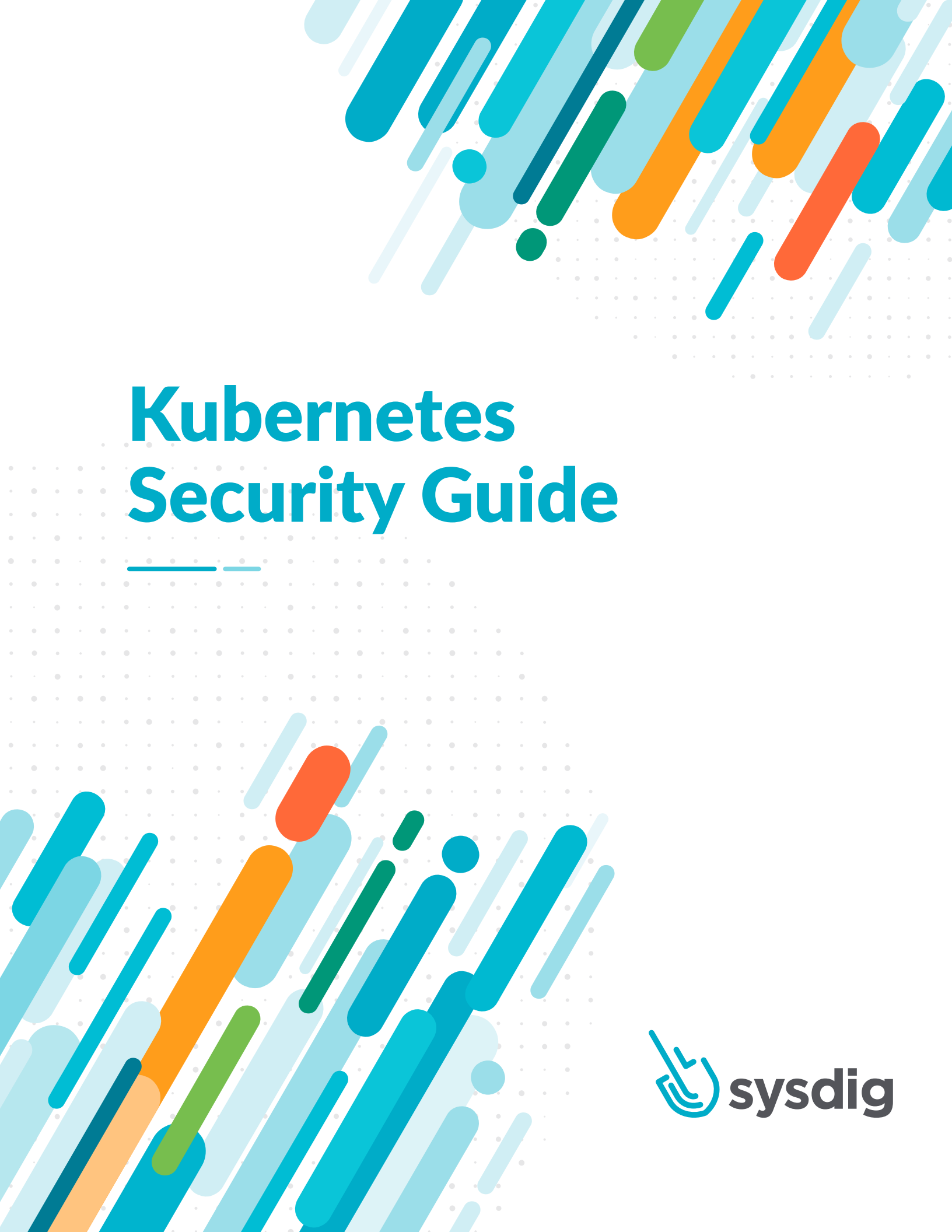
Kubernetes Security Guide



Contents

Intro

Kubernetes has become the de facto operating system of the cloud. This rapid success is understandable, as Kubernetes makes it easy for developers to package their applications into portable microservices. However, Kubernetes can be challenging to operate. Teams often put off addressing security processes until they are ready to deploy code into production.

Kubernetes requires a new approach to security. After all, legacy tools and processes fall short of meeting cloud-native requirements by failing to provide visibility into dynamic container environments. Fifty-four percent of containers live for five minutes or less, which makes investigating anomalous behavior and breaches extremely challenging.

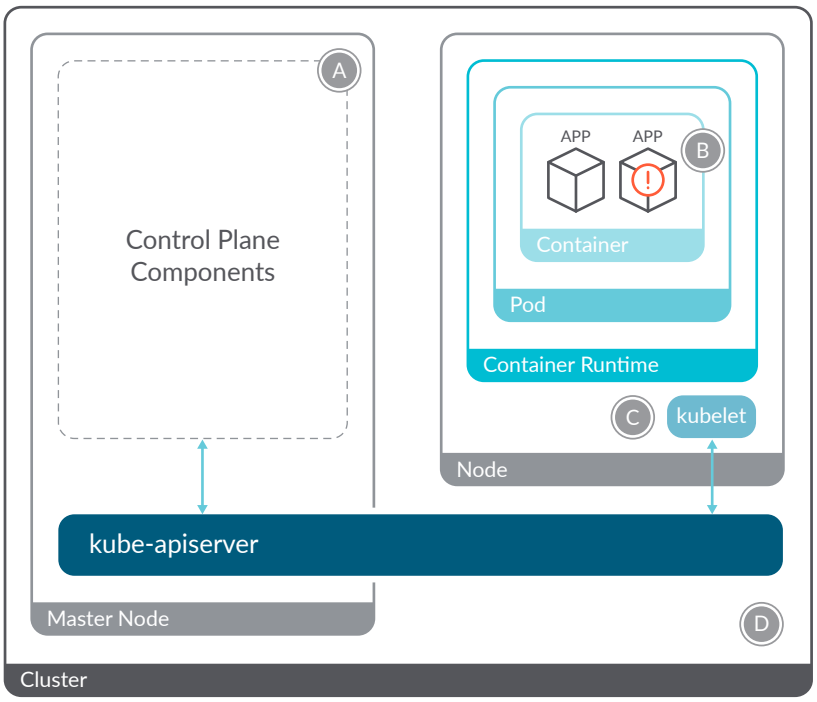
One of the key points of cloud-native security is addressing container security risks as soon as possible. Doing it later in the development life cycle slows down the pace of cloud adoption, while raising security and compliance risks.

