BLOG / MEMBER POST

Securing a Kubernetes pod with Regula and Open Policy Agent

By Becki Lee
March 24, 2022

Guest post originally published on ***Fugue's blog*** *by Becki Lee*

Fugue recently released Kubernetes support in **Regula**, our open source policy engine for checking infrastructure as code. Not only can Regula check your Terraform and CloudFormation files for security and compliance violations, it can now also check Kubernetes YAML manifests!

In this blog post, we'll demonstrate how to run Regula on a Kubernetes manifest to detect an insecure pod, and then we'll secure it. As a bonus, the final version of the manifest will be compliant with **CIS Kubernetes Benchmark v1.6.1**, a set of recommendations for securing Kubernetes environments.

Ready? Let's go!

Getting started

First, install Regula. Homebrew users can execute the following commands:

```
brew tap fugue/regula
brew install regula
```

You can alternatively install a **prebuilt binary for your platform** or, if you prefer, **run Regula with Docker**.

While writing this post, we used Regula v1.5.0.

Next, see our **GitHub gist** and select the Download ZIP button, then extract the files. There are two:

- **pod.yaml**: A noncompliant, insecure Kubernetes manifest
- **pod-compliant.yaml**: A compliant, secured version of the same manifest

As we explain how to fix each violation, you can follow along at home by manually editing `pod.yaml`, or you can just refer to the secured `pod-compliant.yaml` version.

Reviewing the manifest

Our **manifest** declares a simple **pod** named hello with a single **BusyBox** container, also named hello:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello
spec:
  containers:
    - name: hello
      image: busybox
      command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
```

It doesn't *look* insecure, right? But let's run Regula to be sure.

Running Regula

In your terminal, cd into the directory containing the files you downloaded. We're going to run Regula on **pod.yaml** with the `--format compact` flag to save space:

```
regula run pod.yaml --format compact
```

We see the following output:

```
FG_R00484: Service account 'automountServiceAccountToken' should be set to 'false' [Medium]
  [1]: Pod.default.hello in pod.yaml
FG_R00490: Pods should not run containers as the root user [Medium]
  [1]: Pod.default.hello in pod.yaml
FG_R00491: Pods should not run containers with the NET_RAW capability [Medium]
  [1]: Pod.default.hello in pod.yaml
FG_R00493: Pods should not run containers with default capabilities assigned [Medium]
  [1]: Pod.default.hello in pod.yaml
FG_R00495: Pod seccomp profile should be set to 'docker/default' [Medium]
  [1]: Pod.default.hello in pod.yaml
FG_R00496: Pods and containers should apply a security context [Medium]
  [1]: Pod.default.hello in pod.yaml
Found 6 problems.
```

Yikes! Our pod is definitely not as secure as it could be. Regula has found 6 problems.

Not to worry – we can fix them all! Let's take a look at each violation so we can remediate it.

Remediating the violations

Service account tokens

The violation: "Service account 'automountServiceAccountToken' should be set to 'false' [Medium]"

The CIS Kubernetes v1.6.1 control: 5.1.6, "Ensure that Service Account Tokens are only mounted where necessary"

Why it matters: **Service account tokens** are used to authenticate requests from in-cluster processes to the Kubernetes API server. By default, service account tokens are auto-mounted in all pods. However, if a bad actor is able to compromise a single pod, they could potentially use its service account token to launch a privilege escalation attack and gain control of the entire cluster.

So, if a workload doesn't need to communicate with the API server, it's best to avoid auto-mounting a service account token. This is in accordance with the security principle of **least privilege**.

How to fix it: Set `automountServiceAccountToken` to `false` in the pod spec:

```
spec:
  automountServiceAccountToken: false
```

See **line 10** in `pod-compliant.yaml`.

Root user

The violation: "Pods should not run containers as the root user [Medium]"

The CIS Kubernetes v1.6.1 control: 5.2.6, "Minimize the admission of root containers"

Why it matters: Running as root when unnecessary is almost universally a bad idea. If a container runs as root, an attacker can gain root privileges to the host system in the event of a container breakout.

How to fix it: Set runAsUser to any non-zero user ID in the pod spec, since 0 is root:

```
spec:
  securityContext:
    runAsUser: 1001
```

See **lines 8-9** in `pod-compliant.yaml`.

You will need to make sure the user specified here is defined in the Docker image. Often a non-root user is created by running the Linux useradd command in the Dockerfile and then the user ID is specified via the **USER command**.

Linux capabilities

The next two violations are closely related and, in our example, can be solved the same way.

The violation: "Pods should not run containers with the NET_RAW capability [Medium]"

The CIS Kubernetes v1.6.1 control: 5.2.7, "Minimize the admission of containers with the NET_RAW capability"

The violation: "Pods should not run containers with default capabilities assigned [Medium]"

The CIS Kubernetes v1.6.1 control: 5.2.9, "Minimize the admission of containers with capabilities assigned"

Why it matters: When a Linux container runs, it is granted a default set of capabilities, which grant specific root privileges to processes. The NET_RAW capability is especially dangerous, because an attacker can use it to spy on network traffic or generate IP traffic with spoofed addresses.

Many services don't need all of the default capabilities, so it's a good idea to drop them all first and then add the required ones back in. In this example, we're not adding any back, but you could do so with `add: ["FOO"]`.

How to fix it: Set a securityContext for the container and specify capabilities with drop: ["ALL"] to remediate both violations at once (or drop: ["NET_RAW"] to remediate only 5.2.7, if desired):

```
spec:
  containers:
    - name: hello
      securityContext:
        capabilities:
          drop: ["ALL"]
```

See **lines 15-17** in `pod-compliant.yaml`.

Seccomp profile

The violation: "Pod seccomp profile should be set to 'docker/default' [Medium]"

The CIS Kubernetes v1.6.1 control: 5.7.2, "Ensure that the seccomp profile is set to docker/default in your pod definitions"

Why it matters: In Linux, Secure Computing Mode `(seccomp)` restricts which system calls (syscalls) are allowed. Container runtimes such as Docker typically provide a default **seccomp profile** that disables a number of syscalls, improving security.

Note that the `docker/default` profile was **deprecated in Kubernetes 1.11**, so despite what CIS Kubernetes v1.6.1 says, it's better to use `runtime/default` instead.

How to fix it: There are a few ways to accomplish this depending on your Kubernetes version, but in pre-v1.19 versions, you can do so with the `seccomp.security.alpha.kubernetes.io/pod` **annotation** in the pod metadata:

```
metadata:
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: "runtime/default"
```

See **lines 5-6** in pod-compliant.yaml.

Security context

The violation: "Pods and containers should apply a security context [Medium]"

The CIS Kubernetes v1.6.1 control: 5.7.3, "Apply Security Context to Your Pods and Containers"

Why it matters: A **security context** for a pod or container defines a variety of security settings related to access control, Linux capabilities, and privileges. It's wise to set the parameters that are relevant for your specific use case.

How to fix it: You can set a security context at the pod level or container level. If settings are defined at both levels and overlap, the container settings **override the pod settings**.

Earlier we set a `securityContext` at the pod level to prevent running as root, and at the container level to drop all capabilities, so we've already remediated this violation.

See **lines 8-9** and **15-17** in `pod-compliant.yaml`.

Running Regula again

Now that we've remediated all 6 of the violations, let's see what Regula has to say about our newly secured pod. Here's the updated manifest, which we've conveniently provided for you as **pod-compliant.yaml**:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: "runtime/default"
spec:
  securityContext:
    runAsUser: 1001
  automountServiceAccountToken: false
  containers:
    - name: hello
      image: busybox
      command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
      securityContext:
        capabilities:
          drop: ["ALL"]
```

If you've been editing `pod.yaml` along the way, run the same command you ran earlier:

```
regula run pod.yaml --format compact
```

If you haven't been editing `pod.yaml` (no judgement!), you can just run Regula on the `pod-compliant.yaml` manifest:

```
regula run pod-compliant.yaml --format compact
```

And we see this output:

```
No problems found. Nailed it.
```

Ta-dah! We've successfully remediated all of the problems Regula found with our insecure pod. Nice work!

What's next?

Now that you've learned how to use Regula to secure a Kubernetes manifest, read up on some additional **best practices for Kubernetes security**.

You can learn more about Regula at **regula.dev**. There, you'll find a list of all our **Kubernetes rules**. You can also learn how to **waive a rule result** or even **disable a rule altogether** if it isn't relevant to your organization.

By the way, Fugue enables you to secure the entire development lifecycle by applying the same policies to infrastructure as code and runtime resources! Learn more about the **Fugue Platform** and how it can help you move faster in the cloud—without breaking the rules.

SHARE

Other posts to check out

KubeCon + CloudNativeCon 2024 recap: highlights and takeaways

April 19, 2024

Cloud Custodian completes audit to strengthen security posture and enable continuous assessment

April 19, 2024

The challenges of rising MTTR — and what to do

April 18, 2024