Securing Your Kubernetes Deployment

WRITTEN BY RORY MCCUNE, PRINCIPAL CONSULTANT AT NCC GROUP

CONTENTS

- INTRODUCTION
- THREAT MODEL
- SECURING EXPOSED NETWORK SERVICES
- HOST FILESYSTEM ACCESS
- ACCESS TO A SERVICE TOKEN
- NODE KERNEL ACCESS
- KUBERNETES USER SECURITY
- TOOLS AND REFERENCES
- CONCLUSION

INTRODUCTION

Kubernetes has exploded onto the technology scene over the last couple of years, with a large number of major cloud companies and others adopting it as the default way to orchestrate and scale container-based workloads. Of course, as with any new, rapidly developing, and popular technology, there are questions around how best to secure Kubernetes deployments and where teams should focus their efforts to reduce the risk of their clusters being the next target of an attack.

The Kubernetes ecosystem is both comprehensive and rapidly evolving. As such, it can be difficult to know which of the many possible areas should be made a priority. Also, as there are more than 60 different products and projects that offer Kubernetes deployment and installation, that makes it difficult to provide coherent guidance on security, since they all have differing ideas of "secure defaults."

Before getting started, it is worth reviewing relevant Refcardz like Getting Started With Kubernetes and Getting Started With Docker to ensure you're familiar with some of the terms and concepts discussed here.

THREAT MODEL

As with most things in security, one of the first areas to consider is what your threat model is, since thinking about who might attack your system, and how they would do it, could help prioritize your security efforts. For most Kubernetes deployments, there are three major categories of threat vectors:

1. **External attackers**: You can face attacks from outside your cluster when deployed either on premises or in the cloud. Attackers in this class have no credentials for your system, so will focus on exposed network services to attempt to gain access and elevate privileges.
2. **Compromised containers**: Kubernetes clusters are (in general) designed to run a wide variety of workloads. Attackers may be able to compromise a container running within your cluster, and at that point, it's important to contain the attack while minimizing the risk of the initial compromise widening to encompasses the whole cluster. Here, the attacker will have access to the resources of a single container, so restricting container privileges is critical.

aqua

Full Lifecycle Security for Kubernetes Deployments

- Full visibility into K8s cluster security posture
- Trusted image deployment at Kubernetes scale
- Real-time protection of K8s workloads

Learn more ›

Gain full-stack visibility into your K8s cluster security posture

Aqua Open-Source Kube-Tools

kube-hunter

Find the security gaps in your K8s clusters

Use kube-hunter to run penetration tests on your clusters using dozens of known attack vectors

Try kube-hunter ›

kube-bench

Ensure that your K8s clusters comply with security best practices

Use kube-bench to assess your K8s clusters against 100+ CIS K8s benchmark tests

Try kube-bench ›

3. **Malicious users**: Kubernetes is a multi-user system by design. The possibility of an attacker with access to a single user's credentials attempting to gain increased access to the system is likely to be relevant in a number of scenarios. Here, restrictions on what users can do become more important.

Considering these threats, some possible lines of defense from a security perspective would be as follows.

SECURING EXPOSED NETWORK SERVICES

Kubernetes has several services that make up the control plane. It's important to address the security of these services at the cluster level and not rely on external security protections like network firewalls, as an attacker may be able to access these services via vulnerabilities in the applications running on your cluster (e.g. via a server-side request forgery attack) in a way that would bypass traditional network firewall protection.

KUBERNETES API

This is the main entry point for management of a Kubernetes cluster and runs on the control plane node(s). It will typically run on `6443/TCP`, although it may also be running on `443/TCP` or `8443/TCP` depending on the cluster configuration. Additionally, on some clusters, the unsecure API port may be enabled. This is a legacy option but is still seen on some clusters. Typically, this service will be be available on port `8080/TCP`.

There have been a number of cases of unauthorized access to the Kubernetes API and, in particular, the dashboard area leading to system compromise, showing how important it is that this is secured.

