication from kubelet, should be logged and retrievable from a central location within the cluster. This will allow incident responders to audit from a central location a user's action within the cluster.

8. Secrets not encrypted at rest by default

Severity: Low
Type: Cryptography
Difficulty: High
Finding ID: TOB-K8S-TM08

Description
Kubernetes allows users to define secrets, which can be anything from authentication credentials to application configuration options. While secrets are only rarely exposed outside of etcd (such as to kubelets or to the Container Runtime), they are not by default encrypted at rest. An attacker with access to etcd data files, such as via a backup, will have full access to secrets in an unencrypted state. Furthermore, the `--encryption-provider-config` accepts an `identity` provider (the default), which does not actually encrypt data, but rather simply returns the secret unencrypted. Users may misconfigure the ordering of providers, and accidentally send unencrypted data to etcd or other storage locations.

Justification
The difficulty is high for the following reasons:

- Attackers must have access to etcd data files sufficient to read secrets unencrypted.
- etcd is segmented from the rest of the cluster, with heavily restricted file system permissions.

The severity is low for the following reasons:

- In and of itself, this does not increase the risk of exposure more than compromising kubelet or other cluster infrastructure that handles secrets.

Recommendation
Short term, document ideal configurations for various levels of security, and provide standard configurations for users.

Long term, move towards some reasonable default for users besides the identity provider, and warn users when the identity provider is used either as a standalone or within a chain of providers. This will ensure that users cannot accidentally include the identity provider.

References

- Kubernetes cluster administration guide section on encrypting data at rest

etcd findings

etcd is the main storage engine of all cluster-related data. Everything that kube-apiserver wishes to coördinate across hosts and components eventually makes its way into etcd. Additionally, the documentation makes it clear: "Access to etcd is equivalent to root permission in the cluster."

etcd works as a ReST server, accepting JavaScript Object Notation (JSON) objects from clients, and storing these objects at a location requested by the client. Within Kubernetes, these objects are generally stored within the /registry route, and etcd processes all objects processed by kube-apiserver. In order to keep up with the demand for fast reading and writing similar to a traditional database, etcd may be deployed in a separate cluster. It uses the RAFT consensus algorithm to ensure that data is presented and available to all nodes within the cluster eventually.

9. Write-Ahead Log does not use signatures for integrity checking

Severity: Very Low
Type: Data Validation
Difficulty: High
Finding ID: TOB-K8S-TM09

Description
etcd is a high-performance key-value store used to reify all state data within a Kubernetes cluster. As part of this design, it uses a Write-Ahead Log (WAL) file, which is meant to serve as an atomic commit file for changes; should etcd fail before the changes are written to the main database, they can be reconstructed from the WAL file. However, the WAL file does not use cryptographic signatures to ensure validity of data. An attacker with access to the WAL file may tamper with the file without leaving a trace. Furthermore, copying over a noisy or lossy connection could result in data corruption that cannot be detected until a later point in time.

Justification
The difficulty is high for the following reasons:

- An attacker must transit multiple trust boundaries to impact a single etcd node.
- In a multi-master configuration, in order to truly impact the cluster, the attacker must repeat this attack across multiple nodes.

The severity is very low for the following reasons:

- Attackers can only impact one etcd at a time.
- A consensus algorithm is used specifically to prevent attacks with corrupted, incorrect, or outdated data.
- An attacker must attack a plurality of nodes within a cluster at the same time.

Recommendation
Short term, experiment with modes of adding cryptographically secure validation to the WAL file generation. This could be as simple as hashing each entry prior to committing to the WAL, or using something akin to Linked Timestamping, wherein each entry is hashed with the contents of the current entry and the hash of the previous entry. Furthermore, keyed hashes could be used to ensure that a specific etcd node has created and validated data. Then, when the WAL file is committed to both the data and snapshot files, the sum toto of the entries contained within the WAL file may also be hashed. Balancing entry hashing for the faster WAL files versus total file hashing for snapshots and beyond will be key to maintaining relative performance whilst also ensuring valid data.

