Kubernetes

Security Whitepaper

June 17, 2019

Prepared For:
Kubernetes Security WG | *Kubernetes*

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

Dominik Czarnota | *Trail of Bits*
dominik.czarnota@trailofbits.com

Robert Tonic | *Trail of Bits*
robert.tonic@trailofbits.com

Ben Perez | *Trail of Bits*
benjamin.perez@trailofbits.com

Introduction

This document is one of the artifacts produced following Trail of Bits' March 11th to May 10th, 2019 assessment of the security of the open source Kubernetes system. It provides a reference on different aspects of securing Kubernetes, based on the audit team's observations. The white paper defines the key aspects of the Kubernetes attack surface and security architecture.

The assessment yielded a significant amount of knowledge pertaining to the operation and internals of a Kubernetes cluster. This document presents that knowledge in a format useful to the community. We address key aspects of the Kubernetes attack surface and security architecture, enabling administrators, operators, and developers to make sound design and implementation decisions.

In order to provide background for best practice recommendations and guidelines, we describe the components of a cluster, how these components communicate, their internal abstractions, and, at a high level, how these components are hosted on the underlying infrastructure. We also detail the use of cryptography, and the potential threats a Kubernetes cluster could face. Next, we propose and discuss a set of best practices and guideline recommendations.

Throughout many of the topics discussed, multi-tenancy is relevant. To help provide context to the affected tenants, we use these terms:

- Cluster administrators are tenants with access to the underlying hosts, and who have elevated permissions on a cluster.
- Cluster operators are tenants who have limited administrative access to a cluster, to perform operations such as the creation, deletion, and management of cluster workloads.
- Workloads are applications, tasks, or jobs which can be executed and managed by Kubernetes.

As a whole, this document is a summation of thoughts from the assessment team, covering security-adjacent issues uncovered throughout our assessment of Kubernetes. This content presents both general and specific recommendations from a team intimately aware of Kubernetes' internals and wider cloud- and distributed-systems configurations. Additionally, guidance is included to promote further assessments and discussion of Kubernetes from the varied perspectives of administrators, security researchers, and developers.

Kubernetes overview

Components

A Kubernetes cluster requires several base components to operate, specifically: the kubelet, kube-apiserver, kube-scheduler, kube-controller-manager, and a kube-apiserver storage backend. Other components such as controllers and schedulers provide features related to networking, scheduling, or environment management. While these features may be required for certain workloads, they complement and extend the functionality of the base components.

Communications and protocols

The base components of Kubernetes use HTTP API endpoints for state-related communications. The core component of these state communications is the kube-apiserver. The kube-apiserver is the middle-man between the storage backend and the cluster components, allowing reads and modification of cluster state. Other components, such as the kubelet, use the kube-apiserver's API to perform tasks such as retrieving information about which Pods their node should be currently running, or which service ports to configure on a node's host.

Similar to the kube-apiserver, the kubelet also interacts with external services. In order for Pods to execute, the kubelet must interact with a container runtime through a specified container runtime interface (CRI). Supported CRIs can be communicated with through a variety of protocols. Local to the node, standard TCP and sock communications can be used. Remotely, TCP communications are typically used.

Beyond the scheduling and execution of Pods, most cluster workloads require external interaction with Pods. The kube-proxy, CNI, and kubelet are used to route ingress and egress TCP, UDP, and SCTP traffic from the host node to Pod based on service states returned by the kube-apiserver. Without this, Pods could be limited to egress communication abilities, using the underlying node's host networking configuration. To allow ingress traffic, kube-proxy typically uses iptables[1] (or similar tools such as ipvs-adm) to configure the node's host to forward traffic from an external source to an internal Pod.

As a whole, Kubernetes attempts to orchestrate and abstract existing systems to allow for large-scale container deployments through an easy-to-use declarative interface. Because of this, there is significant interaction between Kubernetes components and external systems.

[1] Kube-proxy configuration of iptables, https://kubernetes.io/docs/tasks/debug-application-cluster/debug-service/#is-kube-proxy-writing-iptables-rules

Without this interaction, Kubernetes would not be able to achieve expected functionality for supported workloads.

Abstractions and objects

Once a base Kubernetes cluster has been provisioned and configured, Kubernetes clusters are controlled via operator-defined objects. These objects are abstractions for cluster operations such as service discovery, replica management, port configuration, and the like[2]. The kube-apiserver derives how the state of the cluster should be mutated to reflect these objects. The other components of Kubernetes query the kube-apiserver for the state to maintain and adhere to.

