

# Definitive Guide to Kubernetes Security



StackRox



# Table of Contents

Overview	3
Role-Based Access Control (RBAC)	4
Enabling RBAC	5
Main Concepts	5
Four RBAC Best Practices	6
Network Policies that Maximize Security	7
How to Set Up Network Policies	8
Additional Resources	12
Kubernetes Admission Controllers	14
What are Kubernetes admission controllers?	15
Why You Need Admission Controllers	16
Example: Writing and Deploying an Admission Controller Webhook	17
Building Production-Ready Kubernetes Clusters	21
Containers Done Right	22
Organizing Your Cluster	23
Securing Your Cluster	25
Keeping Your Cluster Stable	26
Advanced Topics	27
Top Nine Kubernetes Settings You Should Check Right Now to Optimize Security	29
How StackRox Secures Containers and Kubernetes	35





# Overview

Kubernetes has become the de facto standard in orchestrators for containers and microservices. Organizations are transforming their businesses by embracing DevOps principles, microservice design patterns, and container technologies across on-premises, cloud, and hybrid environments. Security concerns, however, are impacting most organizations' container and Kubernetes adoption strategies.

Whether you're managing Kubernetes yourself or using one of the many flavors offered by third-party technology vendors (such as EKS, AKS, GKE, or OpenShift), security cannot be an afterthought. Security must instead be built into each phase of the container life cycle: build, deploy, and runtime. Security misconfigurations and vulnerabilities introduced in the build phase, for example, can be very costly to fix after containerized applications are in production.

The most common security considerations include:

- Ensuring images are free of vulnerabilities before moving to deploy phase
- Leveraging Kubernetes admission controllers to enforce governance policies
- Controlling and limiting access to the Kubernetes API and what actions are allowed
- Enforcing limits on Kubelet to protect nodes and containers

- Limiting resource utilization on clusters
- Limiting the privileges that containers run with and enforcing least-privilege best practices
- Protecting sensitive information such as the Kubernetes dashboard, metadata, and secrets
- Restricting network communications to only what's needed for an application to function vs. the "all open" connections enabled by default in Kubernetes
- Detecting and stopping anomalous or malicious activities at runtime

By operationalizing some of the advanced Kubernetes security capabilities and controls, you can begin to address some of these most common security concerns. This guide offers practical tips to help you operationalize Kubernetes security so that you're not only building and deploying production-ready applications and services but also ensuring continued protection in runtime environments. We will take a deep dive in key areas including:

- Role-Based Access Control (RBAC)
- Network Policies
- Admission Controllers

We've also compiled a pair of checklists for you to get started with immediately to shore up your container and Kubernetes security posture and prepare your clusters for production environments.



# Role-Based Access Control (RBAC)

## 4 Best Practices



Kubernetes RBAC configuration is a critical control for the security of your containerized workloads. Properly configuring your cluster RBAC roles and bindings helps minimize the impact of application compromises, user account takeovers, application bugs, or simple human mistakes.

Because the feature is relatively new, your organization might have configured RBAC in a manner that leaves you unintentionally exposed. In a moment we will walk you through four best practices that will ensure you have granted your team members appropriate privileges and addressed unintentional vulnerabilities.

Before we get to the best practices, let's make sure that RBAC is enabled, and then review the main concepts in the Kubernetes RBAC system.

## Enabling RBAC

Different Kubernetes distributions and platforms have enabled RBAC by default at different times, and newly upgraded older clusters may still not enforce RBAC because the legacy Attribute-Based Access Control (ABAC) controller is still active. If you're using a cloud provider, this setting is typically visible in the cloud console or using the provider's command-line tool. For instance, on Google Kubernetes Engine, you can check this setting on all of your clusters using `gcloud`:

```
$ gcloud container clusters list --  
format='table[box] (name, legacyAbac.enabled) '
```

NAME	ENABLED
with-rbac	True
with-abac	

## Main Concepts

Your cluster's RBAC configuration controls which **subjects** can execute which **verbs** on which **resource types** in which namespaces. For example, a configuration might grant user `alice` access to **view** resources of type `pod` in the namespace `external-api`. (Resources are also scoped inside of API groups.)

These access privileges are synthesized from definitions of:

- Roles, which define lists of rules. Each rule is a combination of verbs, resource types, and namespace selectors. (A related noun, Cluster Role, can be used to refer to resources that aren't namespace-specific, such as nodes.)
- Role Bindings, which connect ("bind") roles to subjects (users, groups, and service accounts). (A related noun, Cluster Role Binding, grants access across all namespaces.)

In Kubernetes 1.9 and later, Cluster Roles can be extended to include new rules using the Aggregated ClusterRoles feature. This design enables fine-grained access limits.



## Four RBAC Best Practices

To achieve least privilege without leaving unintentional weaknesses, be sure you haven't made any of the following configuration mistakes.



### Tailor the Role or Cluster Role to Specific Users

The built-in `cluster-admin` role grants effectively unlimited access to the cluster. During the transition from the legacy ABAC controller to RBAC, some administrators and users may have replicated ABAC's permissive configuration by granting `cluster-admin` widely, neglecting the warnings in the relevant documentation. If users or groups are routinely granted `cluster-admin`, account compromises or mistakes can have dangerously broad effects. Service accounts typically also do not need this type of access. In both cases, a more tailored Role or Cluster Role should be created and granted only to the specific users that need it.



### Carefully Review Role Aggregation

In Kubernetes 1.9 and later, Role Aggregation can be used to simplify privilege grants by allowing new privileges to be combined into existing roles. However, if these aggregations are not carefully reviewed, they can change the intended use of a role; for instance, the `system:view` role could improperly aggregate rules with verbs other than view, violating the intention that subjects granted `system:view` can never modify the cluster.



### Check for Duplication in Role Grant

Role definitions may overlap with each other, giving subjects the same access in more than one way. Administrators sometimes intend for this overlap to happen, but this configuration can make it more difficult to understand which subjects are granted which accesses. And, this situation can make access revocation more difficult if an administrator does not realize that multiple role bindings grant the same privileges.



### Remove Unused or Inactive Roles

Roles that are created but not granted to any subject can increase the complexity of RBAC management. Similarly, roles that are granted only to subjects that do not exist (such as service accounts in deleted namespaces or users who have left the organization) can make it difficult to see the configurations that do matter.

In addition, role bindings can reference roles that do not exist. If the same role name is reused for a different purpose in the future, these inactive role bindings can suddenly and unexpectedly grant privileges to subjects other than the ones the new role creator intends.

Removing unused or inactive roles is typically safe and will focus attention on the active roles.



# Network Policies that Maximize Security



Kubernetes is geared towards making it easy for users to get up and running quickly, as well as being backward compatible with earlier releases of Kubernetes that lacked important security features. Consequently, many important Kubernetes configurations are not secure by default.

One important configuration that demands attention from a security perspective is the network policies feature. Network policies specify how groups of pods are allowed to communicate with each other and other network endpoints. You can think of them as the Kubernetes equivalent of a firewall. In Kubernetes, these policies are set to “open” by default, so every asset can talk to every other asset.

## How to Set Up Network Policies

The information that follows is a step-by-step guide on how to set up network policies, with recommendations that significantly improve security. While the network policy spec is intricate and can be difficult to understand and use correctly, users can easily implement the following recommendations without needing to know the spec in detail.

A quick note: we are addressing only ingress network policies. When starting out, the biggest security gains come from applying ingress policies, so we recommend focusing on them first, and then adding egress policies.

### 1. Use a network plugin that supports network policies

First things first – use a network plugin that actually enforces network policies. Although Kubernetes always supports operations on the **NetworkPolicy** resource, simply creating the resource without a plugin that implements it will have no effect. Example plugins include Calico, Cilium, Kube-router, Romana and Weave Net.

### 2. “Isolate” your pods

Each network policy has a **podSelector** field, which selects a group of (zero or more) pods. When a pod is selected by a network policy, the network policy is said to apply to it.

Each network policy also specifies a list of allowed (ingress and egress) connections. When the network policy is created, all the pods that it applies to are allowed to make or accept the connections listed in it. In other words, a network policy is essentially a whitelist of allowed connections – a connection to or from a pod is allowed if it is permitted by at least one of the network policies that apply to the pod.

This tale, however, has an important twist: based on everything described so far, one would think that, if no network policies applied to a pod, then no connections to or from it would be permitted. The opposite, in fact, is true: if no network policies apply to a pod, then all network connections to and from it are permitted (unless the connection is forbidden by a network policy applied to the other peer in the connection.)

This behavior relates to the notion of “isolation”: pods are “isolated” if at least one network policy applies to them; if no policies apply, they are “non-isolated”. Network policies are not enforced on non-isolated pods. Although somewhat counter-intuitive, this behavior exists to make it easier to get a cluster up and running – a user who does not understand network policies can run their applications without having to create one.





Therefore, we recommend you start by applying a “default-deny-all” network policy. The effect of the following policy specification is to isolate all pods, which means that only connections explicitly whitelisted by other network policies will be allowed.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

Without such a policy, it is very easy to run into a scenario where you delete a network policy, hoping to forbid the connections listed in it, but find that the result is that all connections to some pods suddenly become permitted – including ones that weren’t allowed before. Such a scenario occurs when the network policy you deleted was the only one that applied to a particular pod, which means that the deletion of the network policy caused the pod to become “non-isolated”.

**Important Note:** Since network policies are namespaced resources, you will need to create this policy for each namespace. You can do so by running `kubectl -n <namespace> create -f <filename>` for each namespace.