Long term, any added validation must be tested with larger datasets in normal clusters, to ensure that etcd maintains performance, even with the added validation and security. We recommend a gradual approach of adding validation to portions of larger clusters, so that nodes with validation may be compared to nodes without it.

10. Mutual TLS is not the default

Severity: Very Low
Type: Authentication

Difficulty: High
Finding ID: TOB-K8S-TM10

Description
etcd is the holder of cluster state within Kubernetes: additions, changes, and updates are eventually stored in its data repositories. As such, authenticating who is communicating with etcd is an important task, and while etcd supports mutual (or client-side) TLS, it is not the default. An attacker who had transited network boundaries could interact with etcd without further impedance.

Justification
The difficulty is high for the following reasons:

- An attacker must transit multiple trust boundaries in order to affect sufficient position for this attack.
- etcd is specifically segmented from the rest of the cluster in order to prevent such accesses, further increasing the difficulty for attackers.

The severity is very low for the following reasons:

- An attacker with the ability to transit multiple trust boundaries could also likely steal authentication credentials used to secure two-way TLS.
- In and of itself, this represents defense in depth for those situations where etcd is not completely segmented from the rest of the network or fully from the kube-apiserver host.

Recommendation
Short term, fully document how two-way TLS may be fully enabled within etcd. The current documentation provides a simple example, but more automated or robust examples would be helpful to users.

Long term, support mutual TLS by default, and do not allow communications with etcd that are unauthenticated by client TLS certificates. Furthermore, do not use Basic or Digest Authentication for this process, as these are outdated and insecure.

kube-scheduler findings

kube-scheduler is tasked with matching Pods to hosts that can execute their workloads. This is a surprisingly complex task, as the match is based upon a variety of criteria, including the Pod's own specification of what it needs to execute its work.

In order to do this, kube-scheduler operates like most other components within the cluster: it polls kube-apiserver for new Pods and any host changes reported by the various kubelets within the system, and attempts to match new Pods to free or more free kubelets, which then go about executing the Pod with help from the Container Runtime.

Furthermore, it should be noted that there may be multiple kube-schedulers configured for a single cluster, each with a different name and set of parameters. kube-schedulers are supposed to be coöperative, however they needn't be; nothing in the system forces kube-schedulers to only act upon Pod specs with a matching name.

11. Anti-affinity scheduling can be used to claim disproportionate resources

Severity: Low
Type: Denial of Service
Difficulty: High
Finding ID: TOB-K8S-TM11

Description
Kubernetes allows users to specify various mechanisms for Pods. This can be as simple as assigning Pods to a specific node or a complex dance of determining various aspects of nodes and Pods. Users may also specify which Pods cannot be scheduled together, allowing a Malicious Internal User to specify that no other Pods be scheduled on the same host, effectively commoditizing a node to the attacker's workload alone.

Justification
The difficulty is high for the following reasons:

- An attacker must have sufficient privileges to schedule Pods.
- An attacker must know or guess other Pod names against which to claim anti-affinity.

The severity is low for the following reasons:

- Attackers with this level of access could also simply schedule a large number of Pods.
- Attacks that consume entire hosts are noisy and will eventually be investigated.
- Denial-of-Service attacks that consume nodes will not impact currently running Pods, but rather impact the scheduling of future Pods.

Recommendation
Short term, document that features such as anti-affinity may be used in ways that cause host unavailability across the cluster.

Long term, add tooling and processes to aid administrators in reviewing the state of clusters. This will support administrators as well as incident responders to discover and respond to resource-exhaustion events such as this. Furthermore, consider preventing users from selecting which scheduler they can use. Reserve that as an administrative function. This will allow administrators to handle scheduling in a safe way, and prevent attackers from specifying which scheduler should be used.