Through the use of the operator-defined objects, Kubernetes alleviates traditionally complex tasks related to system administration and task management. Component configuration is mostly agnostic of the workloads running atop the cluster, preventing reconfiguration of the underlying Kubernetes components and promoting configuration portability. However, component configuration does impact the effects of certain object definitions.

To provide ease of configuration and portability, many of these abstractions have the requirement of being state- and component-agnostic, adding operational complexity of a different type. Many objects within Kubernetes are composed with other objects. This type of compositional approach allows for the creation of complex configurations, where considerations must account for both cluster-component capabilities, and object presence. Detailed discussions have been included in Infrastructure and cluster composition: Infrastructure management and Infrastructure and cluster composition: Object composition.

[2] Understanding Kubernetes Objects, https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

Infrastructure and cluster composition

Infrastructure management

While Kubernetes facilitates high-availability workload deployments, the underlying hosts, components, and environment of a Kubernetes cluster must be configured and managed. This management has a direct impact on the capabilities of the cluster, and affects the behavior of an operator's composed objects. With this in mind, the options available for configuring components of Kubernetes often fluctuate significantly in supported versions, and vary in their approach to default settings. This leads to a non-trivial amount of configuration required by an administrator to stand-up a functional cluster for a given workload. More effort must then be spent maintaining the cluster to abide by these settings, especially when planning and executing upgrades of Kubernetes components.

The impact of Kubernetes on the underlying node hosts must also be kept in mind. Kubernetes workloads may have operations or dependencies which use host resources, such as network interfaces, volumes, and applications. These types of operations impact the underlying node host in regards to file permissions, volume access, and resource consumption. For example, if an operator is able to schedule privileged Pods, they may have access to administrator resources in the underlying node hosts across the cluster, depending on the access controls in place on each node host. Furthermore, this can have an inverse effect on the security of the cluster, since a privileged process on a node host could gain access to sensitive cluster configuration information such as secrets and certificates.

Configuration of Kubernetes components can also be made more complex due to Kubernetes' ability to partially manage itself. kubelet is able to run Pods specified in static manifest files[3] within a manifest directory, as well as from a specified remote kube-apiserver, at the same time. This allows cluster administrators to run certain cluster components in a kubelet-managed way. As an example, the kube-apiserver can be defined as a Pod in a static manifest, and the managing kubelet can be configured to point to the Pod (via `--pod-manifest-path` flag). Once the kube-apiserver Pod is started from the static manifest and reachable by the kubelet, the kubelet will begin pulling Pod specifications from the kube-apiserver to execute. This type of configuration fundamentally changes the security considerations that must be made when deploying a Kubernetes cluster, since the execution environment of the cluster components changes from running directly on the host to running within a container runtime managed by the kubelet.

[3] Kubernetes Static Pods, https://kubernetes.io/docs/tasks/administer-cluster/static-pod/

To help manage the complexity of configuration and management, various projects have been formed to help configure and manage the underlying hosts and environments of a Kubernetes cluster. Projects such as Kubespray[4], Kops[5], Rancher[6], and many others aim to provide clusters which an administrator can further configure and better maintain. Despite many of these projects aiming to provide "production-ready" clusters, even these have differences in configuration, management, and operation. Furthermore, if extra functionality is required of the cluster, complexity of management increases.

Object composition

In order to use Kubernetes, operators must define objects for the cluster to derive state from. Nearly every aspect of the Kubernetes cluster is controllable through these objects, such as access controls (role based access controls, Pod security policies, etc), deployments, Pods, and volumes.

When defining objects, operators may compose complex objects through references to other objects. At the time of object creation, another referenced object does not need to exist. This allows for trivial state-agnostic configuration, since once a referenced object exists, the cluster state will update and reflect its existence.

While composing objects in a state-agnostic way makes complex-object composition easier—since it is no longer order-dependent—it can be tough for an operator to detect misconfigurations. Because some objects can be successfully created without the presence of an object that may be depended on, expected functionality must be tested to ensure the state properly reflects the operator's intent and configuration. This is especially concerning when considering the impact a misconfiguration could have on a critical control such as Role Based Access Controls (RBAC)[7]. To use RBAC, policies are defined through roles and role bindings. Because roles and role bindings can be created and maintained separately, this could cause misconfigurations by an administrator who assumed a role or binding was created and applied successfully, but in fact was not applied as expected.