The **Cloud/DevOps/DevSecOps** teams are typically responsible for security and compliance as critical cloud applications move to production. This adds to their already busy schedule to keep the cloud infrastructure and application health in good shape.

We've compiled this security guide to provide guidance on choosing your approach to security as you ramp up the use of containers and Kubernetes.

Kubernetes attack surface

Let's first take a glance at a Kubernetes cluster to understand which elements you need to protect.



- A Access via Kubernetes API Proxy etcd API
- B Exploit vulnerability in apps or 3rd party libraries
- C Access via API
- D Access to the servers or virtual machines

The first area to protect is your **applications and libraries**. Vulnerabilities in your base OS images for your applications can be exploited to steal data, crash your servers or scale privileges. Another component you need to secure are third-party libraries. Often, attackers won't bother to search for vulnerabilities in your code because it's easier to use known exploits in your applications libraries.

The next vector is the **Kubernetes control plane** - your cluster brain. Programs like the controller manager, etcd or kubelet, can be accessed via the **Kubernetes API .** An attacker with access to the API could completely stop your server, deploy malicious containers or delete your entire cluster.

Additionally, your cluster runs on servers, so **access** to them needs to be protected. Undesired access to these servers, or the virtual machines where the nodes run, will enable an attacker to have access to all of your resources and the ability to create serious security exposures.

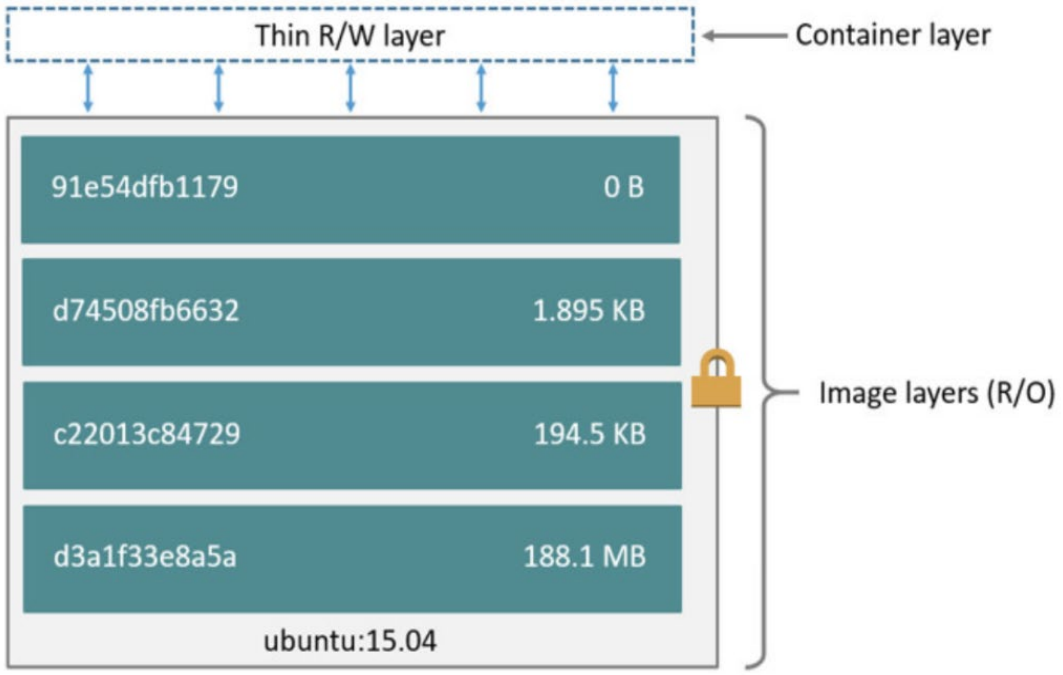
CHAPTER 1

Securing your container images and CI/CD pipeline

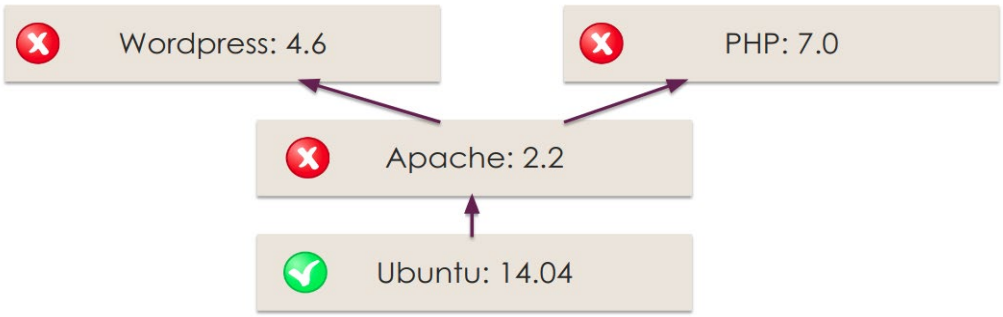
One of the final steps of the CI (Continuous Integration) pipeline involves building the container images that will be pulled and executed in our environment. Therefore, whether you are building Docker images from your own code or using unmodified third party images, it's important to identify any known vulnerabilities that may be present in those images. This process is known as Docker vulnerability scanning.

Image scanning

Docker images are composed of several immutable layers, basically a diff over a base image that adds files and other changes. Each one is associated with a unique hash id:



Any new Docker image that you create will most likely be based on an existing image (FROM statement in the Dockerfile). That's why you can leverage this layered design to avoid having to re-scan the entire image every time you make a new one, with a small change. If a parent image is vulnerable, any other images built on top of it will be vulnerable too.



The Docker build process follows a manifest (Dockerfile) that includes relevant security information that you can scan and evaluate including the base images, exposed ports, environment variables, entrypoint script, external installed binaries and more. By the way, don't miss our Docker security best practices article for more hints in building your Dockerfiles.

In a secure pipeline, Docker vulnerability scanning should be a mandatory step of your CI/CD process, and any image should be scanned and approved before entering "Running" state in the production clusters.