### 3. Explicitly allow internet access for pods that need it

With just the `default-deny-all` policy in place in every namespace, none of your pods will be able to talk to each other or receive traffic from the Internet. For most applications to work, you will need to allow some pods to receive traffic from outside sources. One convenient way to permit this setup would be to designate labels that are applied to those pods to which you want to allow access from the internet and to create network policies that target those labels. For example, the following network policy allows traffic from all (including external) sources for pods having the `networking/allow-internet-access=true` label (again, as in the previous section, you will have to create this for every namespace):

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-same-namespace
spec:
  podSelector: {}
  policyTypes:
    - Ingress
  ingress:
    - from:
      - podSelector: {}
```

For a more locked-down set of policies, you would ideally want to specify more fine-grained CIDR blocks as well as explicitly list out allowed ports and protocols. However, this policy provides a good starting point, with much greater security than the default.



## 4. Explicitly allow necessary pod-to-pod communication

After taking the above steps, you will also need to add network policies to allow pods to talk to each other. You have a few options for how to enable pod-to-pod communications, depending on your situation:

### If You Don't Know Which Pods Need to Talk To Each Other

In this case, a good starting point is to allow all pods in the same namespace to talk to each other and explicitly whitelist communication across namespaces, since that is usually more rare. You can use the following network policy to allow all pod-to-pod communication within a namespace:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: internet-access
spec:
  podSelector:
    matchLabels:
      networking/allow-internet-access: "true"
  policyTypes:
  - Ingress
  ingress:
  - {}
```

### If You Know the Sources and Sinks for Communication

Often, communication between pods in an application follows a hub-and-spoke paradigm, with some central pods that many other pods need to talk to. In this case, you could consider creating a label which designates pods that are allowed to talk to the "hub." For example, if your hub is a database pod and has an `app=db` label, you could allow access to the database only from pods that have a `networking/allow-db-access=true` label by applying the following policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-db-access
spec:
  podSelector:
    matchLabels:
      app: "db"
  policyTypes:
  - Ingress:
    ingress:
    - from:
      - podSelector:
          matchLabels:
            networking/allow-db-access: "true"
```



You could do something similar if you have a server that initiates connections to many other pods. If you want to explicitly whitelist the pods that the server is allowed to talk to, you can set the `networking/allow-server-to-access=true` label on them, and apply the following network policy (assuming your server has the label `app=server`) on them:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-db-access
spec:
  podSelector:
    matchLabels:
      networking/allow-db-access: "true"
  policyTypes:
  - Ingress:
    ingress:
    - from:
      - podSelector:
          matchLabels:
            app: "server"
```

## If You Know Exactly Which Connections Should Be Allowed

Within the same namespace

Advanced users who know exactly which pod-to-pod connections should be allowed in their application can explicitly allow each such connection. If you want pods in deployment A to be able to talk to pods in deployment B, you can create the following policy to whitelist that connection, after replacing the labels with the labels of the specific deployment:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy

metadata:
  name: allow-server-to-access
spec:
  podSelector:
    matchLabels:
      deployment-b-pod-label-1-key: deployment-b-pod-label-1-value
      deployment-b-pod-label-2-key: deployment-b-pod-label-2-value
  policyTypes:
  - Ingress:
    ingress:
    - from:
      - podSelector:
          matchLabels:
            deployment-a-pod-label-1-key: deployment-a-pod-label-1-value
            deployment-a-pod-label-2-key: deployment-a-pod-label-2-value
```



## Across Namespaces

To allow connections across namespaces, you will need to create a label for the source namespace (unfortunately, Kubernetes does not have any labels on namespaces by default) and add a `namespaceSelector` query next to the `podSelector` query. To label a namespace, you can simply run the command: `kubectl label namespace <name> networking/namespace=<name>`

With this namespace label in place, you can allow deployment A in namespace N1 to talk to deployment B in namespace N2 by applying the following network policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy

metadata:
  name: allow-ingress-from-new
  namespace: N2
spec:
  podSelector: {}
  policyTypes:
  - Ingress:
    ingress:
    - from:
      - podSelector:
          matchLabels:
            networking/allow-all-connections: "true"
---

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy

metadata:
  name: allow-ingress-from-new
spec:
  podSelector:
    matchLabels:
      networking/allow-all-connections: "true"
  policyTypes:
  - Ingress:
    ingress:
    - from:
      - podSelector: {}
```

## What About New Deployments?

Although explicitly whitelisting connections in this manner is great for security, this approach does affect usability. When you create new deployments, they will not be able to talk to anything by default until you apply a network policy. To mitigate this potentially frustrating user experience, you could create the following pair of network policies, which allow pods labeled `networking/allow-all-connections=true` to talk to all other pods in the same namespace:



```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy

metadata:
  name: allow-n1-a-to-n2-b
  namespace: N2
spec:
  podSelector:
    matchLabels:
      deployment-b-pod-label-1-key: deployment-b-pod-label-1-value
      deployment-b-pod-label-2-key: deployment-b-pod-label-2-value
  policyTypes:
    - Ingress:
      ingress:
        - from:
            - namespaceSelector:
                matchLabels:
                  networking/namespace: N1
            podSelector:
                matchLabels:
                  deployment-a-pod-label-1-key: deployment-a-pod-label-1-value
                  deployment-a-pod-label-2-key: deployment-a-pod-label-2-value
```

You can then apply the `networking/allow-all-connections=true` label to all newly created deployments, so that your application works until you create specially crafted network policies for them, at which point you can remove the label.

### Additional Resources

While these recommendations provide a good starting point, network policies are a lot more involved. If you're interested in exploring them in more detail, be sure to check out the Kubernetes tutorial (<https://kubernetes.io/docs/tasks/administer-cluster/declare-network-policy/>) as well as some handy network policy recipes (<https://github.com/ahmetb/kubernetes-network-policy-recipes>).

**Note:** All the example YAMLs in this article can be downloaded at <https://github.com/stackrox/network-policy-examples>



# Kubernetes Admission Controllers



Within the range of Kubernetes features that secure production workloads, admission controllers are a set of plug-ins that, when enabled, allow you to use some of the more advanced security features, including pod security policies that enforce a security configuration baseline across an entire namespace.

This section of the guide will give you the tips and tricks that are imperative when leveraging admission controllers to make the most of the security capabilities in Kubernetes. First we will cover some basics about admission controllers, and then we will dig into a powerful example (with accompanying available code) that will help you get started on leveraging these powerful capabilities.

### What are Kubernetes admission controllers?

In a nutshell, Kubernetes admission controllers are plugins that govern and enforce how the cluster is used. They can be thought of as a gatekeeper that intercept (authenticated) API requests and may change the request object or deny the request altogether. The admission control process has two phases: the mutating phase is executed first, followed by the validating phase. Consequently, admission controllers can act as mutating or validating controllers or as a combination of both. For example, the **LimitRanger** admission controller can augment pods with default resource requests and limits (mutating phase), as well as verify that pods with explicitly set resource requirements do not exceed the per-namespace limits specified in the **LimitRange** object (validating phase).

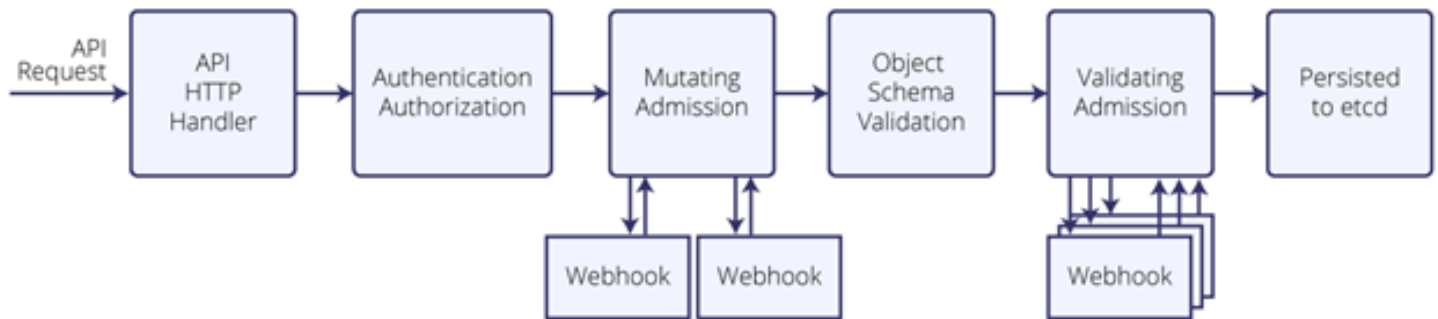


Fig. 1 – Admission controller phases

It is worth noting that some aspects of Kubernetes' operation that many users would consider built-in are in fact governed by admission controllers. For example, when a namespace is deleted and subsequently enters the Terminating state, the **NamespaceLifecycle** admission controller is what prevents any new objects from being created in this namespace.

Among the more than 30 admission controllers shipped with Kubernetes, two take a special role because of their nearly limitless flexibility - **ValidatingAdmissionWebhooks** and **MutatingAdmissionWebhooks**, both of which are in beta status as of Kubernetes 1.13. We will examine these two admission controllers closely, as they do not implement any policy decision logic themselves. Instead, the respective action is obtained from a REST endpoint (a webhook) of a service running inside the cluster. This approach decouples the admission controller logic from the Kubernetes API server, thus allowing users to implement custom logic to be executed whenever resources are created, updated, or deleted in a Kubernetes cluster.



The difference between the two kinds of admission controller webhooks is pretty much self-explanatory: mutating admission webhooks may mutate the objects, while validating admission webhooks may not. However, even a mutating admission webhook can reject requests and thus act in a validating fashion. Validating admission webhooks have two main advantages over mutating ones: first, for security reasons it might be desirable to disable the `MutatingAdmissionWebhook` admission controller (or apply stricter RBAC restrictions as to who may create `MutatingWebhookConfiguration` objects) because of its potentially confusing or even dangerous side effects. Second, as shown in the previous diagram, validating admission controllers (and thus webhooks) are run after any mutating ones. As a result, whatever request object a validating webhook sees is the final version that would be persisted to `etcd`.

The set of enabled admission controllers is configured by passing a flag to the Kubernetes API server. Note that the old `--admission-control` flag was deprecated in 1.10 and replaced with `--enable-admission-plugins`.

```
--enable-admission  
plugins=ValidatingAdmissionWebhook,MutatingAdmissionWebhook
```

Kubernetes recommends the following admission controllers to be enabled by default.

```
--enable-admission-plugins=NamespaceLifecycle,Limi  
tRanger,ServiceAccount,DefaultStorageClass,Default  
TolerationSeconds,MutatingAdmissionWebhook,Validat-  
ingAdmissionWebhook,Priority,ResourceQuota,PodSecurityPolicy
```

The complete list of admission controllers with their descriptions can be found in the official Kubernetes reference (<https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#what-does-each-admission-controller-do>). This discussion will focus only on the webhook-based admission controllers.