12. No back-off process for scheduling

Severity: Informational | Difficulty: Undetermined
Type: Denial of Service | Finding ID: TOB-K8S-TM12

Description
Scheduling a Pod within Kubernetes is an intricate dance of state coördination across multiple components; kube-scheduler may interact with kubelet, the Replica Set Controller, and other processes within the system. However, there is no back-off process when kube-scheduler determines that a kubelet is the appropriate host for a Pod, but the kubelet itself rejects scheduling the Pod. This may create a tight-loop wherein kube-scheduler, the Replica Set Controller, and kubelet cause a contention wherein kube-scheduler continuously schedules a Pod that kubelet rejects.

Justification
This item is of Informational severity and as such represents a noteworthy comment within the system, rather than an actual vulnerability.

Recommendation
Short term, document that this issue exists, and note that developers may be able to accidentally or maliciously introduce a tight-loop within a cluster via this feedback failure loop.

Long term, implement a back-off process for kube-scheduler and support graceful node failure. This may go so far as to include a “decay list” of nodes which are continuously failing to schedule Pods. Such a list can be used for further scheduling decisions within kube-scheduler.

KCM and CCM findings

kube-controller-manager (KCM) and cloud-controller-manager (CCM) are two core components of Kubernetes' interaction with the underlying platform. Like other controllers, such as the Replica Set Controller, KCM and CCM attempt to move the state of the cluster towards the desired state. CCM itself is a reference implementation, meant to separate out cloud-specific controller code from other controller code. In this way, it will allow administrators running on one cloud provider to exclude code meant for another cloud provider.

13. Separate out controllers based on principle of least authority

Severity: Low
Type: Access Control
Difficulty: High
Finding ID: TOB-K8S-TM13

Description
KCM and CCM run several different controllers in a “control loop,” or an infinite loop of feedback. KCM and CCM are packaged as single binaries, with multiple controllers packaged as Go-level modules within the source code used to build the binary. These controllers impact a wide range of items across the cluster, but are generally low-privileged and unable to impact much outside of the narrow slices of policy for which they are defined. However, some of these controllers are highly privileged (such as the Service Account Controller) and can access their own permissions. If an attacker or malicious controller were able to call these functions, they could escalate privileges across the cluster, potentially to administrative-level access.

Justification
The difficulty is high for the following reasons:

- An attacker must know or discover a vulnerability allowing them to call privileged functions.
- They must have position sufficient to use the escalated privileges, such as a Malicious Internal User.

The severity is low for the following reasons:

- Attackers with this level of access could likely impact other items with a lower Difficulty threshold.
- Attackers could escalate privileges across the cluster, or subtly modify resources on the fly.

Recommendation
Short term, plan ways that privileged controllers may be separated from unprivileged ones, and test if this is feasible within the context of both KCM and CCM.

Long term, separate out privileged controllers into their own binary or binaries. Controller managers should not mix levels of privilege, as attackers or even just simple coding mistakes can lead to privilege escalation.

kubelet findings

kubelet is the central orchestrator for Pods within the Kubernetes system. It runs Pods by watching for podspecs that have been allocated to its host (by kube-scheduler), and passes the podspec to the Container Runtime for execution. Aside from this, kubelet also handles reporting the health status of Pods and containers to kube-apiserver, monitoring Pods themselves for failure, working with the Container Runtime to deschedule Pods when so requested, and reporting host status to kube-apiserver (for use by kube-scheduler). Like kube-proxy, kubelet runs on the individual hosts, but with a different trust boundary than Pods themselves, as it is central to the correct operation of the cluster as a whole.

14. kubelet hosts unauthenticated ports that leak pod spec information

Severity: Medium | Difficulty: Medium
Type: Information Disclosure | Finding ID: TOB-K8S-TM14

Description
kubelet, like most components within the Kubernetes system, uses HTTP ports for various tasks such as reporting or task execution. Specific to kubelet, there are three main ports:

- 10250, an authenticated HTTPS server, with authentication provided by delegated authentication from the kube-apiserver, used for task execution and kubelet update.
- 10255, an unauthenticated HTTP server used for status information and health information, but also includes Pod spec information.
- 10248, an unauthenticated HTTP server used for health information.