Furthermore, the objects an operator can define on the cluster can be impacted by the underlying configuration of the cluster. Similar to how an object will be created even if a reference does not exist, Kubernetes allows objects to be created even if the component which would use its configuration does not exist or is disabled. For example, an operator can define a PodSecurityPolicy (PSP) even if the PSP admission controller is not enabled on

[4] Kubespray, https://github.com/kubernetes-sigs/kubespray
[5] Kops, https://github.com/kubernetes/kops
[6] Rancher, https://rancher.com/
[7] Role Based Access Control Overview for Kubernetes, https://kubernetes.io/docs/reference/access-authn-authz/rbac/#api-overview

the kube-apiserver[8], resulting in a policy which will not be enforced, even if it is bound to a role.

Health detection and failures

One of Kubernetes' many goals is to facilitate high-availability, fault-tolerant service management. To achieve this, health detection and fault tolerance is required on multiple layers of the Kubernetes stack.

On the lowest layer, base Kubernetes components contain health checks to ensure they are responsive and healthy, managing unhealthy components through eviction periods and timeouts. This allows the cluster components to regulate workloads based on the status of underlying components.

On the workload layer, Kubernetes supports the ability to define both liveness and readiness checks[9]. The readiness checks allow the kubelet to determine when a Pod is ready to perform work. The liveness checks allow the kubelet to determine if a Pod is continuously suitable for work. Both of these checks support three probing methods of checking a Pod's status: TCP, HTTP, and Executor. When performing the TCP and HTTP(S) status checks, the kubelet will attempt to contact a specified host and port based on the Pod's specification. If a connection is successfully established (TCP) or an acceptable response status (HTTP(S)) has been returned, the check will succeed. In the case of the Executor, the kubelet will attempt to execute a command within the specified Pod. If the command returns a successful status code, the check will succeed.

Regardless of layer, Kubernetes will attempt to mitigate the impact of a workload or node which fails its status checks[10]. For both, eviction and timeout policies are typically used. If a kubelet node is no longer passing its status checks, eviction policies may take effect, evicting a node from the cluster and triggering Pods to be rescheduled away from the evicted node. If a Pod is no longer passing its status checks, it may be restarted on the same node, or rescheduled to another node in an attempt to get the status check to succeed.

When configuring and securing a cluster and its workloads, all aspects of this functionality must be considered. If an attacker gains access to a container, the attacker must do so in a way that does not disrupt a Pod's health checks. If an attacker causes a Pod to fail health checks, this could lead to the container being terminated and started elsewhere. Furthermore, outside constraints such as cgroups could lead to a Pod container's

[8] The PodSecurityPolicy admission controller, https://kubernetes.io/docs/concepts/policy/pod-security-policy/#enabling-pod-security-policies
[9] Liveness and Readiness checks in Kubernetes, https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/
[10] Node conditions, https://kubernetes.io/docs/concepts/architecture/nodes/#condition

termination if resource allocations are exceeded, limiting an attacker’s ability to use available resources.

Beyond avoiding health check disruption, an attacker with the ability to schedule Pods could also use liveness and readiness checks to find out information about a node’s host environment. Using the fact that the kubelet performs probes from the host, an arbitrary host and port can be specified for a TCP and HTTP(S) status probe, allowing an attacker to enumerate host ports on the networks available to the underlying node host’s interfaces.

Host and cluster multi-tenancy

Because Kubernetes can be used for such a wide variety of workloads, it is important to consider the impact and potential effects of multi-tenant cluster operators, cluster workloads, and cluster node hosts.

Multi-tenant node hosts present a unique challenge to the security of a Kubernetes cluster. Underlying host resources can be consumed by other users of the system, resulting in reduced workload capacity. For example, resources such as CPU, memory, ports, and disk space are directly influenced by a multi-tenant node host. Depending on the size of the cluster and the type of workload, sudden node-host resource-availability changes could result in workload-availability issues. From an attacker's perspective, even if direct access to the cluster is not possible, resource exhaustion attacks on multi-tenant hosts still present a potential method of interfering with cluster operations. While this is not the fault of Kubernetes, it requires consideration when designing where node clusters should run.

Within a cluster, there are several tenancy considerations that must be made when building and managing a cluster. Numerous operators of a cluster can exist at the same time, with varying levels of access. Furthermore, multiple workloads can co-exist on a cluster, both in and outside of the same namespace. Between operator multi-tenancy and workload multi-tenancy, there is a significant attack surface for an attacker with access to either a workload or operator account.