## Why You Need Admission Controllers

- **Security:** Admission controllers can increase security by mandating a reasonable security baseline across an entire namespace or cluster. The built-in `PodSecurityPolicy` admission controller is perhaps the most prominent example; it can be used for disallowing containers from running as root or making sure the container's root filesystem is always mounted read-only, for example. Further use cases that can be realized by custom, webhook-based admission controllers include:
  - Allow pulling images only from specific registries known to the enterprise, while denying unknown image registries.
  - Reject deployments that do not meet security standards. For example, containers using the privileged flag can circumvent a lot of security checks. This risk could be mitigated by a webhook-based admission controller that either rejects such deployments (validating) or overrides the privileged flag, setting it to false.





- **Governance:** Admission controllers allow you to enforce the adherence to certain practices such as having good labels, annotations, resource limits, or other settings. Some of the common scenarios include:
  - Enforce label validation on different objects to ensure proper labels are being used for various objects, such as every object being assigned to a team or project, or every deployment specifying an app label.
  - Automatically add annotations to objects, such as attributing the correct cost center for a “dev” deployment resource.
- **Configuration management:** Admission controllers allow you to validate the configuration of the objects running in the cluster and prevent any obvious misconfigurations from hitting your cluster. Admission controllers can be useful in detecting and fixing images deployed without semantic tags, such as by:
  - automatically adding resource limits or validating resource limits,
  - ensuring reasonable labels are added to pods, or
  - ensuring image references used in production deployments are not using the latest tags, or tags with a -dev suffix.


In this way, admission controllers and policy management help make sure that applications stay in compliance within an ever-changing landscape of controls.

## Example: Writing and Deploying an Admission Controller Webhook

To illustrate how admission controller webhooks can be leveraged to establish custom security policies, let’s consider an example that addresses one of the shortcomings of Kubernetes: a lot of its defaults are optimized for ease of use and reducing friction, sometimes at the expense of security. One of these settings is that containers are by default allowed to run as root (and, without further configuration and no USER directive in the Dockerfile, will also do so). Even though containers are isolated from the underlying host to a certain extent, running containers as root does increase the risk profile of your deployment– and should be avoided as one of many security best practices (<https://www.stackrox.com/post/2018/12/6-container-security-best-practices-you-should-be-following/>). The recently exposed runC vulnerability(<https://www.stackrox.com/post/2019/02/the-runc-vulnerability-a-deep-dive-on-protecting-yourself/>) (CVE-2019-5736 (<https://nvd.nist.gov/vuln/detail/CVE-2019-5736>)), for example, could be exploited only if the container ran as root.

You can use a custom mutating admission controller webhook to apply more secure defaults: unless explicitly requested, our webhook will ensure that pods run as a non-root user (we assign the user ID 1234 if no explicit assignment has been made). Note that this setup does not prevent you from deploying any workloads in your cluster, including those that legitimately require running as root. It only requires you to explicitly enable this more risky mode of operation in the deployment configuration, while defaulting to non-root mode for all other workloads.

The full code along with deployment instructions can be found in our accompanying GitHub repository (<https://github.com/stackrox/admission-controller-webhook-demo>). Here, we will highlight a few of the more subtle aspects about how webhooks work.



## Mutating Webhook Configuration

A mutating admission controller webhook is defined by creating a `MutatingWebhookConfiguration` object in Kubernetes. In our example, we use the following configuration:

```
apiVersion:
admissionregistration.k8s.io/vbeta1
kind: MutatingWebhookConfiguration
metadata:
  name: demo-webhook
webhooks:
- name: webhook-server,webhook.demo.svc
  clientConfig:
    service:
      name: webhook-server
      namespace:webhook-demo
      path: "/mutate"
      caBundle: ${CA_PEM_B64}
  rules:
    - operations: [ "CREATE" ]
      apiGroups: [ "" ]
      apiVersions: [ "v1" ]
      resources: [ "pods" ]
```

This configuration defines a webhook `webhook-server.webhook-demo.svc`, and instructs the Kubernetes API server to consult the service `webhook-server` in namespace `webhook-demo` whenever a pod is created by making a HTTP POST request to the `/mutate` URL. For this configuration to work, several prerequisites have to be met.

## Webhook REST API

The Kubernetes API server makes an HTTPS POST request to the given service and URL path, with a JSON-encoded `AdmissionReview` (<https://github.com/kubernetes/api/blob/master/admission/v1beta1/types.go#L29>) (with the `Request` field set) in the request body. The response should in turn be a JSON-encoded `AdmissionReview`, this time with the `Response` field set.

Our demo repository contains a function ([https://github.com/stackrox/admission-controller-webhook-demo/blob/master/cmd/webhook-server/admission\\_controller.go#L132](https://github.com/stackrox/admission-controller-webhook-demo/blob/master/cmd/webhook-server/admission_controller.go#L132)) that takes care of the serialization/deserialization boilerplate code and allows you to focus on implementing the logic operating on Kubernetes API objects. In our example, the function implementing the admission controller logic is called `applySecurityDefaults`, and an HTTPS server serving this function under the `/mutate` URL can be set up as follows:

```
mux := http.NewServeMux()
mux.Handle("/mutate",
admitFuncHandler(applySecurityDefaults))
server := &http.Server{
  Addr:    ":8443",
  Handler: mux,
}
log.Fatal(server.ListenAndServeTLS(certPath, keyPath))
```



Note that for the server to run without elevated privileges, we have the HTTP server listen on port 8443. Kubernetes does not allow specifying a port in the webhook configuration; it always assumes the HTTPS port 443. However, since a service object is required anyway, we can easily map port 443 of the service to port 8443 on the container:

```
apiVersion: v1
kind: Service
metadata:
  name: webhook-server
  namespace: webhook-demo
spec:
  selector:
    app: webhook-server # specified by the deployment/pod
  ports:
    - port: 443
      targetPort: webhook-api # name of port 8443 of the container
```

## Object Modification Logic

In a mutating admission controller webhook, mutations are performed via JSON patches. While the JSON patch standard includes a lot of intricacies that go well beyond the scope of this discussion, the Go data structure in our example as well as its usage should give the user a good initial overview of how JSON patches work:

```
type patchOperation struct {
  Op      string      `json:"op"`
  Path    string      `json:"path"`
  Value   interface{} `json:"value,omitempty"`
}
```

For setting the field `.spec.securityContext.runAsNonRoot` of a pod to true, we construct the following `patchOperation` object:

```
patches = append(patches, patchOperation{
  Op:      "add",
  Path:    "/spec/securityContext/runAsNonRoot",
  Value:   true,
})
```

## TLS Certificates

Since a webhook must be served via HTTPS, we need proper certificates for the server. These certificates can be self-signed (rather: signed by a self-signed CA), but we need Kubernetes to instruct the respective CA certificate when talking to the webhook server. In addition, the common name (CN) of the certificate must match the server name used by the Kubernetes API server, which for internal services is `<service-name>.<namespace>.svc`, i.e., `webhook-server.webhook-demo.svc` in our case. Since the generation of self-signed TLS certificates is well documented across the Internet, we simply refer to the respective shell script in our example.



The webhook configuration shown previously contains a placeholder `${CA_PEM_B64}`. Before we can create this configuration, we need to replace this portion with the Base64-encoded PEM certificate of the CA. The `openssl base64 -A` command can be used for this purpose.

## Testing the Webhook

After deploying the webhook server and configuring it, which can be done by invoking the `./deploy.sh` script from the repository, it is time to test and verify that the webhook indeed does its job. The repository contains three examples (<https://github.com/stackrox/admission-controller-webhook-demo/tree/master/examples>):

- A pod that does not specify a security context (`pod-with-defaults`). We expect this pod to be run as non-root with user id 1234.
- A pod that does specify a security context, explicitly allowing it to run as root (`pod-with-override`).
- A pod with a conflicting configuration, specifying it must run as non-root but with a user id of 0 (`pod-with-conflict`). To showcase the rejection of object creation requests, we have augmented our admission controller logic to reject such obvious misconfigurations.

Create one of these pods by running `kubectl create -f examples/<name>.yaml`. In the first two examples, you can verify the user id under which the pod ran by inspecting the logs, for example:

```
$ kubectl create -f examples/pod-with-defaults.yaml
$ kubectl logs pod-with-defaults
I am running as user 1234
```

In the third example, the object creation should be rejected with an appropriate error message.

```
$ kubectl create -f examples/pod-with-conflict.yaml
Error from server (InternalError): error when creating "examples/pod-with-conflict.yaml": Internal error occurred: admission webhook "webhook-server.webhook-demo.svc" denied the request: runAsNonRoot specific, but runAsUser set to 0 (the root user)
```

Feel free to test this with your own workloads as well. Of course, you can also experiment a little bit further by changing the logic of the webhook and see how the changes affect object creation. More information on how to do experiment with such changes can be found in the repository's readme.



# Building Production-Ready Kubernetes Clusters



Kubernetes is complex. It's difficult to know when you've set things up correctly and it's safe to flip the switch and open the network floodgates to your services. To that end, we've compiled the following 16-point checklist to help you prepare your containers and Kubernetes clusters for production traffic.

## Containers Done Right

Kubernetes provides a way to orchestrate containerized services, so if you don't have your containers in order, your cluster isn't going to be in good shape from the get-go. Follow these tips to start out on the right foot.

### 1. Use minimal base images

**What:** Containers are application stacks built into a system image. Everything from your business logic to the kernel gets packed inside. Minimal images strip out as much of the OS as possible and force you to explicitly add back any components you need.

**Why:** Including in your container only the software you intend to use has both performance and security benefits. You have fewer bytes on disk, less network traffic for images being copied, and fewer tools for potential attackers to access.

**How:** Alpine Linux is a popular choice and has broad support.

### 2. Use a registry that offers the best uptime

**What:** Registries are repositories for images, making those images available for download and launch. When you specify your deployment configuration, you'll need to specify where to get the image with a path `<registry>/<remote name>:<tag>` :

```
apiVersion: v1
kind: Deployment
...
spec:
...
  containers
  - name: app
    image: docker.io/app-image:version1
```

**Why:** Your cluster needs images to run.

**How:** Most cloud providers offer private image registry services: Google offers the Google Container Registry, AWS provides Amazon ECR, and Microsoft has the Azure Container Registry.

Do your homework, and choose a private registry that offers the best uptime. Since your cluster will rely on your registry to launch newer versions of your software, any downtime will prevent updates to running services.



### 3. Use ImagePullSecrets to authenticate your registry

**What:** ImagePullSecrets are Kubernetes objects that let your cluster authenticate with your registry, so the registry can be selective about who is able to download your images.

