

---

## Kubernetes Security - Best Practice Guide

This document acts as a best practice guide to Kubernetes security. K8s is a powerful platform which can be abused in many ways if not configured properly. The authors of this guide are running Kubernetes in production and worked on several K8s projects to learn about security flaws the hard way.

- General Security Guide
- Cloud Provider Security Guide
- Installer Security Guide

The severity or importance of each topic is indicated by an emoji in the topic name.

- :boom: Critical
- :fire: High
- :cloud: Medium
- :partly\_sunny: Low

If you're more into watching talks on YouTube, here are some really nice ones:

- Hacking and Hardening Kubernetes Clusters by Example [I] - Brad Geesaman, Symantec
- Shipping in Pirate-Infested Waters: Practical Attack and Defense in Kubernetes [A] - Greg Castle

### General

#### Your cluster is as secure as the system running it :fire:

Before you start looking into Kubernetes security specifics you should start with your system running Kubernetes. Go through some guides for securing your OS in general.

Here are some to begin with:

- Red Hat Enterprise Linux 6 Security Guide (*Linux in general*)

#### Private topology :partly\_sunny:

If your infrastructure allows for private IP addresses you should host the cluster in a private subnet and only forward ports that are needed from the outside from your NAT gateway to your cluster. If you're running on a cloud provider like AWS this can be achieved through a private VPC.

**Why?** In a private topology all K8s masters and nodes only have private IP addresses assigned. This greatly reduces the risk of exposing anything critical to the public by mistake or lack of knowledge.

---

Also, experience shows that if you need to expose ports explicitly you are more conscious of the potential consequences.

### Firewall ports :fire:

This is a general security best practice: Never expose a port, which doesn't need exposure. IMHO, defining port exposure should be done in the following order:

- Check if you can define a listen IP/interface to bind the service to, if possible 127.0.0.1/lo
- If selectively binding to an IP/interface is not possible, then firewall the port

Kubernetes processes like *kubelet* are opening a few ports on all network interfaces, which should be firewalled from public access. Those ports may “only” allow to query for sensitive information, but some of them allow straight full access to your cluster.

Port	Process	Description
4149/TCP	kubelet	Default cAdvisor port used to query container metrics
10250/TCP	kubelet	API which allows full node access
10255/TCP	kubelet	Unauthenticated read-only port, allowing access to node state
10256/TCP	kube-proxy	Health check server for Kube Proxy
9099/TCP	calico-felix	Health check server for Calico (if using Calico/Canal)
6443/TCP	kube-apiserver	Kubernetes API port

Health check ports are no security threat *per se* stemming from the information they expose, but critical components like the network provider could be DoSed through an exposed health check port, which would affect the whole cluster. Additionally, unknown exploits could potentially endanger security.

**Why?** Every port exposed to a network poses a potential attack vector. To minimize risk, exposure should be avoided if possible.

---

## Bastion host :cloud:

Don't provide straight public SSH access to each Kubernetes node, use a bastion host setup where you expose SSH only on one specific host from which you SSH into all other hosts. There are quite a few articles on how to do this, for example <https://www.nadeau.tv/ssh-with-a-bastion-host/>. Also SSH session recording as described in <https://aws.amazon.com/blogs/security/how-to-record-ssh-sessions-established-through-a-bastion-host/> can be useful.

For general SSH hardening check Hardening OpenSSH and the OpenSSH chapter in Applied Crypto Hardening by bettercrypto.org.

**Why?** SSH is a critical service which is under constant attack if exposed to the public. Still today, key-based authentication is not used consistently and everywhere and dictionary attacks or exploits will eventually lead to intrusion. To minimize risk a hardened bastion host is introduced and SSH blocked for public access on all other nodes.

## Kubernetes Security Scan with kube-bench :fire:

A very helpful tool to eliminate roughly 95% of the configuration flaws is kube-bench. A master or a node and their control-plane components are checked by applying the CIS Kubernetes Benchmark which results in specific guidelines to secure your cluster setup. This should be a first step before going through any specific Kubernetes security issues or security enhancements.

## API settings

**Authorization mode & anonymous auth :boom:** Some installers like *kops* will use the *AlwaysAllow* authorization mode for the cluster. This would grant any authenticated entity full cluster access. Instead, *RBAC* should be used for role-based access control. To find out what your current configuration is, check the `-authorization-mode` parameter of your *kube-apiserver* processes. More information on that topic at <https://kubernetes.io/docs/admin/authorization/>. To enforce authentication, make sure anonymous auth is disabled by setting `-anonymous-auth=false`.

**Note** This doesn't affect the *kubelet* authorization mode. The *kubelet* itself exposes an API to execute commands through which the Kubernetes API can be bypassed completely.

**Insecure Port :boom:** The insecure port (especially relevant for older Kubernetes releases) is an API port without any kind of protection. No SSL/TLS, no auth, no authz! This port really should be disabled by setting `--insecure-port=0`. Sometimes it's not possible, because of health check

---

configuration or bootstrapping mechanisms. If that's the case firewall the port from public and internal access. This flag is deprecated since v1.10.