Justification
The difficulty is medium for the following reasons:

- An attacker must have sufficient position to affect the attack, such as an Internal Attacker or Malicious Internal User.
- Minimal tooling is needed to issue an HTTP request.
- An attacker must know, or guess, the location of kubelet host and ports.

The severity is medium for the following reasons:

- Pod specs do not by default contain secrets, other than potentially ConfigMaps and Repository authentication credentials.
- An attacker armed with this information may gain a better understanding of the layout of a cluster's workload, but minimal other information about the inner workings of the cluster.

Recommendation
Short term, document the leakage of Pod spec information, and plan ways to remove it. Per the RRA discussions, the kubelet team is already planning on removing port 10255, which is mainly in place for cAdvisor, which is deprecated. In more recent versions of Kubernetes than this work focused on, port 10255 is configured off by default, but can be activated either by installers/distributions or cluster administrators.

Long term, remove the deprecated ports, and minimize the attack surface available to an Internal Attacker. This should also include changing port 10250 to a fully bootstrapped TLS certificate by default. In this way, kubelet will present as strong a face as possible to internal attackers.

15. Bootstrap certificate is long-lived and not removed by default

Severity: Low
Type: Configuration
Difficulty: High
Finding ID: TOB-K8S-TM15

Description
Kubernetes can bootstrap certain components, such as kubelet, from certificates by default. These certificates provide a mechanism for components to request enough access of kube-apiserver so as to generate a Certificate Signing Request (CSR) and produce a signed certificate that the component may use for at least client authentication. However, the certificate is long-lived, without a Time to Live (TTL), and is not removed by default.

Justification
The difficulty is high for the following reasons:

- An attacker must transit several trust boundaries, and have host-level access to a Worker node.
- The attacker must then have the ability either to bring up other hosts within the cluster or create their own kubelet under their control.

The severity is low for the following reasons:

- In and of itself, a long-lived bootstrap certificate does not provide an attacker with sufficient direct access.
- An attacker can make CSR requests to the kube-apiserver, which may provide an attacker with access to other credentials within the cluster.

Recommendation
Short term, document that the certificate is long-lived, and must be removed by manual processes.

Long term, issue bootstrapping certificates with an explicit-but-reasonable TTL, such as one week. This should provide administrators plenty of time to bootstrap a cluster, but remove the risk of a stolen bootstrapping certificate from further impacting the cluster. Additionally, if certificate revocation is added to the cluster, bootstrap certificates may be revoked once the CSR has been received.

References

- TLS Bootstrapping

kube-proxy findings

kube-proxy, much like kubelet, is a transitive component within the cluster: it straddles the edge between two trust boundaries, namely the Worker and Container zones. kube-proxy itself works by watching for service, endpoint, and similar network configurations on kube-apiserver, and then implementing the networking request, in conjunction with the Container Network Interface (CNI) in one of several modes:

- As a literal network proxy, handling networking between nodes
- As a bridge between Container Network Interface (CNI), which handles the actual networking, and the host operating system
- `iptables` mode
- `ipvsadm` mode
- two Microsoft Windows-specific modes (not covered by the RRA)

kube-proxy itself is actually a cluster of five programs, which work to create a consistent networking experience across Pods and services. In this way, kube-proxy manages the raw plumbing of networking, connecting the CNI's transport layer to Linux's routing layer (via third-party tools such as `iptables`).

Userspace proxy

The original mode of operation for kube-proxy, wherein kube-proxy received and forwarded packets for Kubernetes' hosted services. While this mode is not often used anymore, due to performance, the setup is the same for most other modes of kube-proxy. Furthermore, it is a core mode that may be useful under certain circumstances.

Setup:

1. Connect to the kube-apiserver.
2. Watch the kube-apiserver for services/endpoints/&c definitions.
3. Build an in-memory caching map: for services, for every port a service maps, open a port, write iptables rule for Virtual IP (VIP) & Virtual Port.
4. Continue with step No. 2, until the cluster is restarted or terminated.