Checking for the configuration of the API server to ensure that it has been configured securely is achieved by reviewing the start-up flags used when launching it. The precise location of these flags will depend on the installation method used. When using kubeadm, these options can be found in `/etc/kubernetes/manifests/kube-apiserver.yaml`. The key parameters to check are:

- `--anonymous-auth`: This should be set to "false" explicitly, as the default is to allow some anonymous access to the API server.
- `--insecure-bind-address`: This should not be set, even to the localhost address.
- `--insecure-port`: This should be set to 0 to ensure that it is not configured.

KUBELET

In addition to the main Kubernetes API, a key network service (and one which is often poorly protected) is the Kubelet which runs on some or all nodes of cluster, depending on the deployment mechanism used. This service is responsible for managing the container runtime on each cluster node (e.g., Docker or CRI-O) and as such has a wide range of privileges to the server it's running on. Unauthenticated Kubelet access is a common problem with older Kubernetes versions, as only recent versions require authentication by default, and as with the Kubernetes API, there have been a number of attacks that exploited this service due to it being left exposed to the Internet without appropriate protection.

The Kubelet will typically be running on two ports on each node. Port `10250/TCP` is the read/write service and port `10255/TCP` is the read-only port, which is used to expose information for cluster monitoring services. Unless required, remove the read-only port entirely from your configuration, as it doesn't have the option to require authentication. All access to the read/write port should require authentication. The Kubelet configuration is managed similarly to that of the Kubernetes API server via start-up parameters. The main ones to look for are:

- `--anonymous-auth`: This should be set to false.
- `--read-only-port`: This should be set to 0.

ETCD

At least one etcd key/value store is provisioned with almost every Kubernetes cluster to provide persistent storage of cluster configuration information. Unauthorized access to the etcd database can have serious consequences for the cluster's security, as it contains sensitive information such as cluster secrets.

As with the other key Kubernetes features, there has been evidence of services being left exposed on the Internet without appropriate security measures, so this is an important area to check.

The etcd database will typically listen on port `2379/TCP` for client access and `2380/TCP` for peer access. All access to it should require authentication by setting the following configuration parameters:

- `--client-cert-auth`: This should be set to true.
- `--peer-client-cert-auth`: This should also be set to true.

It's also worth reviewing your clusters for other instances of the etcd service, as some network plugins make use of separate instances for their own purposes and these may have different security settings than those set on the main database.

SECURING THE CONTAINER ENVIRONMENT

Once you've secured the management interfaces from unauthenticated access from outside the cluster, your next step in securing Kubernetes should be to analyze how attackers might compromise a pod and what might be possible for them to do. In

addition to the access available to external attackers, access to a single container may provide a number of additional avenues of attack.

- Host filesystem access
- Container network access
- Access to a service token
- Node Kernel access

Each of these attack paths can be addressed by Kubernetes security mechanisms.

SECURING CONTAINERS

The first step when considering the individual containers in an environment is trying to stop an attacker from compromising them in the first place. An attack could happen via unpatched application software, configuration issues, or errors in custom code that has been deployed into the containers.

As most container images will come from a common distribution base such as Debian or Alpine, reviewing them for missing patches is handled similarly to the process for any other Linux-based system. Custom tooling may be required, as some patch management systems are not container-aware and, as such, won't effectively scan inside the images.

There are a number of options for this. Aqua Security provides MicroScanner, which scans images based on public and proprietary sources for vulnerabilities and malware, and can be used in conjunction with Aqua's runtime protection to assess image security and block any container suspicious activity based on container runtime profiles. There are also some standalone container vulnerability-scanning tools that could be useful where convenient cloud access isn't available. Both Clair and Dagda can be used offline. It's worth noting how these tools tend to work in reviewing container images for vulnerabilities. Where they're looking at issues in system software (e.g. a web server), they usually base their analysis on the package manager used by the image (e.g. apt in Debian or Ubuntu or yum in Fedora Core). This is important, since when images that aren't based on a common distribution are used, some vulnerability-scanning tools may not be able to detect weaknesses, as there is no central vulnerability database and package metadata to query.

