StackRox

Guide to Kubernetes Network Policies for Maximum Security

Table of Contents

Overview

The container orchestrator war is over, and Kubernetes has won. With companies large and small rapidly adopting the platform, security has emerged as an important concern – partly because of the learning curve inherent in understanding any new infrastructure, and partly because of recently announced vulnerabilities.

Kubernetes brings another security dynamic to the table – its defaults are 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.

How to Set Up Network Policies

We lay out here a step-by-step guide on how to set up network policies. The network policy spec is intricate, and it can be difficult to understand and use correctly. In this guide, we provide recommendations that significantly improve security. Users can easily implement these recommendations without needing to know the spec in detail.

A quick note: this guide focuses just on 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. We will discuss egress policies in detail and provide recommendations in a subsequent guide in this series.

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.

“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.

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: internet-access
spec:
   podSelector:
     matchlabels:
       networking/allow-internet-access: “true”
   policyTypes:
   - Ingress
   ingress:
   - {}
```

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.

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: allow-same-namespace
spec:
   podSelector: {}
   policyTypes:
   - Ingress
   ingress:
   - from:
     - podSelector: {}
```

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-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
```

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-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: {}
```

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.

Summary

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 as well as some handy network policy recipes.

In a future guide in this series, we will release an accompanying open-source tool that allows you to easily apply the recommendations in this article as well as some other best practices.

Here at StackRox, we've spent a lot of time thinking about how to operationalize network policies. The StackRox Kubernetes Security Platform automatically suggests and can generate network policies that enable just those communications paths your applications need. You can learn more about these capabilities in our network policy enforcement discussion.

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

FREE Security Assessment

Changes in the infrastructure of the cloud-native development stack, including containers and orchestrators such as Kubernetes, are changing the security landscape.

To help you understand the state of your container and Kubernetes security in your environment, StackRox offers a FREE risk assessment. StackRox synthesizes information across your various container platforms and tool sets and transforms them into actionable security insights.

The comprehensive report you'll get from this assessment will show you:

1. The overall security health of your clusters
2. Services deployed with high-risk combinations of vulnerabilities and misconfigurations
3. Security failures that may affect compliance with CIS, NIST, PCI, HIPAA
4. Vulnerabilities across your container attack surface
5. Configuration best practices for DevOps teams

AUDIT YOUR CONTAINER SECURITY AT
https://www.stackrox.com/assessment

StackRox

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.

LET'S GET STARTED

Request a demo today!
info@stackrox.com
+1 (650) 489-6769
www.stackrox.com