What is image scanning

The Docker security scanning process typically includes:

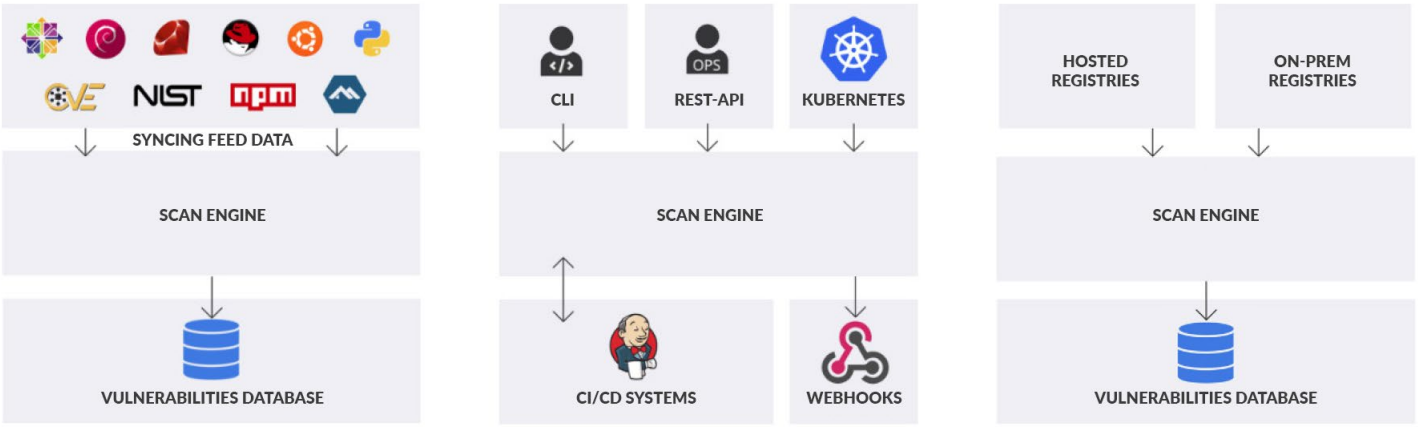
- Checking the software packages, binaries, libraries, operative system files and more against well known vulnerabilities databases. Some Docker scanning tools have a repository containing the scanning results for common Docker images. These tools can be used as a cache to speed up the process.
- Analyzing the Dockerfile and image metadata to detect security sensitive configurations like running as privileged (root) user, exposing insecure ports, using based images tagged with "latest" rather than specific versions for full traceability, user credentials, etc.
- User defined policies, or any set of requirements that you want to check for every image. This includes software packages blacklists, base images whitelists, whether a SUID file has been set, etc.

You can classify and group the different security issues you might find in an image by assigning different priorities: a warning notification is sufficient for some issues, while others will be severe enough to justify aborting the build.

Sysdig Secure Container image scanning: Scan engine

Sysdig Secure Scan Engine addresses Docker vulnerability scanning as part of the Secure DevOps methodology. DevOps teams can use it as a centralized service for inspection and analysis while applying user-defined acceptance policies to allow automated validation and certification of container images.

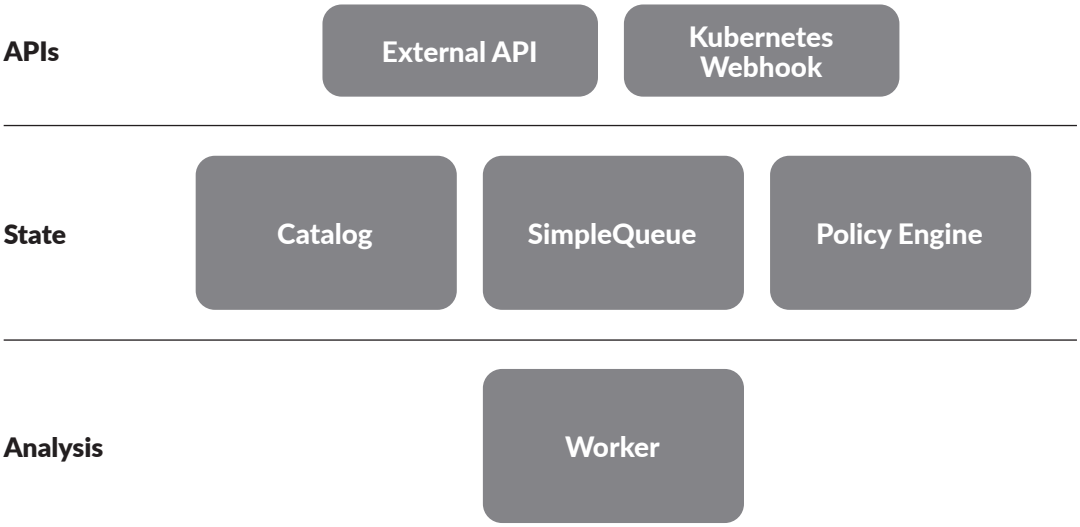
- Scan Engine allows developers to perform detailed analysis on their container
- images, run queries, produce reports and define policies that can be used in CI/CD
- pipelines.
- Using Scan Engine, container images can be downloaded from Docker V2 compatible container registries, as well as analyzed and evaluated against user defined policies.
- It can be accessed directly through a RESTful API or via CLI.
- The scanning includes not just CVE-based security scans but also policy-based scans that can include checks around security, compliance, and operational best practices.



Scan Engine sources and endpoints

Scan Engine architecture is comprised of six components that can either be deployed in a single container or scaled out:

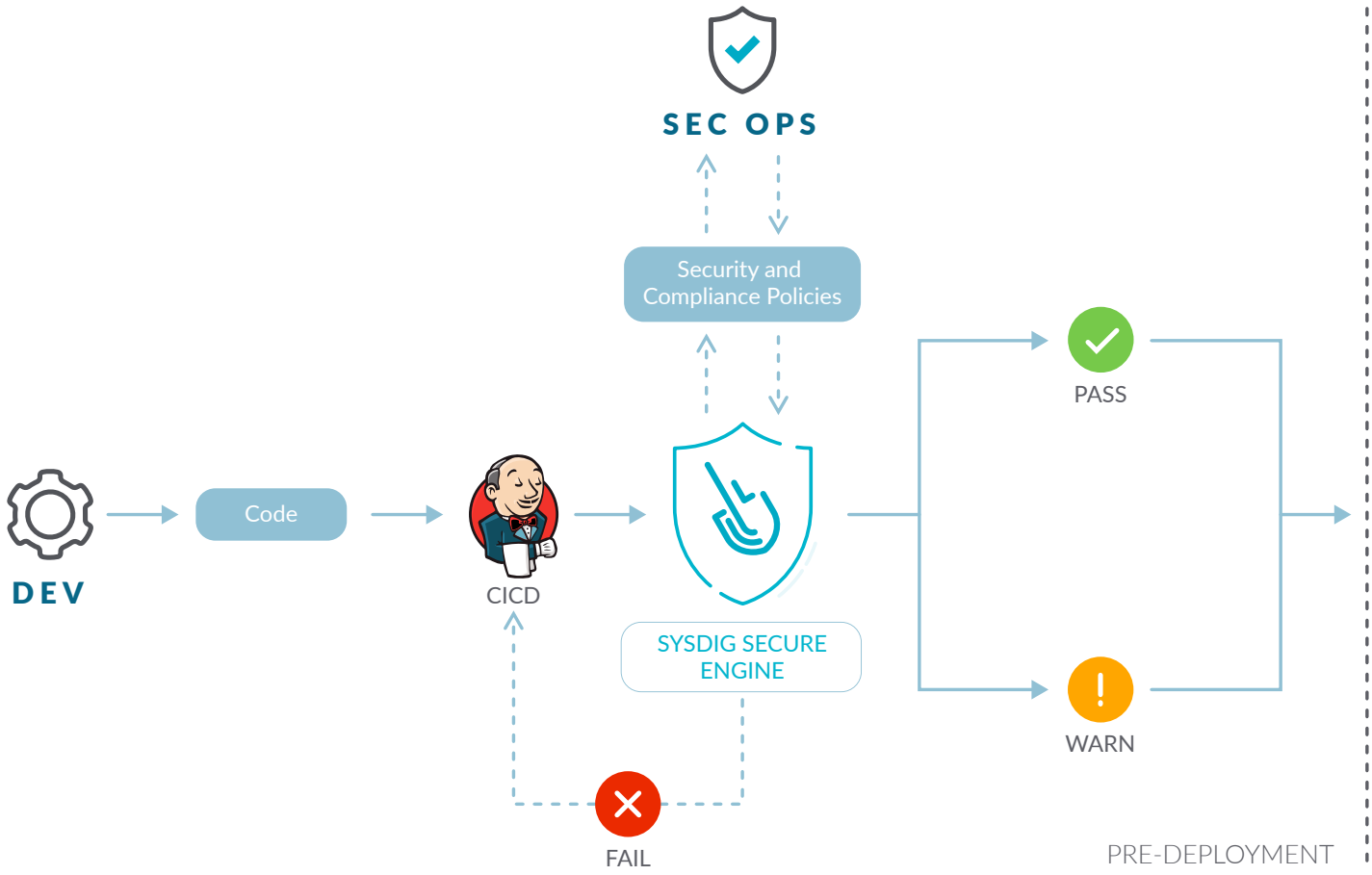
- API Service: Central communication interface that can be accessed by code, using a REST API, or directly, using the command line.
- Image Analyzer Service: Executed by the “worker”, these nodes perform the actual Docker image scanning.
- Catalog Service: Internal database and system state service.
- Queuing Service: Organizes, persists and schedules the engine tasks.
- Policy Engine Service: Policy evaluation and vulnerabilities matching rules.
- Kubernetes Webhook Service: Kubernetes-specific webhook service to validate images before they are spawned.



Scan Engine architecture

Securing your CI/CD pipeline

DevOps has introduced some interesting concepts in the world of software development. "You code it, you run it" means that there are no longer two separated teams for development and operations. This indicates that there is a strong decoupling between the applications and the infrastructure in which they run. CI/CD concepts allow developers to deploy very fast and often. But with these new tools come new challenges. Security teams have lost control over many aspects of the systems running the applications, as the containers are now the atomic unit of working. A container is opaque in many regards, and security teams are no longer responsible for what it has installed. As long as many aspects of development have moved left in the workflow, security has to be moved left as well, creating Secure DevOps: You code it, you run it, you secure it.



This new responsibility of DevOps teams means there is a need for new tools and procedures to establish the new security processes. It is not enough to adapt old security operations; there are new requirements that need new tools.

Image scanning in CI/CD

Secure DevOps teams need to ensure that the containers they're shipping are secure. The best way to do this is to include image scanning in the CI/CD pipelines. Some of the benefits of this are:

- Early detection of security issues. This allows **quicker responses**.
- If the issue is detected in the pipeline, the problem is much easier to fix.
- The problems are detected **before** deployments. This means the chances of outages due to security incidents are reduced by a significant percentage.

Like most things in IT, the earlier you detect container security issues, the easier they are to fix without further consequences.

Embedding container security in your build pipeline is a best practice for several reasons:

- The vulnerabilities will never reach your production clusters, or even worse, a client environment.
- You can adopt a secure-by-default approach by knowing any image available in your Docker container registry has already passed all of the security policies you have defined for your organization, as opposed to manually checking container and Kubernetes compliance after-the-fact.
- The original container builder will be (almost) instantly informed when the developer still has all the context. The issue will be substantially easier to fix, rather than if it was found by another person months later.

Inline scanning

Some security requirements restrict access to some registries in various environments in order to keep them safe. These access or permission restrictions can make image scanning tricky or even impossible. Inline scanning allows you to scan the images locally, at build time, without using any registry.

Metadata from the analysis can be uploaded to a database or security backend in order to store the information and use it for image deployment control.

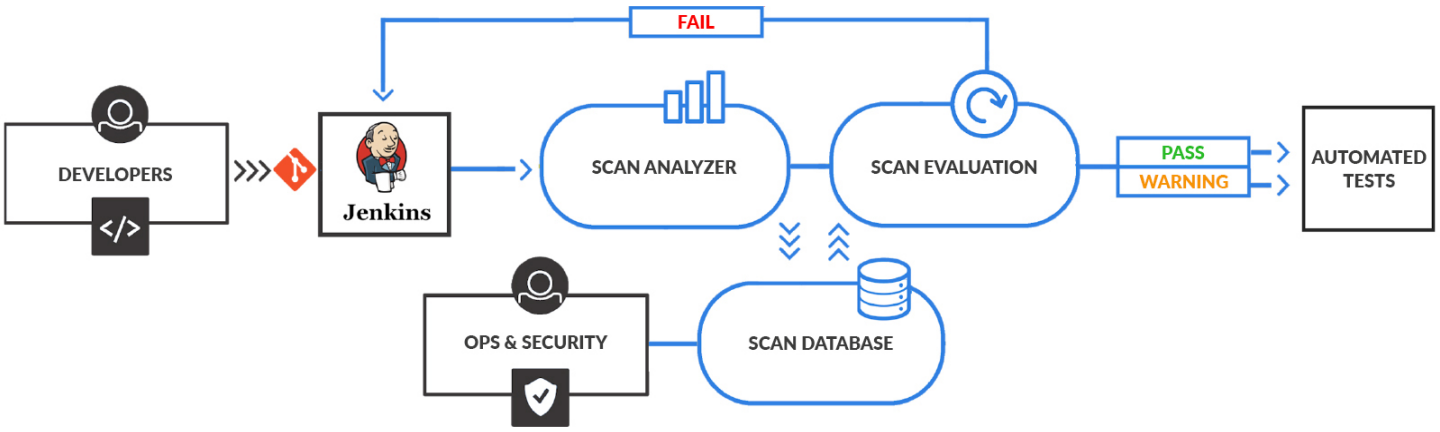
Inline image scanning provides several benefits over traditional image scanning within the registry:

- The analysis is performed inline (locally) on the runner. This means that in case vulnerabilities are found, you can prevent the image from being published at all.
- As verifications are done locally, the image contents are not sent anywhere for analysis, so any confidential information is kept under control without being exposed. During the analysis, only metadata information is extracted from the image.
- The metadata obtained from the analysis can be reevaluated later if new vulnerabilities are discovered or policies are modified, all without requiring a new scanning of the image.
- You can set policies to enforce and adhere to various container compliance standards (like NIST 800-190 and PCI) and provide checks for containers running in Kubernetes and Openshift.

Integrating image scanning with Jenkins

Jenkins is an open source automation server with a plugin ecosystem that supports the typical tools that are a part of your delivery pipelines. Jenkins helps to automate the CI/CD process. Scan Engine has been designed to plug seamlessly into a CI/CD pipeline; a developer commits code into the source code management system, like Git. This change triggers Jenkins to start a build which creates a container image, etc.

In a typical workflow, this container image is then run through various automated testing. If an image does not pass the Docker security scanning (doesn't meet the organization's requirements for security or compliance), then it doesn't make sense to invest the time required to perform automated tests on the image. A better approach is to "learn fast" by failing the build and returning the appropriate reports back to the developer to address the issues.



Docker scanner with Jenkins

You can use the "Sysdig Secure Container Image Scanner plugin" available in the official plugin list that you can access via the Jenkins interface.

You can further read how to integrate image scanning in Jenkins in this article.

Integrating image scanning with Azure pipelines

Azure DevOps gives teams tools like version control, reporting, project management, automated builds, lab management, testing and release management. Azure Pipelines automates the execution of CI/CD tasks, like building the container images when a commit is pushed to your git repository or performing vulnerability scanning on the container image.

Image scanning allows DevOps teams to shift left security, detecting known vulnerabilities and validating container build configuration early in their pipelines. This is done before the containers are deployed in production or images are pushed into any container registry. This allows you to detect and fix issues faster, improving delivery to production time.

You can find here detailed information on how to introduce image scanning in Azure pipelines.

Integrating image scanning with AWS CodePipeline and AWS CodeBuild

AWS provides several tools for DevOps teams: CodeCommit for version control, CodeBuild for building and testing code, and CodeDeploy for automatic code deployment. The block on top of all of these tools is CodePipeline, which allows them to visualize and automate these different stages.

Image Scanning for AWS CodePipeline raises the confidence that DevOps teams have in the security of their deployments, detecting known vulnerabilities and validating container build configuration early in their pipelines. By detecting those issues before the images are published into a container registry or deployed in production, fixes can be applied faster and delivery to production time improves.

You can learn more about how to use inline scanning with AWS CodePipeline in this article.

Integrating image scanning with Bamboo

Atlassian Bamboo is a continuous integration and delivery server integrated with Atlassian software development and collaboration platform. Some of the features that distinguish Bamboo from similar CI/CD tools are its native integration with other Atlassian products (like Jira project management and issue tracker), improved support for Git workflows (branching and merging) and flexible scalability of worker nodes using ephemeral Amazon EC2 virtual machines.

Learn more in the article: Integrating Sysdig Secure with Atlassian Bamboo CI/CD.

Integrating image scanning with Gitlab CI/CD

Gitlab CI/CD is an open source continuous integration and delivery server integrated with the Gitlab software development and collaboration platform.

Once you have configured Gitlab CI/CD for your repo, every time a developer pushes a commit to the tracked repository branches, the pipeline scripts will be automatically triggered.

You can use these pipelines to automate many processes. Common tasks include QA testing, building software distribution artifacts (like Docker images or linux packages) or, as is the case for this article, checking compliance with security policies.

Learn more in the article: Integrating Gitlab CI/CD with Sysdig Secure.

CHAPTER 2

Securing Kubernetes Control Plane

In addition to configuring the Kubernetes security features, a fundamental part of Kubernetes security is securing sensitive Kubernetes components such as kubelet and internal Kubernetes etcd. We also shouldn't forget the common external resources, like the Docker registry, that we pull images from. In this part, we will learn best practices on how to secure the Kubernetes kubelet and the Kubernetes etcd cluster, as well as how to configure a trusted Docker registry.

Kubelet security

The kubelet is a fundamental piece of any Kubernetes deployment. It's often described as the "Kubernetes agent" software, and is responsible for implementing the interface between the nodes and the cluster logic.

The main task of a kubelet is managing the local container engine (i.e. Docker) and ensuring that the pods described in the API are defined, created, run and remain healthy; and also that they are destroyed when appropriate.

There are two different communication interfaces to be considered:

- Access to the Kubelet REST API from users or software (typically just the Kubernetes API entity).
- Kubelet binary accessing the local Kubernetes node and Docker engine.

These two interfaces are secured by default using:

- Security related configuration (parameters) passed to the kubelet binary – Next section (Kubelet security – access to the kubelet API).
- NodeRestriction admission controller – See below Kubelet security – kubelet access to Kubernetes API.
- RBAC to access the kubelet API resources – See below RBAC example, accessing the kubelet API with curl.

Access to the kubelet API

The kubelet security configuration parameters are often passed as arguments to the binary exec. For newer Kubernetes versions (1.10+) you can also use a kubelet configuration file. Either way, the parameters syntax remains the same.