Kubernetes namespaces were developed as a method to help provide workload isolation[11]. Running multiple, potentially multi-tenant, workloads in the same namespace sidesteps the protections of namespaces, resulting in a single large and flat namespace. In an environment with two workloads in the same namespace, an attacker with access to one workload could possibly access the other workload without encountering cluster-namespace restrictions. Depending on workload configuration, an attacker could use this lateral access to move and escalate to a more privileged workload.

Compounding namespace-workload tenancy concerns, operator tenancies and privileges directly affect the security of a cluster. If an attacker gains access to a particular tenant's credentials, the attacker may be able to escalate in and across namespaces and workloads through configuration of cluster objects.

[11] When to use multiple namespaces, https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/#when-to-use-multiple-namespaces

Kubernetes and cryptography

Several core components of the Kubernetes framework require the use of cryptography. In particular users need four features:

1. Encryption of data in etcd
2. Verifiable and tamper-proof encrypted data storage in etcd
3. Encrypted data transports
4. Verifiable component identities

In this section, we discuss how Kubernetes solves these problems and the choices administrators have during setup.

Encryption in etcd

When administrators store information in etcd, they need to be sure that their data is both private and tamper-resistant. Information privacy can be achieved through standard block cipher modes such as AES-CBC. However, privacy and tamper resistance can be accomplished simultaneously by using an authenticated encryption scheme[12]. Kubernetes supports four options when encrypting data, of which only the latter three provide authenticated encryption:

1. AES-CBC with PKCS#7 padding
2. AES-GCM
3. Secretbox
4. KMS

In the documentation, administrators are told that KMS and AES-CBC provide the highest level of security. While Trail of Bits believes that KMS is the best choice in general, by no means is AES-CBC more secure or more performant than Secretbox. In particular, AES-CBC does not provide authenticated encryption and is known to be vulnerable to padding oracle attacks[13]. While it is unlikely an attacker would be able to mount such a padding oracle attack on the data storage component of Kubernetes, future changes to the system may introduce such a vulnerability.

Despite authenticating encrypted data, AES-GCM is an extremely error-prone mode of encryption. It requires administrators to supply a random nonce which, if repeated, allows an adversary to decrypt messages and recover parts of the user's key. To avoid reusing

[12] How to choose an Authenticated Encryption mode, https://blog.cryptographyengineering.com/2012/05/19/how-to-choose-authenticated-encryption/
[13] Padding oracle attack on CBC encryption, https://en.wikipedia.org/wiki/Padding_oracle_attack#Padding_oracle_attack_on_CBC_encryption

nonces, administrators must frequently rotate keys. Since this process requires great care and diligence, it is a major weakness in AES-GCM. Clusters which use AES-GCM should instead use KMS, which uses AES-GCM as its underlying encryption algorithm but automates key rotation. Furthermore, the user-friendly API and documentation of KMS make it substantially less error prone than manually setting up encryption in Kubernetes itself.

Kubernetes should depreciate AES-GCM and AES-CBC. Administrators should be instructed to use KMS or encrypt all sensitive data with Secretbox by default.

Certificates

In order for components within a Kubernetes cluster to prove they are who they say they are, the kube-apiserver issues each component a cryptographic certificate which proves the component's identity. When two components need to communicate, they verify each other's certificates before sending any sensitive information. If a node is taken out of a cluster or is corrupted, administrators need a way to revoke that node's certificate. Currently there is no way to accomplish this in Kubernetes[14].

Perhaps the simplest solution to this problem is to have nodes maintain a certificate revocation list (CRL). This requires all components of the cluster to periodically check with the kube-apiserver to ensure their CRL is up-to-date. For small clusters this may be acceptable, but the bandwidth requirements become prohibitive for larger systems. We do note, however, that this scheme has been implemented in etcd[15].

OCSP stapling would be a more sensible alternative[16]. In this scheme, the kube-apiserver manages a revocation list. When a node needs to verify that a certificate is valid, it sends the certificate to the kube-apiserver which subsequently sends a signed response indicating the status of the certificate. This scheme reduces bandwidth and prevents adversaries from taking advantage of gaps between CRL updates.