When a consumer connects to the port:

1. The desired service is running on a VIP:VPort pair.
2. The Root NS lookup, which is routed by an iptables defintion, which eventually points to the kube-proxy port.
3. When a connection is received, look at the src/dst port, check the map, pick a service on that port at random (if that fails, try another until either success or a retry count has exceeded).

4. Shuffle bytes back and forth between backend service and client until termination or failure.

iptables

iptables is a common mode of operation for kube-proxy; it interacts directly with iptables in order to build routing rules for VIP:VPort pairs. However, this mode does not require kube-proxy to actually intercept or communicate with client connects. Instead, kube-proxy uses iptables to create rewriting rules for the intended host, and has no futher interaction with the system, until such time that iptables restore command sets must be updated.

1. Same initial setup as the userspace proxy, sans opening a port directly.
2. Build an iptables restore command set, which is simple a giant string of services.
3. Map user VIP to a random backend, rewriting packets at the kernel level, so kube-proxy never sees the data.
4. At the end of the sync loop, write batches to avoid iptables contentions/
5. Perform no more routing table updates until service updates, from watching kube-apiserver or a time out.

NOTE: rate limited (bounded frequency) of iptables updates:

- No later than 10 minutes by default
- No sooner than 15s by default, if there are no service map updates

ipvs

1. Similar setup to iptables & userspace proxy modes.
2. Here, we use the `ipvsadm` and `ipset` commands instead of iptables
3. This does have some potentially unintended consequences:
    - IP address needs a dummy adapter
    - **NOTE** Any service bound to 0.0.0.0 is also bound to **all** adapters
    - This is somewhat expected because of the binding to 0.0.0.0, but can still lead to interesting behavior

Networking Concerns

Low-level network attacks may still impact kube-proxy, such as ARP Poisoning. Furthermore, endpoint selection is namespace **and** Pod-based, so an injection could theoretically overwrite this mapping. Additionally, further work may be needed to use only `CAP_NET_BIND`, which allows a process or containerbind to low ports, without root permissions, for containers/pods, to alleviate concerns surrounding attacks such as ARP Poisoning via `CAP_NET_RAW`.

16. Race condition in Pod IP reuse

Severity: Low
Type: Timing
Difficulty: High
Finding ID: TOB-K8S-TM16

Description
kube-proxy coördinates with Pods, kubelet, and other components to “string the wire,” so to speak, of communications within a cluster. This includes Pod IPs, which generally have a larger allocation than there are Pods within a cluster by a factor of two. However, if an attacker were able to cause a churn in Pod IPs, they could potentially win a race condition, and trick kube-proxy into forwarding traffic to a Pod controlled by the attacker, rather than the Pod expected by the cluster.

Justification
The difficulty is high for the following reasons:

- An attacker must have sufficient position to cause a large volume of turnover in Pod IPs.
- The attacker must also have sufficient privileges to launch malicious Pods or have previously compromised a Pod with the Pod IP they wish to control.

The severity is low for the following reasons:

- Attackers with position sufficient to cause Pod IP reuse could likely use other attacks, such as ARP Poisoning, to achieve a similar effect with less work.
- The attack itself is largely theoretical, concerning a possible method by which an attacker could win a race condition against the Pod IP assignment algorithm.

Recommendation
Short term, document the issue, so that users may be aware of a possible race condition.

Long term, determine a method for a back-off process within kube-proxy, and ways of ensuring that tight loops cannot allow attackers to win race conditions. It is possible that the best arbiter of routing truth may be kube-apiserver, however, this would require larger architectural changes to the system as a whole. An achievable goal would be to simply back off assignments when tight-loop Pod IP churn is noticed, and allow the normal network process to reach equilibrium prior to further assignments.