Let's use this example configuration as reference:

```
/home/kubernetes/bin/kubelet
-v=2
-kube-
reserved=cpu=70m,memory=1736Mi
-allow-privileged=true
-cgroup-root=/
-pod-manifest-path=/etc/kubernetes/manifests
-experimental-mounter-path=/home/kubernetes/containerized_mounter/
mounter
-experimental-check-node-capabilities-before-mount=true -cert-dir=/
var/lib/kubelet/pki/
-enable-debugging-handlers=true
-bootstrap-kubeconfig=/var/lib/kubelet/bootstrap-kubeconfig
-kubeconfig=/var/lib/kubelet/kubeconfig
-anonymous-auth=false
-authorization-mode=Webhook
-client-ca-file=/etc/srv/kubernetes/pki/ca-certificates.crt
-cni-bin-dir=/home/kubernetes/bin
-network-plugin=cni
-non-masquerade-cidr=0.0.0.0/0
-feature-gates=ExperimentalCriticalPodAnnotation=true
```

Verify the following Kubernetes security settings when configuring kubelet parameters:

- **`anonymous-auth`** is set to **`false`** to disable anonymous access (it will send 401 Unauthorized responses to unauthenticated requests).
- kubelet has a **`--client-ca-file flag`**, providing a CA bundle to verify client certificates.
- **`--authorization-mode`** is not set to **`AlwaysAllow`**, as the more secure Webhook mode will delegate authorization decisions to the Kubernetes API server.
- **`--read-only-port`** is set to 0 to avoid unauthorized connections to the read-only endpoint (optional).

Kubelet access to Kubernetes API

As we mentioned in the first part of this guide, the level of access granted to a kubelet is determined by the NodeRestriction Admission Controller (on RBAC-enabled versions of Kubernetes, stable in 1.8+).

kubelets are bound to the system:node Kubernetes clusterrole.

If NodeRestriction is enabled in your API, your kubelets will only be allowed to modify their own Node API object, and only modify Pod API objects that are bound to their node. It's just a static restriction for now.

You can check whether you have this admission controller from the Kubernetes nodes executing the apiserver binary:

```
$ ps aux | grep apiserver | grep admission-control
--admission-control=Initializers,NamespaceLifecycle,LimitRanger,Servi
ceAccount,PersistentVolumeLabel,DefaultStorageClass,DefaultToleration
Seconds,NodeRestriction,ResourceQuota
```

RBAC example, accessing the kubelet API with curl

Typically, only the Kubernetes API server will need to use the kubelet REST API. As we mentioned before, this interface needs to be protected as you can execute arbitrary pods and exec commands on the hosting node.

You can try to communicate directly with the kubelet API from the node shell:

```
# curl  -k https://localhost:10250/pods
Forbidden (user=system:anonymous, verb=get, resource=nodes,
subresource=proxy)
```

Kubelet uses RBAC for authorization and it's telling you that the default anonymous system account is not allowed to connect.

You need to impersonate the API server kubelet client to contact the secure port:

```
# curl --cacert /etc/kubernetes/pki/ca.crt --key /etc/kubernetes/pki/
apiserver-kubelet-client.key --cert /etc/kubernetes/pki/apiserver-
kubelet-client.crt -k https://localhost:10250/pods | jq .
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {},
  "items": [
    {
      "metadata": {
        "name": "kube-controller-manager-kubenode",
        "namespace": "kube-system",
...
```

Your port numbers may vary depending on your specific deployment method and initial configuration.

Kubernetes API audit and security log

Kube-apiserver provides a security-relevant chronological set of records documenting the sequence of activities that have affected the system by individual users, administrators or other components. It allows the cluster administrator to answer the following questions:

- What happened?
- When did it happen?
- Who initiated it?
- What was affected?
- Where was it observed?
- From where was it initiated?
- To where was it going?

The API audit output, if correctly filtered and indexed, can become an extremely useful resource for the forensics, early incident detection and traceability of your Kubernetes cluster.

The audit log uses the JSON format by default, a log entry has the following aspect:

```
{
    “kind”: “Event”,
    “apiVersion”: “audit.k8s.io/v1beta1”,
    “metadata”: {
        “creationTimestamp”: “2018-10-08T08:26:55Z”
    },
    “level”: “Request”,
    “timestamp”: “2018-10-08T08:26:55Z”,
    “auditID”: “288ace59-97ba-4121-b06e-f648f72c3122”,
    “stage”: “ResponseComplete”,
    “requestURI”: “/api/v1/pods?limit=500”,
    “verb”: “list”,
    “user”: {
        “username”: “admin”,
        “groups”: [“system:authenticated”]
    },
    “sourceIPs”: [“10.0.138.91”],
    “objectRef”: {
        “resource”: “pods”,
        “apiVersion”: “v1”
    },
    “responseStatus”: {
        “metadata”: {},
        “code”: 200
    },
    “requestReceivedTimestamp”: “2018-10-08T08:26:55.466934Z”,
    “stageTimestamp”: “2018-10-08T08:26:55.471137Z”,
    “annotations”: {
        “authorization.k8s.io/decision”: “allow”,
        “authorization.k8s.io/reason”: “RBAC: allowed by
ClusterRoleBinding “admin-cluster-binding” of ClusterRole “cluster-
admin” to User “admin””
    }
}
```

From this document you can easily extract the user (or serviceaccount software entity) that originated the request, the request URI, API objects involved, timestamp and the API response, **allow**, in this example.

You can define which events you want to log passing a YAML-formatted policy configuration to the API executable.

For instance, if you configure append the following parameters to the **kube-apiserver** command line:

```
    - --audit-log-path=/var/log/apiserver/audit.log
    - --audit-policy-file=/extra/policy.yaml
```

The API will load the configuration from the path above and output the log to **`/var/log/apiserver/audit.log`**

There are many other flags you can configure to tune the audit log, like log rotation, max time to live, and more. It's important to note that you can also configure your API to send audit entries, using a webhook trigger, in case you want to store and index them using an external engine (like ElasticSearch or Splunk).

Audit log policies configuration

The YAML policies file has the following structure:

```
apiVersion: audit.k8s.io/v1beta1
kind: Policy
omitStages:
  - "RequestReceived"
rules:
  - level: Request
    users: ["admin"]
    resources:
      - group: ""
        resources: ["*"]
  - level: Request
    user: ["system:anonymous"]
    resources:
      - group: ""
        resources: ["*"]
```

Using this config, you can match the different keys of a log entry to a specific value, set of values or a catch-all wildcard. The example above will log the requests made by the admin user as well as any request made by an anonymous system user.