**Why:** If your registry is exposed enough for your cluster to pull images from it, then it's exposed enough to need authentication.

**How:** The Kubernetes website has a good walkthrough on configuring ImagePullSecrets, which uses Docker as an example registry, [here](#).

## Organizing Your Cluster

Microservices by nature are a messy business. A lot of the benefit of using microservices comes from enforcing separation of duties at a service level, effectively creating abstractions for the various components of your backend. Some good examples are running a database separate from business logic, running separate development and production versions of software, or separating out horizontally scalable processes.

The dark side of having different services performing different duties is that they cannot be treated as equals. Thankfully Kubernetes gives you many tools to deal with this problem.

### 4. Isolate environments by using Namespaces

**What:** Namespaces are the most basic and most powerful grouping mechanism in Kubernetes. They work almost like virtual clusters. Most objects in Kubernetes are, by default, limited to affecting a single namespace at a time.

**Why:** Most objects are namespace scoped, so you'll have to use namespaces. Given that they provide strong isolation, they are perfect for isolating environments with different purposes, such as user serving production environments and those used strictly for testing, or to separate different service stacks that support a single application, like for instance keeping your security solution's workloads separate from your own applications. A good rule of thumb is to divide namespaces by resource allocation: If two sets of microservices will require different resource pools, place them in separate namespaces.

**How:** It's part of the metadata of most object types:

```
apiVersion: v1
kind: Deployment
metadata:
  name: example-pod
  namespace: app-pod
...
```

Note that you should always create your own namespaces instead of relying on the 'default' namespace. Kubernetes' defaults typically optimize for the lowest amount of friction for developers, and this often means forgoing even the most basic security measures.



## 5. Organize your clusters with Labels

**What:** Labels are the most basic and extensible way to organize your cluster. They allow you to create arbitrary key:value pairs that separate your Kubernetes objects. For instance, you might create a label key which separates services that handle sensitive information from those that do not.

**Why:** As mentioned, Kubernetes uses labels for organization, but, more specifically, they are used for selection. This means, when you want to give a Kubernetes object a reference to a group of objects in some namespace, like telling a network policy which services are allowed to communicate with each other, you use their labels. Since they represent such an open-ended type of organization, do your best to keep things simple, and only create labels where you require the power of selection.

**How:** Labels are a simple spec field you can add to your YAML files:

```
apiVersion: v1
kind: Deployment
metadata:
  name: example-pod
  ...
  matchLabels:
    userexposed: true
    storespii: true
```

## 6. Use Annotations to track important system changes, etc.

**What:** Annotations are arbitrary key-value metadata you can attach to your pods, much like labels. However, Kubernetes does not read or handle annotations, so the rules around what you can and cannot annotate a pod with are fairly liberal, and they can't be used for selection.

**Why:** They help you track certain important features of your containerized applications, like version numbers or dates and times of first bring up. Annotations, in the context of Kubernetes alone, are a fairly powerless construct, but they can be an asset to your developers and operations teams when used to track important system changes.

**How:** Annotations are a spec field similar to labels.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  ...
  annotations:
    version: four
    launchdate: tuesday
```





## Securing Your Cluster

Alright, you've got a cluster set up and organized the way you want - now what? Well, next thing is getting some security in place. You could spend your whole lifetime studying and still not discover all the ways someone can break into your systems. A blog post has a lot less room for content than a lifetime, so you'll have to settle for a couple of strong suggestions.

### 7. Implement access control using RBAC

**What:** RBAC (Role Based Access Control) allows you to control who can view or modify different aspects of your cluster.

**Why:** If you want to follow the principle of least privilege, then you need to have RBAC set up to limit what your cluster users, and your deployments, are able to do.

**How:** If you're setting up your own cluster (i.e., not using a managed Kube service), make sure you are using "--authorization-mode=Node,RBAC" to launch your kube apiserver. If you are using a managed Kubernetes instance, you can check that it is set up to use RBAC by querying the command used to start the kube apiserver. The only generic way to check is to look for "--authorization-mode..." in the output of `kubectl cluster-info dump`.

Once RBAC is turned on, you'll need to change the default permissions to suit your needs. The Kubernetes project site provides a walk-through on setting up Roles and RoleBindings (<https://kubernetes.io/docs/reference/access-authn-authz/rbac/>). Managed Kubernetes services require custom steps for enabling RBAC - check out Google's guide for GKE (<https://cloud.google.com/kubernetes-engine/docs/how-to/role-based-access-control>) or Amazon's instructions for AKS (<https://docs.microsoft.com/en-us/azure/aks/concepts-identity#role-based-access-controls-rbac>).

### 8. Prevent risky behavior using Pod Security Policies

**What:** Pod Security Policies are a resource, much like a Deployment or a Role, and can be created and updated through kubectl in same way. Each holds a collection of flags you can use to prevent specific unsafe behaviors in your cluster.

**Why:** If the people who created Kubernetes thought limiting these behaviors was important enough to create a special object to handle it, then they are likely important.

**How:** Getting them working can be an exercise in frustration. I recommend getting RBAC up and running, then check out the guide from the Kubernetes project (<https://kubernetes.io/docs/concepts/policy/pod-security-policy/>). The most important to use, in my opinion, are preventing privileged containers (<https://kubernetes.io/docs/concepts/policy/pod-security-policy/#privileged>), and write access to the host file system (<https://kubernetes.io/docs/concepts/policy/pod-security-policy/#volumes-and-file-systems>), as these represent some of the leakier parts of the container abstraction.

### 9. Implement network control/firewalling using Network Policies

**What:** Network policies are objects that allow you to explicitly state which traffic is permitted, and Kubernetes will block all other non-conforming traffic.

**Why:** Limiting network traffic in your cluster is a basic and important security measure. Kubernetes by default enables open communication between all services. Leaving this "default open" configuration in place means an Internet-connected service is just one hop away from a database storing sensitive information.



**How:** Here's a great write-up published at Cloud Native Computing Foundation that will get you started: <https://www.cncf.io/blog/2019/04/19/setting-up-kubernetes-network-policies-a-detailed-guide/>.

## 10. Use Secrets to store and manage necessary sensitive information

**What:** Secrets are how you store sensitive data in Kubernetes, including passwords, certificates, and tokens.

**Why:** Your services may need to authenticate one another, other third-party services, or your users, whether you're implementing TLS or restricting access.

**How:** The Kubernetes project offers a guide (<https://kubernetes.io/docs/concepts/configuration/secret/>). One key piece of advice: avoid loading secrets as environment variables, since having secret data in your environment is a general security no-no. Instead, mount secrets into read only volumes in your container - you can find an example in this Using Secrets write up (<https://kubernetes.io/docs/concepts/configuration/secret/#using-secrets>).

## 11. Use an image scanner to identify and remediate image vulnerabilities

**What:** Scanners inspect the components installed in your images. Everything from the OS to your application stack. Scanners are super useful for finding out what vulnerabilities exist in the versions of software your image contains.

**Why:** Vulnerabilities are discovered in popular open source packages all the time. Some notable examples are Heartbleed and Shellshock. You'll want to know where such vulnerabilities reside in your system, so you know what images may need updating.

**How:** Scanners are a fairly common bit of infrastructure - most cloud providers have an offering. If you want to host something yourself, the open source Clair project is a popular choice.

## Keeping Your Cluster Stable

Kubernetes represents a tall stack. You have your applications, running on baked-in kernels, running in VMs (or on bare metal in some cases), accompanied by Kubernetes' own services sharing hardware. Given all these elements, plenty of things can go wrong, both in the physical and virtual realms, so it is very important to de-risk your development cycle wherever possible. The ecosystem around Kubernetes has developed a great set of best practices to keep things in line as much as possible.

## 12. Follow CI/CD methodologies

**What:** Continuous Integration/Continuous Deployment is a process philosophy. It is the belief that every modification committed to your codebase should add incremental value and be production ready. So, if something in your codebase changes, you probably want to launch a new version of your service, either to run tests or to update your exposed instances.

**Why:** Following CI/CD helps your engineering team keep quality in mind in their day-to-day work. If something breaks, fixing it becomes an immediate priority for the whole team, because every change thereafter, relying on the broken commit, will also be broken.



**How:** Thanks to the rise of cloud deployed software, CI/CD is in vogue. As a result, you can choose from tons of great offerings, from managed to self-hosted. If you're a small team, I recommend going the managed route, as the time and effort you save is definitely worth the extra cost.

### 13. Use Canary methodologies for rolling out updates

**What:** Canary is a way of bringing service changes from a commit in your codebase to your users. You bring up a new instance running your latest version, and you migrate your users to the new instance slowly, gaining confidence in your updates over time, as opposed to swapping over all at once.

**Why:** No matter how extensive your unit and integration tests are, they can never completely simulate running in production - there's always the chance something will not function as intended. Using canary limits your users' exposure to these issues.

**How:** Kubernetes, as extensible as it is, provides many routes to incrementally roll out service updates. The most straightforward approach is to create a separate deployment that shares a load balancer with currently running instances. The idea is you scale up the new deployment while scaling down the old until all running instances are of the new version.

### 14. Implement monitoring and integrate it with SIEM

**What:** Monitoring means tracking and recording what your services are doing.

**Why:** Let's face it - no matter how great your developers are, no matter how hard your security gurus furrow their brows and mash keys, things will go wrong. When they do, you're going to want to know what happened to ensure you don't make the same mistake twice.

**How:** There are two steps to successfully monitor a service - the code needs to be instrumented, and the output of that instrumentation needs to be fed somewhere for storage, retrieval, and analysis. How you perform instrumentation is largely dependent on your toolchain, but a quick web search should give you somewhere to start. As far as storing the output goes, I recommend using a managed SIEM (like Splunk (<https://www.splunk.com/>) or Sumo Logic (<https://www.sumologic.com/>)) unless you have specialized knowledge or need - in my experience, DIY is always 10X the time and effort you expect when it comes to anything storage related.

## Advanced Topics

Once your clusters reach a certain size, you'll find enforcing all of your best practices manually becomes impossible, and the safety and stability of your systems will be challenged as a result. After you cross this threshold, consider the following topics:

### 15. Manage inter-service communication using a service mesh

**What:** Services meshes are a way to manage your inter-service communications, effectively creating a virtual network that you use when implementing your services.

**Why:** Using a service mesh can alleviate some of the more tedious aspects of managing a cluster, such as ensuring communications are properly encrypted.



**How:** Depending on your choice of service mesh, getting up and running can vary wildly in complexity. Istio seems to be gaining momentum as the most used service mesh, and your configuration process will largely depend on your workloads.

A word of warning: If you expect to need a service mesh down the line, go through the agony of setting it up earlier rather than later - incrementally changing communication styles within a cluster can be a huge pain.

## **16. Use Admission Controllers to unlock advanced features in Kubernetes**

**What:** Admission controllers are a great catch-all tool for managing what's going into your cluster. They allow you to set up webhooks that Kubernetes will consult during bring up. They come in two flavors: Mutating and Validating. Mutating admission controllers alter the configuration of the deployment before it is launched. Validating admission controllers confer with your webhooks that a given deployment is allowed to be launched.

**Why:** Their use cases are broad and numerous - they provide a great way to iteratively improve your cluster's stability with home-grown logic and restrictions.

**How:** Check out this great guide on how to get started with Admission Controllers (<https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers/>).



# Top Nine Kubernetes Settings You Should Check Right Now to Optimize Security



While properly leveraging Kubernetes network policies, admission controllers, and RBAC are a key piece to ensuring the long-term security of your containerized applications, you have many other steps you can take immediately to improve the security posture of your Kubernetes deployments.

We've synthesized the nine Kubernetes settings you should audit right now to ensure your Kubernetes cluster is configured for maximum security. Some of these points were also noted in previous sections because of how critical they are to building a resilient Kubernetes environment.

## 1. Upgrade to the Latest Version

New security features – and not just bug fixes – are added to every quarterly update, and to take advantage of them, we recommend you run the latest stable version. The very best thing to do is to run the latest release with its most recent patches, but if you can't do that, at least run the last one or two versions. Upgrades and support can become more difficult the farther behind you fall, so plan to upgrade at least once per quarter. Using a managed Kubernetes provider can make upgrades very easy.

Check your version now:

```
$ kubectl version --shortt
Client Version: v1.8.6t
Server Version: v1.8.10-gke.0
```


## 2. Enable Role-Based Access Control (RBAC)

As discussed in earlier sections, Kubernetes RBAC is a critical security capability that should be used. The first step to using it is to enable it so you can control who can access the Kubernetes API and what permissions they have. RBAC is usually enabled by default in Kubernetes 1.6 and beyond (later for some managed providers), but if you have upgraded since then and haven't changed your configuration, you'll want to double-check your settings. Because of the way Kubernetes authorization controllers are combined, you must both enable RBAC and disable legacy Attribute-Based Access Control (ABAC).

If you are running in Google Container Engine, you can check this setting using `gcloud`:

```
$ gcloud container clusters list -
format='table[box] (name,legacyAbac.
enabled)'
```

NAME	ENABLED
with-rbac with-abac	True



Once RBAC is being enforced, you still need to use it effectively. Cluster-wide permissions should generally be avoided in favor of namespace-specific permissions. Avoid giving anyone cluster admin privileges, even for debugging – it is much more secure to grant access only as needed on a case-by-case basis.

You can explore the cluster roles and roles using `kubectl get clusterrolebinding` or `kubectl get rolebinding --all-namespaces`. Quickly check who is granted the special “cluster-admin” role; in this example, it’s just the “masters” group:

```
$ kubectl describe clusterrolebinding cluster-admin
Name:          cluster-admin
Labels:        kubernetes.io/bootstrapping=rbac-defaults
Annotations:   rbac.authorization.kubernetes.io/autoupdate=true
Role:
  Kind: ClusterRole
  Name: cluster-admin
Subjects:
  Kind  Name          Namespace
  ----  -
  Group  system:masters
```

If your application needs access to the Kubernetes API, create service accounts individually and give them the smallest set of permissions needed at each use site. This is better than granting overly broad permissions to the default account for a namespace.

Most applications don’t need to access the API at all; `automountServiceAccountToken` can be set to “false” for these.

### 3. Use Namespaces to Establish Security Boundaries

Creating separate namespaces is an important first level of isolation between components. We find it’s much easier to apply security controls such as Network Policies when different types of workloads are deployed in separate namespaces.

Is your team using namespaces effectively? Find out now by checking for any non-default namespaces:

```
kubectl get ns
NAME          STATUS    AGE
default       Active    16m
kube-public   Active    16m
kube-system   Active    16m
```

### 4. Separate Sensitive Workloads

To limit the potential impact of a compromise, it’s best to run sensitive workloads on a dedicated set of machines. This approach reduces the risk of a sensitive application being accessed through a less-secure application that shares a container runtime or host. For example, a compromised node’s kubelet credentials can usually access the contents of secrets only if they are mounted into pods scheduled on that node – if important secrets are scheduled onto many nodes throughout the cluster, an adversary will have more opportunities to steal them.

You can achieve this separation using node pools (in the cloud or on-premises) and Kubernetes namespaces, taints, tolerations, and other controls.



## 5. Secure Cloud Metadata Access

Sensitive metadata, such as kubelet admin credentials, can sometimes be stolen or misused to escalate privileges in a cluster. For example, a recent Shopify bug bounty disclosure detailed how a user was able to escalate privileges by confusing a microservice into leaking information from the cloud provider's metadata service. GKE's metadata concealment feature changes the cluster deployment mechanism to avoid this exposure, and we recommend using it until it is replaced with a permanent solution. Similar countermeasures may be needed in other environments.

## 6. Create and Define Cluster Network Policies

Network Policies allow you to control network access into and out of your containerized applications. To use them, you'll need to make sure that you have a networking provider that supports this resource; with some managed Kubernetes providers such as Google Kubernetes Engine (GKE), you'll need to opt in. (Enabling network policies in GKE will require a brief rolling upgrade if your cluster already exists.) Once that's in place, start with some basic default network policies, such as blocking traffic from other namespaces by default.

If you are running in Google Container Engine, you can check whether your clusters are running with policy support enabled:

```
$ gcloud container clusters list --  
format='table[box] (name,addonsConfig.networkPolicyConfig) '
```

NAME	NETWORK_POLICY_CONFIG
without-policy	{u'disabled': True}
with-policy	{}

In other environments, check that you're using one of the network providers that support policies. (<https://kubernetes.io/docs/tasks/administer-cluster/declare-network-policy/>)

Once you know policies are supported, check to see if you have any:

```
$ kubectl get networkpolicies --all-namespaces  
No resources found.
```

## 7. Run a Cluster-wide Pod Security Policy

A Pod Security Policy sets defaults for how workloads are allowed to run in your cluster. Consider defining a policy and enabling the Pod Security Policy admission controller – instructions vary depending on your cloud provider or deployment model. As a start, you could require that deployments drop the NET\_RAW capability to defeat certain classes of network spoofing attacks.





## 8. Harden Node Security

- Ensure the host is secure and configured correctly. One way to do so is to check your configuration against CIS Benchmarks; many products feature an autochecker that will assess conformance with these standards automatically.
- Control network access to sensitive ports. Make sure that your network blocks access to ports used by kubelet, including 10250 and 10255. Consider limiting access to the Kubernetes API server except from trusted networks. Malicious users have abused access to these ports to run cryptocurrency miners in clusters that are not configured to require authentication and authorization on the kubelet API server.
- Minimize administrative access to Kubernetes nodes. Access to the nodes in your cluster should generally be restricted – debugging and other tasks can usually be handled without direct access to the node.

## 9. Turn on Audit Logging

Make sure you have audit logs enabled and are monitoring them for anomalous or unwanted API calls, especially any authorization failures – these log entries will have a status message “Forbidden.” Authorization failures could mean that an attacker is trying to abuse stolen credentials. Managed Kubernetes providers, including GKE, provide access to this data in their cloud console and may allow you to set up alerts on authorization failures.

Following these recommendations is a first step towards a more secure Kubernetes cluster. We also recommend adhering to the Center for Internet Security Benchmarks for Kubernetes to further harden your environment (latest documentation for CIS k8s benchmarks can be found at <https://www.cisecurity.org/benchmark/kubernetes/>). As you improve the security of your tech stack, look for tools that provide a central point of governance for your container and Kubernetes deployments and deliver continuous monitoring and protection for your containers and cloud-native applications.

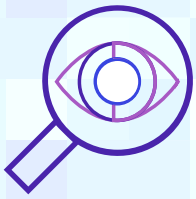


# How StackRox Secures Containers and Kubernetes



StackRox is the only Kubernetes-native container security platform purpose built to secure Kubernetes deployments and protect cloud-native applications across build, deploy, and runtime. DevOps and security teams choose the StackRox platform to address their most pressing security use cases including:

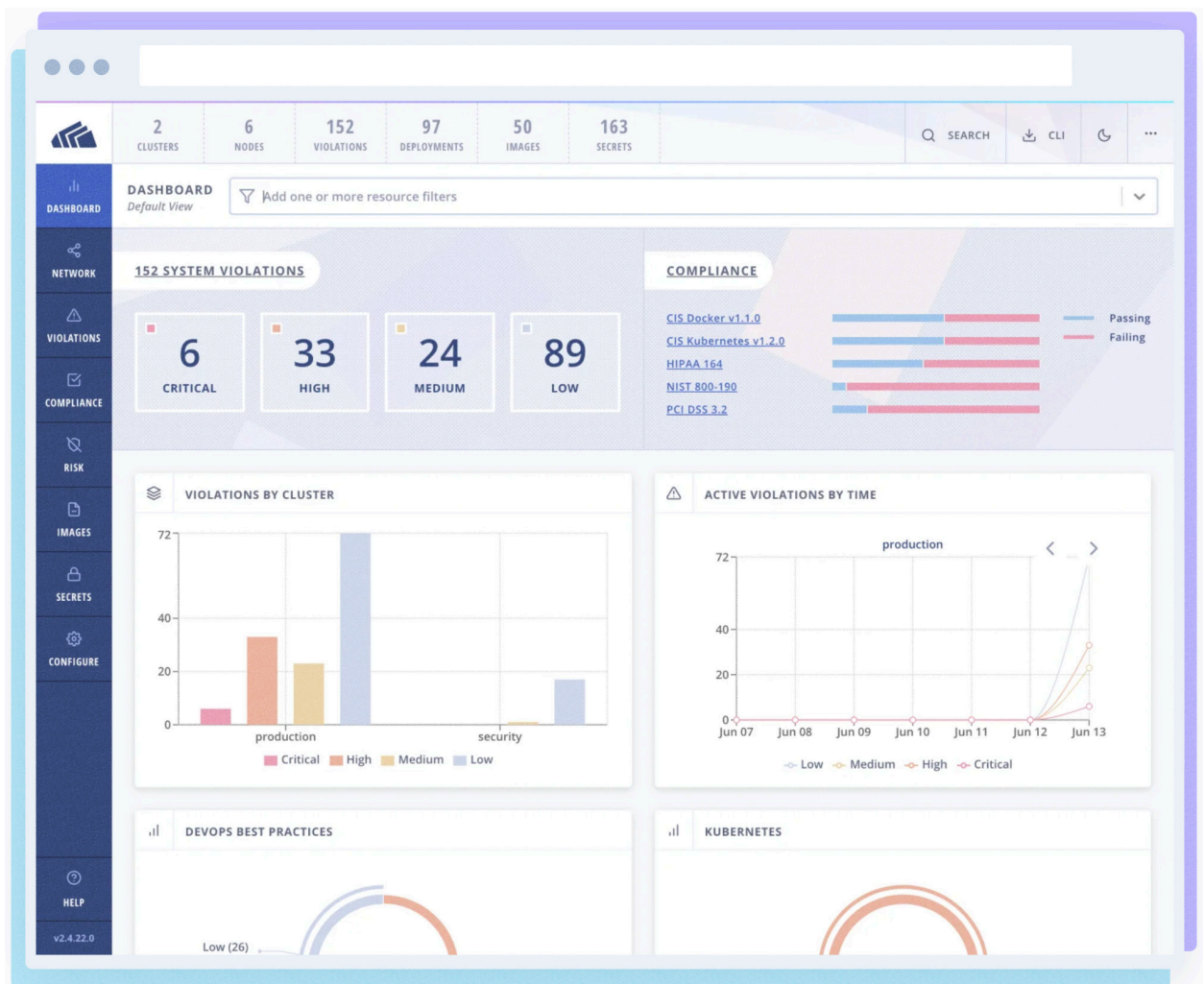
- **Visibility** sees your entire landscape of images, registries, containers, deployments, and runtime behavior
- **Vulnerability management** goes beyond vulnerability scoring to enforce configuration best practices at build, deploy, and runtime
- **Compliance** checks whether your systems meet standard-specific controls for CIS Benchmarks, NIST, PCI, and HIPAA
- **Network segmentation** leverages the native controls in Kubernetes to isolate assets, block deployments, or kill pods
- **Risk profiling** provides a stack-ranked list of all deployments, from most to least risky, with details on risk factors to identify highest priority fixes first and steps for remediation
- **Configuration management** applies best practices for Docker and Kubernetes to build your systems securely from the start
- **Threat detection** uses rules, whitelists, and baselining to accurately identify suspicious activity in your systems
- **Incident response** takes action, from alerting to killing pods to thwarting attacks, using the container and Kubernetes infrastructure for enforcement

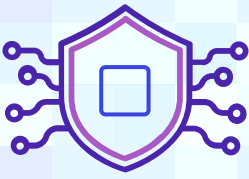


## Visibility

StackRox provides visibility into all your cloud-native infrastructure, including images, containers, pods, namespaces, deployments, and clusters. You get at-a-glance views of risk across your environment, compliance status, and active suspicious traffic. Each summary view enables you to drill into more detail. Organizations use StackRox to gain continuous visibility into their:

- network communications by analyzing active traffic between applications and simulate, preview, and enforce more secure network policy rules.
- on-going compliance posture across their infrastructure components against the controls in CIS Benchmarks for Docker and Kubernetes, NIST SP 800-190, PCI-DSS, and HIPAA.
- all running deployments, process executions, privilege escalation, and network communications





## Vulnerability Management

One of the most critical steps in securing containers and Kubernetes is to prevent images or containers with known high-risk vulnerabilities from being deployed as well as to identify and stop running containers that have vulnerabilities. StackRox ships with a built-in image scanner to ensure that images are free from vulnerabilities before they move into production, or our system can take in data from third-party scanners.

In addition, customers leverage StackRox to run on-demand vulnerability searches across images, running deployments, and clusters to enforce policies at build, deploy, and runtime. StackRox integrates with your CI/CD pipeline to fail a build if it contains an egregious vulnerability, while providing the developer details on why the build failed and how to remediate it. Alternatively, StackRox can block insecure deployments or scale them to zero if they have a vulnerability deemed critical.

The screenshot displays the StackRox Vulnerability Management interface. At the top, a summary bar shows 2 clusters, 6 nodes, 152 violations, 97 deployments, 50 images, and 163 secrets. The left sidebar contains navigation links for Dashboard, Network, Violations, Compliance, Risk, Images, Secrets, Configure, and Help. The main content area is titled 'IMAGES' and shows a list of 50 images. The table columns are Image, Created at, Components, CVEs, and Fixable CVE. The right sidebar shows a detailed view of the 'US.GCR.IO/ULTRA-CURRENT-825/STRUTS...' image, listing CVEs with their CVSS scores and whether they are fixed.

Image	Created at	Components	CVEs	Fixable CVE
us.gcr.io/ultra-current-825/struts-violations/visa-processor:latest	03/29/2019   1:33:22AM	223	428	221
us.gcr.io/ultra-current-825/struts-violations/mastercard-processor:latest	11/07/2018   8:45:33PM	223	428	221
us.gcr.io/ultra-current-825/struts-violations/backend-atlas:latest	03/29/2019   1:35:50AM	223	428	221
us.gcr.io/ultra-current-825/struts-violations/asset-cache:latest	07/12/2017   7:19:16AM	223	428	221
docker.io/library/wordpress:latest	06/12/2019   6:56:42PM	122	413	0
gcr.io/ultra-current-825/srox/monitor:latest	04/23/2019   12:27:14AM	37	256	154
gcr.io/ultra-current-825/srox/jump-host:latest	04/22/2019   11:30:02PM	28	127	47
gcr.io/ultra-current-825/srox/asset-cache:sidecar-latest	06/10/2019   6:18:41PM	32	109	3
gcr.io/ultra-current-825/srox/reporting:latest	04/23/2019   1:32:44PM	23	109	39
gcr.io/ultra-current-825/srox/backend-atlas:sidecar-latest	06/10/2019   6:18:45PM	22	73	3
gcr.io/ultra-current-825/srox/visa-processor:sidecar-latest	06/10/2019   6:20:51PM	22	73	3
docker.io/grafana/grafana:6.0.2	03/19/2019   6:42:21AM	79	59	0
k8s.gcr.io/fluend-gcp-scaler:0.5.1	03/12/2019   3:36:14AM	56	41	0

Name	Version	CVEs
openjdk-7	7u131-2.6.9-2-deb8u1	75
struts	2.3.12	33

CVE	CVSS	Fixed
<a href="#">CVE-2016-3082</a> - XSLTResult in Apache...	10	
<a href="#">CVE-2013-4316</a> - Apache Struts 2.0.0...	10	
<a href="#">CVE-2017-5638</a> - The Jakarta Multipa...	10	
<a href="#">CVE-2018-11776</a> - Apache Struts vers...	9.3	
<a href="#">CVE-2013-1966</a> - Apache Struts 2 bef...	9.3	
<a href="#">CVE-2013-1965</a> - Apache Struts Show...	9.3	
<a href="#">CVE-2016-3081</a> - Apache Struts 2.x b...	9.3	
<a href="#">CVE-2013-2251</a> - Apache Struts 2.0.0...	9.3	
<a href="#">CVE-2013-2135</a> - Apache Struts 2 bef...	9.3	
<a href="#">CVE-2013-2134</a> - Apache Struts 2 bef...	9.3	
<a href="#">CVE-2013-2115</a> - Apache Struts 2 bef...	9.3	
<a href="#">CVE-2016-0785</a> - Apache Struts 2.x b...	9	
<a href="#">CVE-2016-4461</a> - Apache Struts 2.x b...	9	
<a href="#">CVE-2017-12611</a> - In Apache Struts 2...	7.5	
<a href="#">CVE-2014-0113</a> - CookieInterceptor i...	7.5	
<a href="#">CVE-2014-0112</a> - ParametersIntercep...	7.5	