HOST FILESYSTEM ACCESS

An attacker who can compromise one of your containers might access any external mounts that have been made into that container. Since many containers run as the 'root' user, this could potentially allow an attacker the ability to change key operating system files if these have been exposed to a compromised container.

The obvious way to mitigate this risk is to ensure that critical host files are not mounted into exposed containers. This can be mandated by the cluster administrator using the Kubernetes PodSecurityPolicy feature. This is an admission controller that can prevent new pods from having specific privileges --- in this case, from mounting files from the underlying node operating system.

Enabling PodSecurityPolicy is carried out by adding it to the list of plugins passed to the API server using the `--enable-admission-plugins` start-up flag. However, before this change to the API server start-up process is made, an appropriate PodSecurityPolicy should be created — as if it is enabled without any policies in place, no pods can be created on the cluster.

To ensure that this kind of attack isn't possible, the Pod Security Policies used by the cluster should specify the types of volumes allowed, and this whitelist should not include the `hostPath` volume type.

CONTAINER NETWORK ACCESS

By default, Kubernetes clusters provide a flat open network for all containers running on them. It's a fundamental point of Kubernetes networking that pods should be able to contact each other at a network level.

However, as clusters grow, it's necessary to consider limiting access provided to individual applications running in the cluster so that the impact of a single compromised container can be limited.

Kubernetes provides a feature called Network Policy to enable cluster operators to limit access to and from sets of pods within the cluster. Network policies work similarly to the access control lists used on firewalls in that they can limit access to specific IP address and port combinations; however, they are aware of the cluster configuration, which means that they understand concepts like Kubernetes labels, allowing for more flexibility in how they are applied. However this is done at the pod level and not at the container level. Inbound/outbound network rules should be defined at the container level as well.

The best approach to network policies, from a security standpoint, is to apply a default deny policy to ingress and egress for all pods and containers running on the cluster, and then to allow specific access as needed — the so-called "least privilege" approach. Of course, this needs to be balanced against the practicality of maintaining these policies on the cluster.

While limiting inbound traffic should be practical to achieve, some environments may find it difficult to specify what specific egress access should be allowed. In such cases, consideration should be given to limiting access to the control plane services running on the cluster nodes. Blocking access to the ports mentioned earlier can help to prevent attackers with some level of access to the cluster from gaining further privileges.

One additional consideration regarding container network access is the use of host network access. Where a container has this access, it essentially has the same IP addresses as the underlying node operating system, including the same localhost interface. This can allow for attacks in which a service is bound to localhost with the expectation that this provides some level of isolation from attack. Host networking should be avoided wherever possible, from a security perspective. Access to the host network can be restricted using `PodSecurityPolicies`. Ensure that one of the in-use `PodSecurityPolicies` sets `hostNetwork` to `false` to prevent containers in the cluster from using host networking.

ACCESS TO A SERVICE TOKEN

One of the more unexpected features of Kubernetes, for those more used to other systems, is the use of service account tokens. These are credentials that are mounted, by default, into every pod created on the cluster. This means that every pod has some level of access to the API server to execute commands based on the rights provided to this token. From the perspective of a "compromised container" attack, this means that our attacker would have access to execute arbitrary commands against the Kubernetes API server.

Earlier versions of Kubernetes, before RBAC was widely deployed, suffered from quite a severe weakness in this scenario, as the default was to provide each pod with a token that effectively had cluster administrator level access, making it trivial for an attacker who had compromised one container to control the entire cluster.

With recent Kubernetes versions, the rights provided to the service token should be restricted based on the RBAC role assigned to it. As such, ensuring that a least-privilege approach is used for these tokens is key in maintaining the security of the cluster.

By default, service accounts should only be provided the rights of the `system:authenticated` and `system:serviceaccounts` groups. In most standard configurations, these should be fairly limited, but it's important to review them regularly to ensure that no inappropriate access has been provided. Details on how to audit the rights provided to a user or group are included in the review tools section below.