HTTPS Connections

The Kubernetes system allows administrators to set up a public key infrastructure (PKI), but often fails to authenticate TLS connections between components, negating any benefit to using a PKI. This failure to authenticate components within the system is extremely dangerous and should be changed to use authenticated HTTPS by default. Lack of authentication opens up the possibility for malicious entities to trick the cluster into believing they have privileges they do not. Until this is fixed, administrators need to be

[14] Support for managing revoked certs, https://github.com/kubernetes/kubernetes/issues/18982
[15] Support for managing revoked certs, https://github.com/etcd-io/etcd/issues/4034
[16] Online Certificate Status Protocol, https://en.wikipedia.org/wiki/Online_Certificate_Status_Protocol

aware that much of their inter-node communications are not authenticated, and they must manually enable HTTPS in kubelet[17].

[17] TLS bootstrapping, https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-tls-bootstrapping/

Positioning Threat Actors within a Kubernetes cluster

Understanding threat actors and their associated positions is critical to understanding the risk of any finding: *who* can do *what* to *whom* from *where*. Kubernetes is a large system, with many attack surfaces exposed to users only at specific privilege levels. This section of the white paper focuses on what is exposed to attackers at each level. Our assessment focused on three main classes of attackers:

- External attackers, who did not have access to the cluster
- Internal attackers, who had transited a trust boundary, such as access to a Pod
- Malicious Internal users, who abuse their privilege within the cluster

Each threat in the threat model or finding in the technical report was rated with these three attackers and their positions in mind.

Internal and External Network Access

Kubernetes' components are highly networked: components retrieve configuration from kube-apiservers via HTTP. kube-apiserver itself retrieves status from components via HTTP servers which they themselves expose. While the kube-apiserver is protected with strong authentication and authorization, many other components in the cluster are not. Attackers with access to the network in an unfiltered capacity may have access to a wide range of information, including Pod specifications (from kubelet), namespace names, and secrets' names.

Therefore, Kubernetes must be protected by ancillary network controls. This should ensure that cluster components are separated from the wider internet and even from themselves. For example, Pods should rarely if ever need access to the kube-apiserver or etcd, and thus may be segmented away from those internal components. In contrast, kubelet and kube-proxy, which may live on the same physical host as a Pod, do need this access. The cluster as a whole should be protected at the perimeter from unauthenticated user access. Only direct, service-specific access should be granted to cluster-level hosted services and Pods.

Within the cluster, care should be paid to restricting egress from core components like kube-apiserver and from Pods to internal components, such as kubelet or kube-apiserver. Cutting off internal ingress and egress restrictions and external access to the cluster can frustrate most network-level attacks, or force an attacker to reach even deeper into a cluster, requiring the compromise of core hosts or other exposed components. Additionally, the cluster is not immune to attacks such as ARP poisoning, so normal server hardening should be taken into account when determining network and host configuration.

Host Access

Eventually, all items within a cluster, from Pods to Kubernetes' own components, must run on a specific host, which may have administrators who are not active in the cluster itself. For example, a host may have a systems-administration team that handles disk space and network access, but does not have access to the larger cluster. Kubernetes relies on functionality on hosts for low-level details; for example, kube-proxy and kubelet make heavy use of the Linux routing system to accomplish their tasks. Furthermore, several Kubernetes components rely on Unix Domain Sockets to protect access to sensitive functionality; attackers with access to hosts could access a wider array of cluster functionality.

Internal Attackers or Malicious Internal Users with access to a host could impact a number of cluster components. Kube-proxy makes heavy use of iptables or ipvs for inter-Pod routing and service connections. An attacker who could change routing tables could route traffic to other hosts and expose sensitive cluster data to malicious hosts. Additionally, host-level users with access to privileged accounts could theoretically impact functionality via Unix Domain Sockets which lack further authentication. Hosts backing cluster components should never be shared across any other functionality, and should for all intents and purposes be considered "closed servers," with minimal ability for host-level users to login. Furthermore, hosts should adopt all normal hardening measures, ensuring that Kernel Security Modules (KSMs) and heavy auditing are configured, so as to determine if and when a malicious user undertook a specific action.

Pod Access

Pods are an abstraction level of containers within a cluster, but do not provide strong isolation protections by themselves. As Pods run arbitrary user workloads, Internal Attackers may be able to get an initial foothold through a vulnerable client application. Likewise, because Pods are scheduled by users of the cluster, a Malicious Internal User may be able to abuse Kubernetes' functionality to consume extra resources or create various "noisy neighbor" problems (wherein a tenant or tenants of a cluster consume far more resources than other neighbors at a virtual machine or API level, which are difficult to detect at a higher level) within the cluster. Lastly, Kubernetes does not yet have a strong sense of multi-tenant isolation. Deciding on a single direction for multi-tenancy will help foment a set of rules and understandings that may be applied more fully to the threat profile of a given cluster.