<a href="#">CVE-2016-4436</a> - Apache Struts 2 bef...	7.5	
<a href="#">CVE-2014-7809</a> - Apache Struts 2.0.0...	6.8	
<a href="#">CVE-2017-9805</a> - The REST Plugin in A...	6.8	
<a href="#">CVE-2016-3090</a> - The TextParseUtil.tr...	6.5	
<a href="#">CVE-2014-0116</a> - CookieInterceptor i...	5.8	



## Compliance

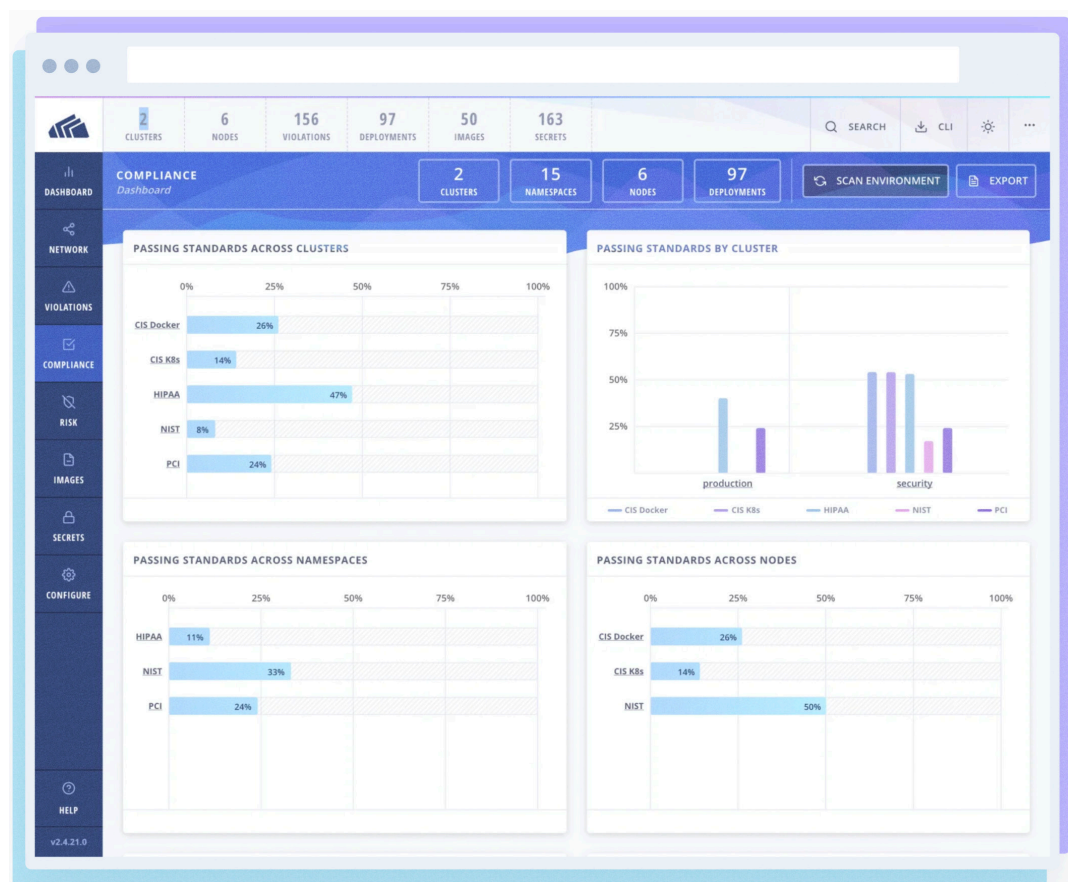
DevOps moves fast and relies on automation for continuous improvement; therefore, DevOps teams need a security solution built to complement, not inhibit, DevOps activities. The StackRox Kubernetes Security Platform is architected from the ground up to seamlessly integrate with DevOps workflows and methodologies.

In that vein, StackRox delivers automated and continuous checks that highlight where systems are failing to comply with controls defined in:

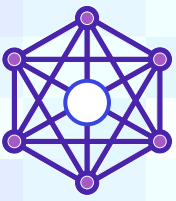
- CIS Benchmarks for Docker and Kubernetes
- National Institute of Standards and Technology (NIST) SP 800-190
- Payment Card Industry Data Security Standard (PCI DSS)
- Health Insurance Portability and Accountability Act (HIPAA)

Because customers not only need to adhere to compliance requirements but also show proof of their adherence, StackRox delivers on-demand exec summary reports that visually show the overall compliance status for each specification. Customers can also export detailed CSV files with a click of a button to give to auditors, with a list of each control relevant to containers and Kubernetes and details on the status of each compliance check.

Beyond adhering to industry standards, customers can also leverage the StackRox platform to define a set of internal policies for configurations and other best practices to prevent non-compliant builds or deployments from being pushed to production.





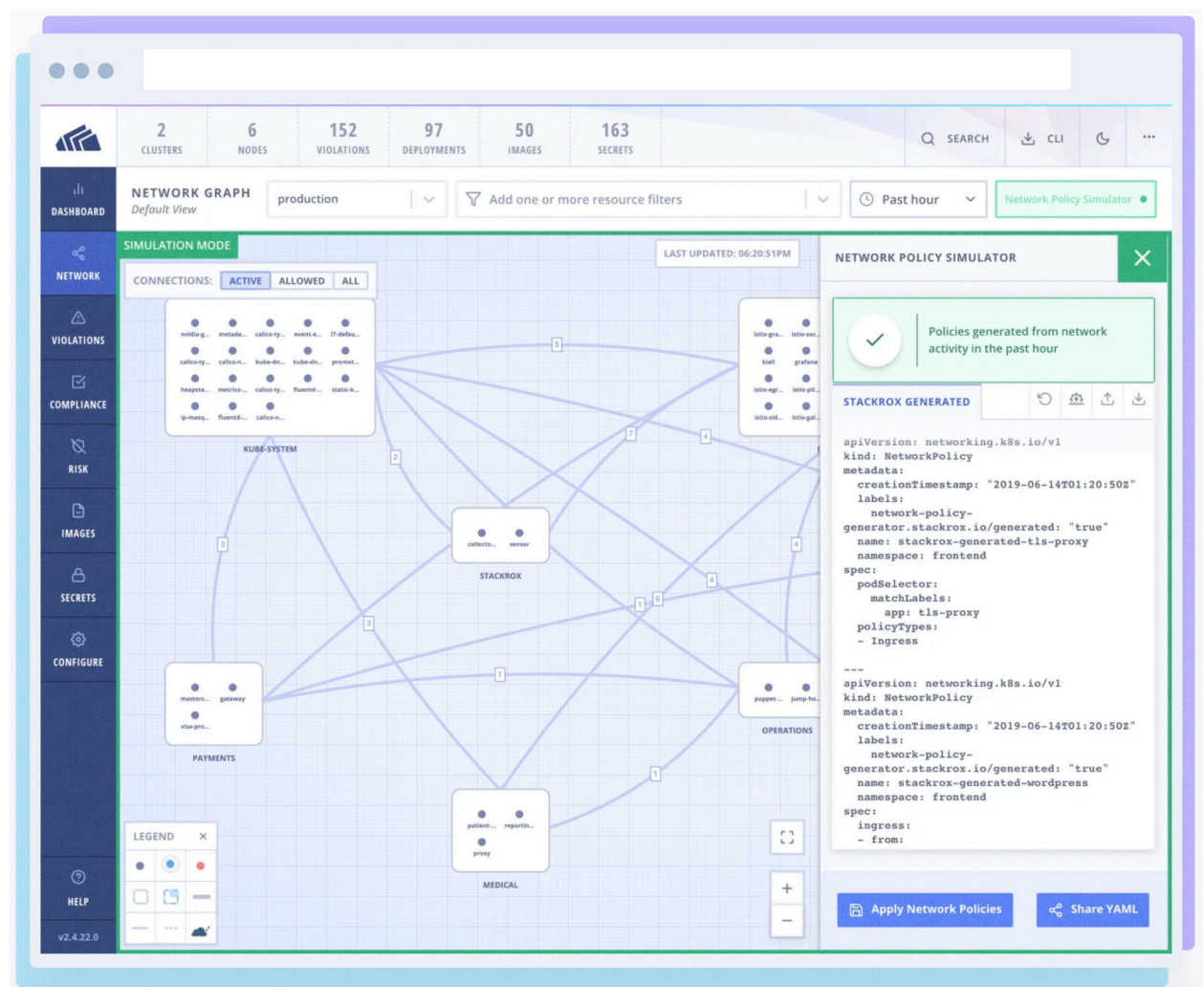


## Network Segmentation

As discussed in the previous network policies section, network policies essentially behave as firewall rules, specifying how groups of pods are allowed to communicate with each other and other network endpoints. By default, pods are non-isolated—meaning they allow all ingress and egress traffic from all sources—which is a highly insecure configuration.

As a first step, StackRox displays your network policies visually, showing allowed vs. active paths and highlights which nodes are not isolated and which are open to the Internet, providing you an instant view of your entire attack surface.

By analyzing the allowed vs. active communications paths, the StackRox platform can make recommendations on network segmentation policies that would reduce your risk. You can then use StackRox to simulate new policies that reduce the attack surface, generate an updated YAML file that instantiates that policy, and apply the YAML file directly to your Kubernetes instance or send it to your DevOps teams to apply.





## Risk Profiling

One common pain point our customers experience is being inundated with security alerts and incidents that need investigation without any guidance on prioritization. This approach inevitably leads to instances where high-risk security issues trail low/medium risk issues in remediation simply because teams cannot identify which problems present the highest risk. For example, dozens of containers might include the same vulnerability or misconfiguration, but one instance might be riskier because it's running in production vs. test, is open to the Internet, and serves a critical revenue-based application.

StackRox helps customers prioritize which of their Kubernetes deployments require immediate attention. Instead of just providing a long list of vulnerabilities or misconfigurations, StackRox helps you understand where misconfigurations or missed best practices increase the likelihood that your containers could be compromised or a vulnerability could be exploited.

StackRox accomplishes this prioritization by leveraging our deep integrations with Kubernetes, giving customers rich context for each deployment. We use that context to stack rank all the deployments from most to least risky, tying in details such as:

- vulnerability
- network exposure
- misconfigurations
- test vs. production environments
- access to secrets
- containers running in privileged mode
- suspicious processes execution

For example, a deployment that is affected by a vulnerability and has a wide network exposure will be ranked higher in risk than another deployment with the same vulnerability but with a smaller blast radius or no connection to the Internet.

The screenshot shows the StackRox Risk Dashboard. At the top, there are summary statistics: 2 CLUSTERS, 6 NODES, 152 VIOLATIONS, 97 DEPLOYMENTS, 50 IMAGES, and 163 SECRETS. Below this is a search bar and a filter button. The main section is titled 'RISK' and 'Default View'. It displays a table of 97 deployments, sorted by priority. The table has columns for Name, Updated, Cluster, Namespace, and Priority. The deployments are ranked from 1 to 13, with the highest risk at the top. To the right of the table, there is a 'VISA-PROCESSOR' panel with a 'RISK INDICATORS' tab. This panel lists various policy violations, such as 'Apache Struts: CVE-2017-5638 (severity: Critical)', 'Fixable CVSS >= 6 and Privileged (severity: High)', and 'Process with UID 0 (severity: High)'. Each violation is accompanied by a severity level and a brief description.

Name	Updated	Cluster	Namespace	Priority
visa-processor	06/13/2019   11:37:13AM	production	payments	1
asset-cache	06/13/2019   11:38:57AM	production	frontend	2
backend-atlas	06/13/2019   11:38:49AM	production	backend	3
mastercard-processor	06/13/2019   11:38:13AM	production	payments	4
jump-host	06/13/2019   11:39:10AM	production	operations	5
fluentd-gcp-v3.2.0	06/13/2019   11:30:53AM	production	kube-system	6
monitor	06/13/2019   11:30:19AM	production	frontend	7
fluentd-gcp-v3.2.0	06/13/2019   11:30:19AM	security	kube-system	8
reporting	06/13/2019   11:39:03AM	production	medical	9
wordpress	06/13/2019   11:38:57AM	production	frontend	10
puppet-master	06/13/2019   11:39:10AM	production	operations	11
sensor	06/13/2019   11:36:20AM	production	stackrox	12
collector	06/13/2019   11:36:11AM	security	stackrox	13





## Configuration Management

The configuration options for container and Kubernetes environments run deep and can often seem daunting to get right. In sprawling container and Kubernetes environments, it's impossible to manually check each security configuration for each asset to assess its risk. While the CIS Benchmarks for Docker and Kubernetes provide helpful guidance and a useful framework for hardening your environment, they contain hundreds of checks for different configuration settings. Ensuring continuous adherence to the CIS benchmarks can be challenging without an automated management layer.

As a container-native and Kubernetes-native platform, StackRox helps customers manage and secure the configuration of their container and Kubernetes environments.

StackRox not only identifies misconfigurations across images, containers, clusters, Kubernetes, RBAC settings, and network policies but also provides policy enforcement capabilities such as failing builds or blocking deployments that contain misconfigurations or don't follow best practices.

The screenshot displays the StackRox Risk Management dashboard. At the top, a navigation bar shows counts for 2 Clusters, 6 Nodes, 152 Violations, 97 Deployments, 50 Images, and 163 Secrets. A search bar and icons for search, CLI, and refresh are also present. The left sidebar contains a vertical menu with options: Dashboard, Network, Violations, Compliance, Risk (selected), Images, Secrets, Configure, and Help. The main content area is titled 'RISK' and shows 'Filtered View' with a search filter for 'Image: us.gcr.io/ultra-current-825/struts-violations/mastercard-processor:latest'. Below this, a table lists '1 DEPLOYMENT MATCHED' with columns for Name, Updated, Cluster, Namespace, and Priority. The table shows one entry: 'mastercard-processor' updated on '06/13/2019 | 11:38:13AM' in the 'production' cluster, namespace 'payments', with a priority of 4. To the right of the table, a detailed view for 'MASTERCARD-PROCESSOR' is shown, including sections for 'RISK INDICATORS' (listing 'Port 15090 is exposed in an unknown manner'), 'Components Useful for Attackers' (listing 'Image docker.io/istio/proxyv2:1.1.8 contains components useful for attackers: apt, bash, curl, netcat'), 'Number of Components in Image' (stating 'Image us.gcr.io/ultra-current-825/struts-violations/mastercard-processor:latest contains 323 components'), 'Image Freshness' (stating 'Deployment contains an image 217 days old'), and 'RBAC Configuration' (stating 'Deployment is configured to automatically mount a token for service account "default"' and 'Service account "default" is configured to mount a token into the deployment by default').