NODE KERNEL ACCESS

The final thing that access to a single container provides an attacker is the ability to attack the kernel of the underlying host operating system for the cluster node. Standard Linux containers make use of a shared kernel, so a vulnerability in that kernel can allow an attacker to break out to the underlying node.

There are a couple of strategies that should be considered to reduce this risk. The first basic one is to ensure that the kernels used on the nodes in your clusters are regularly upgraded as security patches are applied.

Secondly, restricting the privileges of containers can help to reduce the risk of a breakout via exploiting a vunerable kernel version. Many container images run as the root user, which provides more opportunities for breakout, so avoiding this will help to reduce the risk of an attack on the kernel. As with the filesystem access area mentioned above, PodSecurityPolicies can be put in place to prevent containers from running as the root user. There is a PodSecurityPolicy setting called `MustRunAsNonRoot` that will ensure that no containers which run as root can operate in the cluster.

KUBERNETES USER SECURITY

The third threat model to consider is where an attacker gets authenticated access to a cluster and can attempt to elevate privileges to get cluster administrator level access. The controls against this form of attack focus on how users are authenticated to the cluster and what authorization controls are available to limit the access that individual users have.

AUTHENTICATION

Authentication in Kubernetes is somewhat unusual for administrators who are used to "traditional" multi-user network services in that for most clusters, Kubernetes won't store details of user credentials locally but instead will rely on external data to provide that information. This can complicate the setup and management of user accounts in a cluster, so it's an important point to consider.

Kubernetes does provide two methods of authentication where credentials are managed on the master nodes of the cluster, but these are generally not considered suitable for production use as they store credentials in cleartext on the API server nodes and require a restart of the API server to update. It's possible to verify whether these are enabled by looking at the start-up flags on the Kubernetes API server. For HTTP basic authentication, the `--basic-auth-file` must be present and pointing at a file on-disk that stores user credentials. For token authentication, the `--token-auth-file` would be set.

The next (and most commonly used) authentication method is X.509 client certificates. In this scenario, the API server will look for a certificate signed by a trusted authority and take the username and group information from specific fields in the certificate (CN for username and O for groups).

From a security standpoint, this mechanism does have some drawbacks that make it less than ideal. Kubernetes currently has no facility for certificate revocation. This means that if a certificate is lost

or stolen, the only effective mitigation is to recreate the certificate authority and re-issue all certificates. Additionally, the standard configuration of Kubernetes expects the private key of the certificate authority to be available online to allow for periodic rotation of certificates. As such, any unauthorized access to this file can lead to a persistent compromise of the cluster's security for the duration of that key (which will typically be measured in years).

The general recommendation for user authentication is, therefore, for it to make use of external authentication providers. There are multiple options available to Kubernetes users in this scenario:

- **OIDC**: Kubernetes can be configured to use an OpenID Connect (OIDC) compatible system. This can be used alongside providers like GitHub and Google. Additionally, tools like Dex and Keycloak can be used to integrate other identity services (e.g. Active Directory) with Kubernetes via OIDC.
- **Webhook authentication**: Kubernetes can be configured to delegate authentication to any compatible webhook service.
- **Proxy authentication**: Sitting the API server behind a proxy server is also an option, although this is likely to be more complex to implement in many cluster architectures.

AUTHORIZATION

Similarly to the scenario around user authentication, Kubernetes provides a number of mechanisms for authorization of user requests to the API server. The primary option that is currently recommended for use is Role Based Access Control (RBAC). This provides for rights to be assigned at both the Kubernetes namespace level and cluster-wide.

To effectively implement Kubernetes authorization, it's important to understand the objects involved in providing rights to users.

The `role` and `clusterrole` objects describe the access to the API to be provided. Role objects grant access to resources in a single namespace, while `clusterrole` objects provide access to cluster-wide resources.

To go with these two objects, we have `rolebindings` and `clusterrolebindings`. These associate a subject with a role, essentially saying who has access to that role. There are a couple of important points to note, though: While a `rolebinding` ties a subject in a single namespace to a specific role, it can tie to any role or `clusterrole` object, so you can grant rights to a single user in a namespace across the whole cluster.