Similarly to host access, Internal Attackers may parlay access to a Pod into wider cluster access. At the time of this report, Kubernetes mounted default credentials in every Pod; an Internal Attacker could use these credentials to access other resources within the cluster, such as the kublet. From there, the Internal Attacker may be able to move laterally throughout the cluster to wider access. For cluster administrators, care should be taken that vulnerable applications and Pods are patched as soon as possible, so that Internal Attackers may not gain an initial foothold within the cluster. Additionally, audit all component logs within the system for lateral movement; this should include regular checks of accesses of cluster components by Pods and other user workloads.

Finally, Malicious Internal Users may abuse functionality within a cluster to consume resources and cause a denial of service. For example, if a Malicious Internal User can schedule Pods, not only can they schedule whatever Pod they'd like, including ones with malicious software, they may simply use an Anti-affinity scheduler to claim whole host nodes to themselves. Monitoring and auditing the workload of a cluster is a must. Reviewing the allocation structure of every host is as important as ensuring the correct kube-scheduler configurations are available to minimize resource hogging.

Cluster Access

The cluster itself has functionality that may be manipulated by Malicious Internal Users. Kubernetes is architected around a "spoke and wheel" style architecture: components watch for their resource type from kube-apiserver, and in turn update related resources within the kube-apiserver, leading to further changes in other components. In terms of design, the kube-apiserver should be the central arbiter of truth and consistency within a cluster. However, several components within the system, such as kubelet, allow users with certain Role-Based Access Control (RBAC) roles to access functionality directly, bypassing the kube-apiserver.

While kube-apiserver contains logs that reconstruct user actions, individual components do not by default log with enough granularity to reconstruct a Malicious Internal User's actions. Restrict network access even for privileged users to the smallest amount of surface necessary. If possible, only allow a cluster's users access to the kube-apiserver itself, as this will reduce the number of components a Malicious Internal User may be able to communicate with, and will allow cluster administrators to recreate an attacker's path via kube-apiserver alone.

Recommendations for cluster administrators

Attribute Based Access Controls vs Role Based Access Controls

When comparing the permissions systems, Role-Based Access Controls (RBAC) are heavily recommended over the use of Attribute-Based Access Controls (ABAC). RBAC may be configured dynamically while a cluster is operational. In contrast, the static nature of ABAC can increase the difficulty of ensuring proper deployment and enforcement of controls.

Clusters should not use both ABAC and RBAC. This could grant users unintended permissions, since if one validation fails, but the other succeeds, the cluster operator will still be able to perform the action[18]. Administrators who are migrating off of ABAC should be extremely careful of this drawback, as it could allow operators to perform actions they are not allowed to perform during migration.

RBAC best practices

When interacting with RBAC, it is important to remember that the configuration of Kubernetes' components impacts the defined policies. As mentioned in the Kubernetes overview: Abstractions and objects section, objects can be composed by referencing objects that may not yet exist. Additionally, objects can be created even if the component using the object does not exist. This functionality can be very dangerous when constructing RBAC policies, since functionality must be tested to ensure the configuration works in the expected manner. This could lead an administrator to believe that policies are in effect, when in fact they are not. To avoid this type of mistake, administrators should test to ensure: that policies defined on the cluster are backed by an appropriate component configuration, that policies are properly tied to roles, and that the policies properly restrict behavior.

Node-host configurations and permissions

When configuring a host to run Kubernetes, file permissions should be as restrictive as possible, especially for the kubelet and control plane components. An attacker with access to the underlying host (either through a Pod, or direct access) can use certificates, tokens, and other sensitive information on disk to gain privileged access to the underlying host or Kubernetes cluster.

Exposed services on the underlying host should be restricted through network policy and authentication to prevent unauthorized access from a Pod scheduled on a node. Kubernetes itself is composed of various components which expose themselves as services

[18] RBAC Support in Kubernetes, https://kubernetes.io/blog/2017/04/rbac-support-in-kubernetes/

on the underlying host to facilitate bidirectional communications. Because of this, it is important to ensure that appropriate authentication and access controls are in place for the cluster nodes, since an attacker with network access to a single node could use Kubernetes components to compromise other nodes.