Container Runtime findings

The last-but-not-least component that the team reviewed was the Container Runtime. Container Runtime is technically an interface, like Container Networking, meant to support multiple Linux container runtime systems (e.g. Docker) with a single API. Container Runtime itself does not execute a container until instructed to do so by kubelet, as shown in the process below:

1. Container Runtimes expose an IPC endpoint such as a Unix Domain Socket
2. kubelet retrieves Pods to be executed from the kube-apiserver
3. kubelet issues a request to the Container Runtime web server
4. The web server returns a URL with a single-time-use token
5. kubelet issues a request to the URL via gRPC over Unix Domain Socket
6. The Container Runtime Interface then executes the necessary commands/requests from the actual container system (e.g. Docker) to run the Pod

17. Search space for single-use token is too small

Severity: Low
Type: Cryptography
Difficulty: High
Finding ID: TOB-K8S-TM17

Description
Container Runtime coördinates with kubelet via two mechanisms: a TCP/IP web server, and a gRPC server running via Unix Domain Sockets. The gRPC request is authenticated via a single-use token issued to kubelet in response to a scheduling request. However, the token is small, being only eight characters long, meaning an attacker could feasibly generate many valid tokens in a short amount of time.

Justification
The difficulty is high for the following reasons:

- An attacker must have sufficient position to access the Unix Domain Socket.
- They must then generate many tokens (potentially up to 2^64).
- These tokens must be discovered before a one-minute timeout has elapsed.

The severity is low for the following reasons:

- An attacker with Host access could impact far more sensitive items than attempting to brute force a scheduling token.
- The attack would merely stop a Pod from being scheduled, which would either result in it being rescheduled or in another host scheduling the pod, minimizing total impact.

Recommendation
Utilize a standard, cryptographically secure token such as a UUIDv4. This will ensure that the search space is too large for practical searches, and utilize standard, well understood token-generation practices.

A. RRA Template

Overview

- Component:
- Owner(s):
- SIG/WG(s) at meeting:
- Service Data Classification:
- Highest Risk Impact:

Service Notes

The portion should walk through the component and discuss connections, their relevant controls, and generally lay out how the component serves its relevant function. For example a component that accepts an HTTP connection may have relevant questions about channel security (TLS and Cryptography), authentication, authorization, non-repudiation/auditing, and logging. The questions aren't the only drivers as to what may be discussed -- the questions are meant to drive what we discuss and keep things on task for the duration of a meeting/call.

How does the service work?

Are there any subcomponents or shared boundaries?

What communications protocols does it use?

Where does it store data?

What is the most sensitive data it stores?

How is that data stored?

Data Dictionary

Name	Classification/Sensitivity	Comments
Data	Goes	Here

Control Families

We're interested in these areas of controls based on the audit working group's choices.

When we say "controls," we mean a logical section of an application or system that handles a security requirement. Per CNSSI:

> "The management, operational, and technical controls (i.e., safeguards or countermeasures) prescribed for an information system to protect the confidentiality, integrity, and availability of the system and its information."

For example, a system may have authorization requirements that say:

- users must be registered with a central authority.
- all requests must be verified to be owned by the requesting user, and
- each account must have attributes associated with it to uniquely identify the user and so on.

For this assessment, we're looking at six basic control families:

- Networking
- Cryptography
- Secrets Management
- Authentication
- Authorization (Access Control)
- Multi-tenancy Isolation

Obviously we can skip control families as "not applicable" in the event that the component does not require it. For example, something with the sole purpose of interacting with the local file system may have no meaningful Networking component; this isn't a weakness, it's simply "not applicable."

For each control family we want to ask:

- What does the component do for this control?
- What sorts of data passes through that control?
    - for example, a component may have sensitive data (Secrets Management), but that data never leaves the component's storage via Networking
- What can an attacker do with access to this component?
- What's the simplest attack against it?
- Are there mitigations that we recommend (i.e. "Always use an interstitial firewall")?
- What happens if the component stops working (via DoS or other means)?

- Have there been similar vulnerabilities in the past? What were the mitigations?

Threat Scenarios

- An External Attacker without access to the client application
- An External Attacker with valid access to the client application
- An Internal Attacker with access to cluster
- A Malicious Internal User

Networking

Cryptography

Secrets Management

Authentication

Authorization

Multi-tenancy Isolation

Summary

Recommendations