## Threat Detection

Once container images are built and deployed into production, they are exposed to new security challenges and a broad set of potential adversaries. The primary goal of security in the runtime phase is to detect and respond to malicious activity in an automated and scalable way while minimizing false positives and alert fatigue.

Customers use StackRox to identify and stop threats during runtime using rules, whitelists, baselines, and behavioral models. StackRox delivers out-of-the-box rules to detect:

- privilege escalation
- network reconnaissance
- package installation
- cryptocurrency mining
- modification of host configuration
- execution of reverse shell or other remote access tools
- data exfiltration

For enforcement, StackRox enables you to define a tiered response based on the severity of the detected threat, from alerting an admin, to scaling a service to zero, to killing a pod.

The screenshot displays the StackRox dashboard interface. At the top, a summary bar shows 2 clusters, 6 nodes, 152 violations, 97 deployments, 50 images, and 163 secrets. The left sidebar contains navigation links for Dashboard, Network, Violations, Compliance, Risk, Images, Secrets, Configure, and Help. The main content area is titled 'VIOLATIONS' and shows a list of 152 violations. The table columns are Deployment, Cluster, Namespace, Policy, Enforced, and Severity. The table lists various violations, including 'visa-processor' (High severity), 'asset-cache' (Low and High severity), and 'backend-atlas' (Low and High severity). A detailed view of a violation is shown on the right, titled 'VISA-PROCESSOR (42B545BA-8E0A-11E9-...)'. This view includes a 'VIOLATION' tab, 'ENFORCEMENT', 'DEPLOYMENT', and 'POLICY' sub-tabs. The 'VIOLATION' tab shows details about the violation, including the container ID, time, user ID, and arguments. The 'ENFORCEMENT' tab shows the enforcement action, including the container ID, time, user ID, and arguments. The 'DEPLOYMENT' tab shows the deployment details, including the container ID, time, user ID, and arguments. The 'POLICY' tab shows the policy details, including the container ID, time, user ID, and arguments.

Deployment	Cluster	Namespace	Policy	Enforced	Severity
<input type="checkbox"/> visa-processor	production	payments	<a href="#">Process with UID 0</a>	No	High
<input type="checkbox"/> visa-processor	production	payments	<a href="#">Ubuntu Package Manager Execution</a>	No	Low
<input type="checkbox"/> visa-processor	production	payments	<a href="#">Shell Spawned by Java Application</a>	No	High
<input type="checkbox"/> visa-processor	production	payments	<a href="#">Netcat Execution Detected</a>	No	Medium
<input type="checkbox"/> asset-cache	production	frontend	<a href="#">Ubuntu Package Manager Execution</a>	No	Low
<input type="checkbox"/> asset-cache	production	frontend	<a href="#">Shell Spawned by Java Application</a>	No	High
<input type="checkbox"/> asset-cache	production	frontend	<a href="#">Process with UID 0</a>	No	High
<input type="checkbox"/> backend-atlas	production	backend	<a href="#">Ubuntu Package Manager Execution</a>	No	Low
<input type="checkbox"/> backend-atlas	production	backend	<a href="#">Shell Spawned by Java Application</a>	No	High
<input type="checkbox"/> backend-atlas	production	backend	<a href="#">Process with UID 0</a>	No	High
<input type="checkbox"/> monitor	production	frontend	<a href="#">Process Targeting Kubernetes Service Endpoint</a>	No	High
<input type="checkbox"/> monitor	production	frontend	<a href="#">Process with UID 0</a>	No	High
<input type="checkbox"/> monitor	production	frontend	<a href="#">Iptables Executed in</a>	No	High

**VIOLATION**

Detected executions of 13 binaries with 13 different arguments with UID '0'

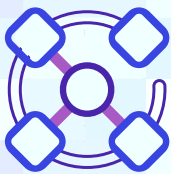
**First Occurrence:** 06/13/2019 | 11:39:01AM  
**Last Occurrence:** 06/13/2019 | 5:44:35PM

**Container ID:** 2311a3319596  
**Time:** 06/13/2019 | 11:39:01AM  
**User ID:** 0  
**Arguments:** /usr/local/tomcat/bin/catalina.sh run

**/bin/uname**  
**Container ID:** 2311a3319596  
**Time:** 06/13/2019 | 11:39:01AM  
**User ID:** 0  
**Arguments:**  
**Ancestors:** /usr/local/tomcat/bin/catalina.sh

**/usr/bin/dirname**  
**Container ID:** 2311a3319596  
**Time:** 06/13/2019 | 11:39:01AM  
**User ID:** 0  
**Arguments:** /usr/local/tomcat/bin/catalina.sh  
**Ancestors:** /usr/local/tomcat/bin/catalina.sh

**/usr/bin/tty**  
**Container ID:** 2311a3319596  
**Time:** 06/13/2019 | 11:39:01AM  
**User ID:** 0  
**Arguments:** /usr/local/tomcat/bin/catalina.sh  
**Ancestors:** /usr/local/tomcat/bin/catalina.sh



## Incident Response

The StackRox platform's incident response capabilities span the full container life cycle and can be tailored to both the phase of the life cycle (build, deploy, runtime) and the severity of the incident. In addition to fully automated responses, StackRox also supports forensic investigations by providing detailed information for each incident to understand context, such as suspicious files or processes launched.

StackRox can also be integrated with existing incident management and response tools such as a SIEM or a cloud provider's security services such as Google Cloud Security Command Center for incident aggregation and correlation.

Name	Description	Lifecycle	Severity
Linux User Add Execution	Detects when the 'useradd' or 'adduser' binary is executed, which can be used to add a new linux user.	Runtime	High
Linux Group Add Execution	Detects when the 'addgroup' or 'groupadd' binary is executed, which can be used to add a new linux group.	Runtime	High
Iptables Executed in Privileged Container	Alert on privileged pods that execute iptables	Runtime	High
Cryptocurrency Mining Process Execution	Cryptocurrency mining process spawned	Runtime	High
Compiler Tool Execution	Detects execution of binaries which are used to compile software	Runtime	Low
Alpine Linux Package Manager Execution	Alert on deployments with the Alpine Linux package manager (apk) is executed in runtime	Runtime	Low
Wget in Image	Alert on deployments with wget present	Deploy	Low
Ubuntu Package Manager in Image	Alert on deployments with components of the Debian/Ubuntu package management system in the image.	Deploy	Low
Secure Shell (ssh) Port Exposed in Image	Alert on deployments exposing port 22, commonly reserved for SSH access.	Deploy	High
Secure Shell (ssh) Port Exposed	Alert on deployments exposing port 22, commonly reserved for SSH access.	Deploy	High
Required Label: Owner	Alert on deployments missing the 'owner' label	Deploy	Low



## Tying it All Together

The native controls inherent in containers and Kubernetes offer the potential for building the most secure applications we've ever created. But getting all the knobs and dials set correctly can be daunting. A combination of following best practices outlined here and tapping into next-generation container security platforms that help enforce the guardrails automatically will help you establish the optimal security posture.

## References

- <https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/>
- [https://docs.okd.io/latest/architecture/additional\\_concepts/dynamic\\_admission\\_controllers.html](https://docs.okd.io/latest/architecture/additional_concepts/dynamic_admission_controllers.html)
- <https://kubernetes.io/blog/2018/01/extensible-admission-is-beta/>
- <https://medium.com/ibm-cloud/diving-into-kubernetes-mutatingadmissionwebhook-6ef3c5695f74>
- <https://github.com/kubernetes/kubernetes/blob/v1.10.0-beta.1/test/images/webhook/main.go>
- <https://github.com/istio/istio>
- <https://www.stackrox.com/post/2019/02/the-runc-vulnerability-a-deep-dive-on-protecting-yourself/>



StackRox helps enterprises secure their containers and Kubernetes environments at scale. The StackRox Kubernetes Security Platform enables security and DevOps teams to enforce their compliance and security policies across the entire container life cycle, from build to deploy to runtime. StackRox integrates with existing DevOps and security tools, enabling teams to quickly operationalize container and Kubernetes security. StackRox customers span cloud-native start-ups Global 2000 enterprises, and government agencies.