Default settings and backwards compatibility

Kubernetes contains many default settings which negatively impact the security posture of a cluster. These settings also have conflicting usage semantics, where some use either opt-in or opt-out specifications. The conflicting usage generally boils down to the preservation of backwards compatibility for both workload and component configurations. Ensuring appropriate configuration of all options requires significant attention by cluster administrators and operators.

Cluster operators and administrators must ensure component and workload settings can be rapidly changed and redeployed in the event of compromise or required update. Furthermore, post-deployment tests of both workloads and components should account for the presence of opt-in and opt-out settings to ensure implicit configuration has not occurred.

Networking

Due to the complexity of Kubernetes networking and the impact a container-networking interface has on a cluster's network requirements, it is difficult to provide specific recommendations suitable for all use cases. However, there are general guidelines that can be followed across these different configurations.

Proper segmentation and isolation rules of the underlying cluster hosts should be defined. Hosts tasked with executing control-plane components should be isolated to the greatest extent possible. Any interactions with control-plane components should be whitelisted explicitly. Next, hosts tasked with executing cluster workloads (kubelet nodes) should be as segmented as possible. While workloads may make host segmentation difficult due to service-discovery and -availability requirements, it is recommended to ensure host firewalls adequately restrict all network access regardless of the cluster workloads. Finally, ensure the container network interface is as restrictive as possible through the definition of cluster network policies.

Environment considerations

The environment a cluster is operated in affects the security considerations that must be made. Kubernetes contains many features which depend on cloud environments, such as load balancers and persistent volumes. These features directly impact resources which could be external to the hosted cluster within the operating environment. Conversely, the security of these resources directly impacts the cluster itself. If one of these environment resources (e.g. a persistent volume) is compromised and the cluster uses this resource, an attacker could use this functionality to pivot into the cluster.

As a whole, it is recommended that the security of a cluster's operating environment is addressed. If a cluster is hosted on a cloud provider, administrators should ensure that best-practice hardening rules are implemented. Furthermore, administrators should audit the access controls applied to cloud resources created and used by a cluster. Resources shared across an operating environment should be monitored closely.

Logging and alerting

Kubernetes can run in many environments, including those where underlying cluster hosts for worker nodes are ephemeral. Furthermore, a cluster can facilitate ephemeral workloads. Centralized logging of both workload and cluster host logs is recommended to enable debugging and event reconstruction.

The security of centralized logs and their corresponding alerts is extremely important. Depending on the logging levels used for components and workloads, it is possible for sensitive information to be disclosed through logs. Limit access to these logs, and make a best effort to filter sensitive information from logs.

Recommendations for developers

Avoid hardcoding paths to dependencies

Hardcoding paths to external resources or dependencies, especially for large, long-living projects like Kubernetes is problematic. A path change in a dependency can go unnoticed in the future, especially if the code using the path is rarely used or when there is a fallback mechanism.

An example can be seen in Kubernetes' kubelet process, where a dependency on hardcoded paths for PID files led to a race condition which could allow an attacker to escalate privileges. These hardcoded PID file paths were intended to be used to retrieve the PID of a running process. When they weren't found, it fell back to an insecure version of finding processes' PIDs by traversing the `/proc` directory and checking for a process name. This made it possible for an unprivileged user to spoof a process and potentially get more privileged access to devices (see findings TOB-K8S-21 and TOB-K8S-22 in the *Kubernetes Security Assessment*).

It is important to be conservative and cautious when handling external paths. Users should be warned if a path was not found, and have an option to specify it through a configuration variable.

Testing all hardcoded paths during end-to-end tests and carefully reasoning about cases where they might be different is recommended. Centralization of operator path configurations should be considered. This could reduce operator error, since path configuration would be less disparate.

File permissions checking

To configure components of Kubernetes, configuration files must often reside on disk. Furthermore, many of these configuration files contain sensitive information which, if an attacker is able to gain access to, could allow for privilege escalation on the host or cluster. Kubernetes currently does not enforce minimum file permissions to prevent this. It is recommended that Kubernetes support the ability to perform file permissions checking, and enable this feature by default. This will help prevent common file permissions misconfigurations and help promote more secure practices.