**Disable Profiling :cloud:** It's recommended to disable the profiling API endpoint by setting `--profiling=false`.

**Why?**

Sensible program or system information can be uncovered by profiling data and because of the amount of data and load induced by profiling your cluster could be DoSed by this feature.

**AdmissionController** Add the following plugins to `--admission-control=`

**AlwaysPullImages :cloud:** `--admission-control=...,AlwaysPullImages`

**Why?**

By default Pods can specify their own image pull policy. Once an image is pulled (even if once pulled from a secured registry with credentials) other Pods could reuse the locally stored image and get access to potentially confidential information. By enabling the `AlwaysPullImages` policy the controller modifies every new Pod to force the image pull policy to Always, which ensures that credentials need to be provided every time.

**DenyEscalatingExec :boom:** `--admission-control=...,DenyEscalatingExec`

**Why?**

If pods are scheduled with `privileged: true`, `hostPID: true` or `hostIPC: true` it's possible to escalate privileges through attaching to a privileged pod or executing a command in it. The `DenyEscalatingExec` denies attach and exec for such Pods.

**PodSecurityPolicy :boom:** `--admission-control=...,PodSecurityPolicy`

**Why?**

If `PodSecurityPolicy` is not enabled, defined Pod Security Policies are not enforced and Pods violating defined policies will still be scheduled.

**Warning:** Before enabling `PodSecurityPolicy` you should have Pod security policies already in place or Pods will fail to be scheduled. See Use Pod Security Policies for a basic setup.

---

## Kublet settings

**Authorization mode & anonymous auth :boom:** The *kubelet* offers a command API used by *kube-apiserver* through which arbitrary commands can be executed on the specific node. On top of fire-walling the port (10250/TCP) from public access, the *kubelet* settings *-authorization-mode=Webhook* and *-anonymous-auth=false* should be ensured.

## Auto mount default Service Account

**With RBAC enabled: :partly\_sunny:**

**Without RBAC enabled: :boom:**

The *Admission Controller* ensures that all Pods have a service account assigned by default, which is called “default”. The credentials for this service account will be mounted into the containers file system running in the Pod unless the auto mounting feature is disabled. The mounted token can be used to query the Kubernetes API.

```
1 kubectl patch serviceaccount default -p "automountServiceAccountToken: false"
```

This will disable the auto mounting of the service account token and needs to be done on a per namespace basis.

**Note** For every new namespace, the *Admission Controller* will create the *default* service account. Changes to this service account need be applied accordingly.

## Use Network Policies :cloud:

Network Policies are firewall rules for Kubernetes. If you’re using a network provider which supports Network Policies, you should definitely use them to secure internal cluster communication and external cluster access. By default, there are no restrictions in place to limit pods from communicating with each other.

Check Kubernetes Network Policy Recipes for an awesome starting point. If your network provider doesn’t support network policies, consider switching to one which does, check <https://kubernetes.io/docs/concepts/cluster-administration/networking/>.

## Use Pod Security Policies :cloud:

Pod security policies allow for controlling security sensitive aspects of the pod specification.

---

## Why?

Most (almost all) of your Pods don't need privileged access or even host access, so it should be ensured that a Pod requesting such access needs to be white listed explicitly. By default no one should be able to request privileges above the default to avoid being vulnerable through misconfiguration or malicious content of a Docker image.

A very basic setup consists of a unprivileged and a privileged policy. The unprivileged is called "default" and the privileged is called, well, "privileged".

**Important:** Check if you're cluster nodes support AppArmor or not, as different default policies need to be created accordingly.

With AppArmor:

```
1 kubectl apply -f https://raw.githubusercontent.com/freache/kubernetes-security-best-practice/master/PSP/default.psp.yaml
2 kubectl apply -f https://raw.githubusercontent.com/freache/kubernetes-security-best-practice/master/PSP/privileged.psp.yaml
```

Without AppArmor:

```
1 kubectl apply -f https://raw.githubusercontent.com/freache/kubernetes-security-best-practice/master/PSP/default-non-apparmor.psp.yaml
2 kubectl apply -f https://raw.githubusercontent.com/freache/kubernetes-security-best-practice/master/PSP/privileged.psp.yaml
```

Pod Security Policies are evaluated based on access to the policy. When multiple policies are available, the pod security policy controller selects policies in the following order:

1. If any policies successfully validate the pod without altering it, they are used.
2. If it is a pod creation request, then the first valid policy in alphabetical order is used.
3. Otherwise, if it is a pod update request, an error is returned, because pod mutations are disallowed during update operations.

In order to use a policy, the requesting user **or** target pod's service account must be authorized to use the policy, by allowing the *use* verb on the policy. So, when defining access rules for policies, we need to think about which user is creating/updating the Pod. The user is different if a Pod is created directly through *kubectl* or through a deployment.