The second important point to note in relation to RBAC are the types of subjects that can be associated with different roles. There are three options:

1. **User**: This is a single user account as identified on the cluster. Each of the authentication methods described earlier in this Refcard will extract a username from the credentials presented and this is used by the RBAC system to assign rights.
2. **Group**: Groups can also be subjects for role bindings and cluster role bindings. What's notable here is that the membership of groups is not recorded anywhere inside the Kubernetes cluster, so there is no effective way of auditing group membership with only access to a cluster and you also need access to all the approved authentication mechanisms defined on the cluster to gather this information.
3. **Service account**: Service accounts can also be provided as the subjects for role bindings. Unlike users and groups, service accounts are stored within Kubernetes itself and so their usage can be tracked and audited with only access to the cluster.

An additional thing to note is that Kubernetes provides a number of built-in roles and some of them, like cluster-admin, provide a wide range of access to the environment and should be used sparingly.

Reviewing existing RBAC configurations can be somewhat laborious, as there's no easy way to see what rights a given subject has via the `kubectl` command. As with any Kubernetes API objects, it is possible to extract the information in JSON or YAML format by passing the appropriate flags to a get command. For example, the following two commands will export the `clusterrole` and `clusterrolebinding` objects from a cluster in JSON format:

- `kubectl get clusterroles -o json`
- `kubectl get clusterrolebindings -o json`

Alternatively, there are tools that can help to review Kubernetes RBAC information. For example, RBAC Lookup from ReactiveOps can be used to review the roles assigned to specific users. This script also shows and example of how information can be presented from the Kubernetes RBAC objects to allow for permissions to be reviewed.

TOOLS AND REFERENCES

There's a range of resources that you can use to help secure your Kubernetes clusters. The first one to mention is the main Kubernetes documentation pages. They have a good range of information about setting up, configuring, and securing clusters and are a good first port of call when looking for information about these topics.

STANDARDS AND GUIDES

The CIS Benchmark for Kubernetes is the main standard currently available. Currently, in version 1.3 (which covers Kubernetes 1.11), it provides guidance on secure configuration of your clusters. One thing to note, however, is this is security standard rather than

a configuration guide. Simply dogmatically applying all of the recommendations is unlikely to provide a good outcome. Instead, this is best used as starting point of available security options.

SECURITY ASSESSMENT TOOLS

There are a number of tools available that can help to assess the security of your clusters and make recommendations for hardening. Kube-Bench from Aqua Security checks an existing cluster against the CIS Benchmark tests. Similarly, kube-auto-analyzer carries out similar tasks, although this tool is more targeted at security reviewers with some of its checks.

Another tool focused on penetration testing is Aqua's Kube-Hunter, which probes running clusters for common security weaknesses and has capabilities to actually exploit some issues to more easily demonstrate their impact.

CONCLUSION

It's fair to say that Kubernetes is a relatively complex tool, and as with any new technology, there are security challenges to be addressed. Modern Kubernetes clusters provide many mechanisms for security to be effectively implemented. Probably the most important point to consider is ensuring that secure defaults are in place before you start using Kubernetes for production workloads.

Having a strong level of base security will help provide a good basis for ongoing development and can also minimize the risk of major changes being applied to your security posture after you've gone live.

Written by **Rory McCone**, Principal Consultant at NCC Group

Rory has worked in the Information and IT Security arena for the last 18 years in a variety of roles — from financial services to running a small security testing company to working for large companies as a consultant. These days, he spends most of his work time on application, cloud, and container security. He's an active contributor to the container security world, helping with the CIS Docker and Kubernetes guides and working on a Kubernetes Security Scanner. He has presented on application, container, and general IT Security topics at a wide range of conferences from OWASP AppsecEU to a variety of BSides conferences and KubeconEU. When he's not working, he can generally be found out and about enjoying the scenery in the highlands of Scotland if the midges aren't biting!

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300