Monitoring processes on Linux

A Linux process can be uniquely identified in the user-space via a process identifier or PID. A given PID will point to a given process as long as the process is alive. If it dies, the PID can be reused by another spawned process. PIDs are usually assigned incrementally skipping already taken PIDs. When they reach a maximum number they overflow and start back from 0. The maximum PID number is defined in `/proc/sys/kernel/pid_max` file on Linux. It is usually set to 32768.

Process properties can be read and sometimes modified through the “proc” virtual filesystem which contains a directory for each PID. Those directories contain significant process metadata, such as links to the path name of the executed command, its current working directory, environment variables, opened files and sockets, security attributes, memory mappings and much more.

When using the “proc” virtual filesystem, it can be easy to make incorrect assumptions about expected behaviour and errors. For example if the process binary is located in a very long path, one can read the binary file via `/proc/<pid>/exe` but a readlink of it will result in an `ENAMETOOLONG` error. Furthermore, when the binary itself has been deleted, it is still possible to read its content via reading the file the link points to, but it’s readlink will return the old path concatenated with a “(deleted)” string.

It’s also important to note that PIDs are not process handles. They can’t be operated on directly like file descriptors. When reading multiple files from `/proc/<pid>/` it is important to first grab a `/proc/<pid>/` directory stream or file descriptor, and then access files through it. This prevents a race condition where a PID could be reused by another process between reading two files from `/proc/<pid>/` for a given PID. This is the result of an open directory stream pointing to the inode of the old process `/proc/<pid>` directory, while the new process `/proc/<pid>` will have a new inode.

Moving processes to a cgroup

A process and its threads can be moved to a v1 cgroup[19] on Linux systems by writing the process’s PID to `/sys/fs/cgroup/<controller>/<control group>/cgroup.procs`. Because this solution also uses PIDs, it is also vulnerable to race conditions. When moving a given process to less restricted cgroup it is necessary to validate that the process is the correct process after performing the movement.

Future cgroup considerations for Kubernetes

Both Kubernetes and the components it uses (runc, Docker) have no support for cgroups v2[20]. While this is currently not an issue as most current Linux systems come with support

[19] Control Group v1 documentation, https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
[20] Control Group v2 documentation, https://www.kernel.org/doc/Documentation/cgroup-v2.txt

for both cgroups v1 and v2 by default, it would be good to track this topic[21] as it might change in the future.

Future process handling considerations for Kubernetes

While the exact future of handling processes (or threads) on Linux is yet to be determined, tracking and participating in it's development is recommended. There are currently efforts to implement a "PIDFD," a race-free process descriptor[22] [23] [24]. The main goal is to send signals to processes in a race-free fashion.

Best practices for spawning processes

Many CLI programs support passing both optional and positional arguments. When passing them programmatically it is important to be cautious with spawning them via a shell, as this could let an attacker use the shell's glob patterns or exploit expansions. Go mitigates this by the fact that the `exec` package does not invoke shell directly to launch a process[25].

However, there is still room for programmer error by letting users pass an optional argument in the place of a positional argument. As an example, when we invoke `ls DIR` and the DIR is user controlled, the user can pass either a file, directory name, or an optional argument such as `--help`. The latter might have tremendous consequences in some programs. Examples of this abuse include in `tar`, `chown`, `chmod` or `rsync`[26].

In most situations these problems can be mitigated by passing "`--`" after specifying all desired optional arguments. The "- -" is an argument supported by most parsing libraries, informing them to stop parsing optional arguments. As a result, an invocation of "`ls -- --help`" will list a file or directory named "`--help`".

Although the "--" mitigation can work in many cases, there are programs that do manual parsing instead of using tested libraries that support "- -". Therefore, it is recommended to always check whether a given program supports "- -". If it doesn't, add additional validation to the positional parameters passed from user input.

[21] "cgroupv2: Linux's new unified control group system", https://chrisdown.name/2017/03/01/cgroupv2-linux-new-cgroup-hierarchy.html
[22] Towards race free process signaling, https://lwn.net/Articles/773459/
[23] Race-free pidfd access example https://github.com/torvalds/linux/commit/43c6afee48d4d866d5eb984d3a5dbbc7d9b4e7bf
[24] Pidfds: Process file descriptors on Linux https://kernel-recipes.org/en/2019/talks/pidfds-process-file-descriptors-on-linux/
[25] Golang docs: exec package overview, https://golang.org/pkg/os/exec/
[26] Back To The Future: Unix Wildcards Gone Wild, https://www.defensecode.com/public/DefenseCode_Unix_WildCards_Gone_Wild.txt