First we make sure, that any user has access to the *default* policy, which ensures that Pods will be unprivileged by default.

```
1 kubectl apply -f https://raw.githubusercontent.com/freache/kubernetes-security-best-practice/master/PSP/default-psp.clusterrolebinding.yaml
```

---

```
2 kubectl apply -f https://raw.githubusercontent.com/freache/kubernetes-security-best-practice/master/PSP/default-psp.clusterrole.yaml
```

Some Pods in the cluster, especially if *kube-apiserver*, *kube-controller-manager*, *kube-scheduler* or *etcd* are running inside the cluster, need privileged access. To ensure those services will still start after introducing the *PodSecurityPolicy* controller, we need to grant cluster nodes and the legacy kubelet user access to the privileged policy for the *kube-system* namespace. For this to work make sure you've `--authorization-mode=Node, RBAC`.

```
1 kubectl apply -f https://raw.githubusercontent.com/freache/kubernetes-security-best-practice/master/PSP/privileged-psp.clusterrole.yaml
2 kubectl apply -f https://raw.githubusercontent.com/freache/kubernetes-security-best-practice/master/PSP/privileged-psp-nodes.rolebinding.yaml
```

Your network provider will also need privileged access. Depending on which you're using the used service account is different. For canal you need to create the following role binding.

```
1 kubectl create -n kube-system -f - <<EOF
2 kind: RoleBinding
3 apiVersion: rbac.authorization.k8s.io/v1
4 metadata:
5   name: privileged-psp-canal
6   namespace: kube-system
7 roleRef:
8   kind: ClusterRole
9   name: privileged-psp
10 apiGroup: rbac.authorization.k8s.io
11 subjects:
12 - kind: ServiceAccount
13   name: canal
14   namespace: kube-system
15 EOF
```

With a *kops* setup you will also need a role binding for the *dns-controller* and *kube-dns-autoscaler*.

```
1 kubectl create -n kube-system -f - <<EOF
2 kind: RoleBinding
3 apiVersion: rbac.authorization.k8s.io/v1
4 metadata:
5   name: privileged-psp-dns
6   namespace: kube-system
7 roleRef:
8   kind: ClusterRole
9   name: privileged-psp
10 apiGroup: rbac.authorization.k8s.io
11 subjects:
12 - kind: ServiceAccount
13   name: kube-dns-autoscaler
```

---

```
14 namespace: kube-system
15 - kind: ServiceAccount
16   name: dns-controller
17   namespace: kube-system
18 EOF
```

Before enabling the *PodSecurityPolicy* controller you should check your namespaces for Pods requiring privileged access and create role bindings accordingly. If you're certain all Pods are covered you can add "PodSecurityPolicy" to `--admission-control=...` of your *kube-apiserver* configuration and restart the API.

You can test your setup by creating a deployment like this:

```
1 kubectl create -f -<<EOF
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: privileged
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10      name: privileged
11   template:
12     metadata:
13       labels:
14         name: privileged
15     spec:
16       containers:
17         - name: pause
18           image: k8s.gcr.io/pause
19           securityContext:
20             privileged: true
21 EOF
```

If all is fine, the *privileged* deployment should fail to create the Pod.

### Restrict “docker image pull” :fire:

Docker images are a completely uncontrolled environment. Everyone with access to the Docker socket or Kubernetes API can pull any image they like. Because of that many Kubernetes clusters secretly became Bitcoin miners, because of infected Docker images or Kubernetes security issues. The Docker plugin Docker Image policy plugin will help you with that problem. The plugin hooks into the internal Docker API and enforces a set of black and white list rules to restrict what images can be pulled.



---

Ultimately Docker is pulling an image, so securing Docker is considered a good approach but alternatively Kubernetes also provides a way. The *AdmissionController* provides the *ImagePolicyWebhook* through which a provided web service can intercept image pulls.

### Kubernetes Dashboard :boom:

Prio to version 1.8.0, the `kubernetes-dashboard` plugin was granted a service account with full cluster access to be able to see and manage all aspects of the cluster.

Verify that there is no *ClusterRolebinding* to `cluster-admin` left behind. Otherwise clicking *SKIP* on the sign-in page will grant full access.

```
kubectl -n kube-system get clusterrolebinding kubernetes-dashboard -o yaml
```

By default the dashboard is not exposed to the public Internet and it should be avoided to change that. Reasons why we could see with the Tesla hack discovered by RedLock.

If you're using a network provider plugin which supports network policies you should also block requests to the dashboard coming from inside the cluster (other Pods). This will not block requests coming through `kubectl proxy`.

```
1 kind: NetworkPolicy
2 apiVersion: networking.k8s.io/v1
3 metadata:
4   name: deny-dashboard
5   namespace: kube-system
6 spec:
7   podSelector:
8     matchLabels:
9       k8s-app: kubernetes-dashboard
10  policyTypes:
11    - Ingress
```

### Securing a Cluster (by Kubernetes project)

The Kubernetes project itself also has some notes on how to secure a cluster. See the *Securing a Cluster* chapter from the Kubernetes docs.

### Cloud Provider Guide

- AWS

---

## **Installer Guide**

- kops

## **Author**

- Simon Pirschel - Kubernetes Consultant & DevOps Specialist

Thanks to all the contributors to this guide assuring good quality.