If you create a new user (see above) that is not associated to any Role or ClusterRole, and then try to get the list of pods:

```
kubectl get pods
No resources found.
Error from server (Forbidden): pods is forbidden: User
"system:anonymous" cannot list pods in the namespace "default"
```

The log will register the request:

```
{
    “kind”: “Event”,
    “apiVersion”: “audit.k8s.io/v1beta1”,
    “metadata”: {
        “creationTimestamp”: “2018-10-08T10:00:20Z”
    },
    “level”: “Request”,
    “timestamp”: “2018-10-08T10:00:20Z”,
    “auditID”: “5fc5eab3-82f5-480f-93d2-79bfb47789f1”,
    “stage”: “ResponseComplete”,
    “requestURI”: “/api/v1/namespaces/default/pods?limit=500”,
    “verb”: “list”,
    “user”: {
        “username”: “system:anonymous”,
        “groups”: [“system:unauthenticated”]
    },
    “sourceIPs”: [“10.0.141.137”],
    “objectRef”: {
        “resource”: “pods”,
        “namespace”: “default”,
        “apiVersion”: “v1”
    },
    “responseStatus”: {
        “metadata”: {},
        “status”: “Failure”,
        “reason”: “Forbidden”,
        “code”: 403
    },
    “requestReceivedTimestamp”: “2018-10-08T10:00:20.605009Z”,
    “stageTimestamp”: “2018-10-08T10:00:20.605191Z”,
    “annotations”: {
        “authorization.k8s.io/decision”: “forbid”,
        “authorization.k8s.io/reason”: “”
    }
}
```

You have a comprehensive audit policy example here. Rule ordering is important because decision is taken in a top-down first match fashion.

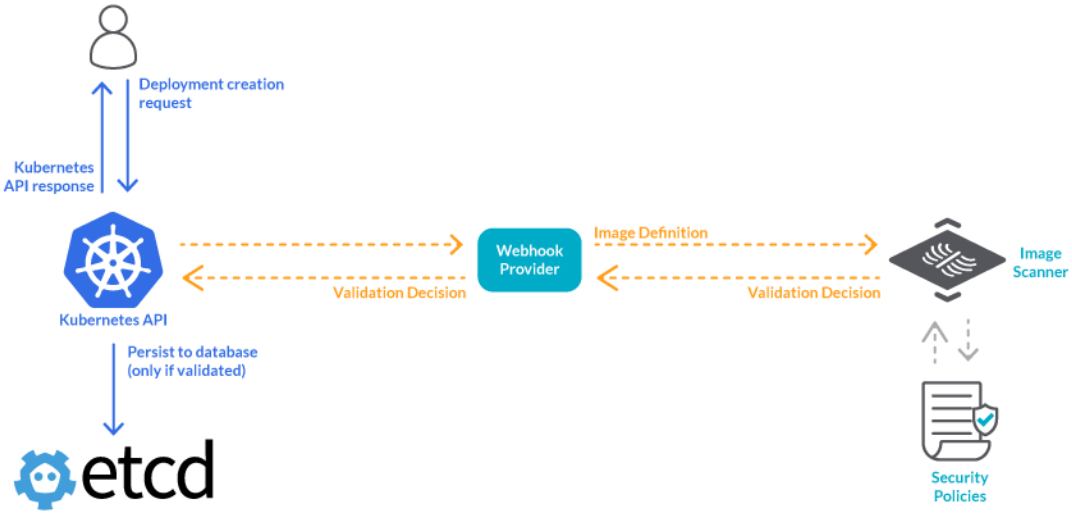
Extending the Kubernetes API using security admission controllers

Kubernetes was designed to be highly extensible, offering you the possibility to plug any security software that you might want to use to process and filter the workloads launched in your system.

A feature that makes admission webhooks especially interesting for the security compliance is that they are evaluated **before** actually executing the requests. That means you can block the access to any suspicious software before the pods are even created.

You can create your own admission controller implementing the webhook interface defined by Kubernetes.

They can also block pods from running if the cluster is out of resources or if the images are not secure. And, as we saw earlier, they can even mutate the request to tweak the resources request from a pod.



Again, all of this is done **before the request is persisted in etcd**, which means **before it is executed**. This is what makes Kubernetes admissions controllers such a perfect candidate to **deploy preventive security controls** on your cluster.

There are three specific admission controllers let you expand the API functionality via webhooks:

- **ImagePolicyWebhook** to decide if an image should be admitted.
- **MutatingAdmissionWebhook** to modify a request.
- **ValidatingAdmissionWebhook** to decide whether the request should be allowed to run at all.

Let's imagine we want to implement an ImagePolicyWebhook.

First, we'll need to make sure that the webhook is enabled when we start kube-apiserver:

```
kube-apiserver --enable-admission-plugins=ImagePolicyWebhook …
```

We also need to configure the webhook server that will be called by the API server:

```
kube-apiserver --admission-control-config-file=admission-config.yaml …
```

An example `admission-config.yaml` contains an `AdmissionConfiguration` object:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: ImagePolicyWebhook
  configuration:
    imagePolicy:
      kubeConfigFile: <path-to-kubeconfig-file>
      allowTTL: 50
      denyTTL: 50
      retryBackoff: 500
      defaultAllow: true
```

And then, the webhook `server` is configured into a kubeconfig file:

```
# clusters refers to the remote service.
clusters:
- name: name-of-remote-imagepolicy-service
  cluster:
    certificate-authority: /path/to/ca.pem    # CA for verifying the
remote service.
    server: https://images.example.com/policy # URL of remote service
to query. Must use 'https'.
# users refers to the API server's webhook configuration.
users:
- name: name-of-api-server
  user:
    client-certificate: /path/to/cert.pem # cert for the webhook
admission controller to use
    client-key: /path/to/key.pem          # key matching the cert
```

Please refer to the ImagePolicyWebhook documentation for a detailed description of the configuration options and alternatives.

We can now code our HTTP server to attend the webhook requests.

Once the Kubernetes API server receives a request for a deployment, our webhook will receive a JSON request similar to:

```
{
  “apiVersion”:”imagepolicy.k8s.io/v1alpha1”,
  “kind”:”ImageReview”,
  “spec”:{
    “containers”:[
      {
        “image”:”myrepo/myimage:v1”
      }
    ],
    “namespace”:”mynamespace”
  }
}
```

Then, you can process this request on your server. For example, checking that image against your image scanner to ensure that it doesn't contain vulnerabilities or misconfigurations.

In our example, that image does not conform with our company policies, so we would respond with the following JSON payload:

```
{
  “apiVersion”: “imagepolicy.k8s.io/v1alpha1”,
  “kind”: “ImageReview”,
  “status”: {
    “allowed”: false,
    “reason”: “image runs as root”
  }
}
```

Because we have rejected one part of the request, the entire API request is immediately rejected, the image won't be deployed, and an error is returned to the end-user.

Additionally, Sysdig Secure Scan Engine provides a webhook service specifically designed to enable this feature.

Sysdig's Admission Controller builds upon Kubernetes and enhances the capacity of the image scanner to check images for Common Vulnerabilities and Exposures (CVEs), misconfigurations, outdated images, etc., elevating the scan policies from detection to actual prevention. Container images that do not fulfill the configured admission policies will be rejected from the cluster before being assigned to a node and allowed to run.

Securing Kubernetes etcd

etcd is a key-value distributed database that persists Kubernetes state. The etcd configuration and upgrading guide stresses the security relevance of this component:

"Access to etcd is equivalent to root permission in the cluster so ideally, only the API server should have access to it. Considering the sensitivity of the data, it is recommended to grant permission to only those nodes that require access to etcd clusters."

You can enforce these restrictions in three different (complementary) ways:

- Regular Linux firewalling (iptables/netfilter, etc).
- Run-time access protection.
- PKI-based authentication + parameters to use the configured certs.

PKI-based authentication for etcd

Ideally, you should create two sets of certifspec:

```
  containers:
  - name: private-reg-container
    image: <your-private-image>
  imagePullSecrets:
  - name: regcredicate and key pairs to be used exclusively for etcd.
One pair will verify member to member connections and the other pair
will verify Kubernetes API to etcd connections.
```

Conveniently, the etcd project provides these scripts to help you generate the certificates.

Once you have all of the security artifacts (certificates, keys and authorities), you can secure etcd communications using the following configuration flags:

etcd peer-to-peer TLS

This will configure authentication and encryption between etcd nodes. To configure etcd with secure peer to peer communication, use the flags:

- –peer-key-file=<peer.key>
- –peer-cert-file=<peer.cert>
- –peer-client-cert-auth
- –peer-trusted-ca-file=<etcd-ca.cert>

Kubernetes API to etcd cluster TLS

To allow Kubernetes API to communicate with etcd, you will need:

- etcd server parameters:
  - –cert-file=
  - –key-file=
  - –client-cert-auth
  - –trusted-ca-file= (can be the same you used for peer to peer)
- Kubernetes API server parameters:
  - –etcd-certfile=k8sclient.cert
  - –etcd-keyfile=k8sclient.key

It may seem like a lot of parameters at first sight, but it's just a regular PKI design.

Using a trusted Docker registry

If you don't specify otherwise, Kubernetes will just pull the Docker images from the public registry Docker Hub. This is fine for testing or learning environments, but it's not convenient for production, as you probably want to keep images and its content private within your organization.

Allowing users to pull images from a public registry is essentially giving access inside your Kubernetes cluster to any random software found on the Internet. Most of the popular Docker image publishers curate and secure their software, however, you don't have any guarantee that your developers are going to pull from trusted authors only.

Providing a trusted repository using cloud services (Docker Hub subscription, Quay.io, Google/AWS/Azure also provide their own service) or locally rolling your own (Docker registry, Portus or Harbor, etc), are two ways to solve this problem.

You will pre-validate and update every image in your registry. Apart from any QA and testing pipeline you regularly apply to your software, this usually means scanning your Docker images for known vulnerabilities and bad security practices.

Assuming you already have a pre-populated trusted repository, you need to tell Kubernetes how to pull from it and ideally, forbid any other unregistered images.

Configure private Docker registry in Kubernetes

Kubernetes provides a convenient way to configure a private Docker registry and store access credentials, including server URL, as a secret:

```
kubectl create secret docker-registry regcred --docker-server=<your-
registry-server> --docker-username=<your-name> --docker-
password=<your-pword> --docker-email=<your-email>
```

This data will be base64 encoded and included inline as a field of the new secret:

```
{
    "apiVersion": "v1",
    "data": {
        ".dockercfg":
"eyJyZWdpc3RyeS5sb2NhbCI6eyJ1c2VybmFtZSI6ImpvaG5kb3ciLCJwYXNzd29yZCI
6InNlY3JldHBhc3N3b3JkIiwiZW1haWwiOiJqb2huQGRvZSIsImF1dGgiOiJhbTlvYm1
SdmR6cHpaV055WlhSd1lYTnpkMjl5WkE9PSJ9fQ=="
    },
    "kind": "Secret",
    "metadata": {
        "creationTimestamp": "2018-04-08T19:13:52Z",
        "name": "regcred",
        "namespace": "default",
        "resourceVersion": "1752908",
        "selfLink": "/api/v1/namespaces/default/secrets/regcred",
        "uid": "f9d91963-3b60-11e8-96b4-42010a800095"
    },
    "type": "kubernetes.io/dockercfg"
}
```

Then, you just need to import this secret using the label imagePullSecrets in the pod definition.

```
spec:
  containers:
  - name: private-reg-container
    image: <your-private-image>
  imagePullSecrets:
  - name: regcred
```

You can also associate a serviceAccount with imagePullSecrets. The deployments / pods using such serviceAccount will have access to the secret containing registry credentials.

Kubernetes trusted image collections: Banning non trusted registry

Once you have created your trusted image repository and Kubernetes pod deployments are pulling from it, the next security measure is to forbid pulling from any non-trusted source.

There are several, complementary ways to achieve this. You can, for example, use ValidatingAdmissionWebhooks. This way, the Kubernetes control plane will delegate image validation to an external entity.

You have an example implementation here, using Grafeas to only allow container images signed by a specific key, configurable via a configmap.

Using our Sysdig Secure, you can also create an image whitelist based on image sha256 hash codes. Any non-whitelisted image will fire an alarm and container execution will be immediately stopped.

Kubernetes TLS certificates rotation and expiration

Modern Kubernetes deployments and managed cloud Kubernetes providers will properly configure TLS, so the communication between the API server and the kubelets, users and pods is already secured. Thus, we will only focus on the maintenance and rotation aspects of these certificates.

Setting a certificate rotation policy from the start will protect you against the usual key mismanagement or leaking that is bound to happen over long periods of time, an occurrence that is often overlooked.

Let's explore three Kubernetes TLS certificate rotation and expiration scenarios:

- kubelet TLS certificate rotation & expiration.
- serviceAccount token rotation.
- Kubernetes user cert rotation & expiration.

Note that the current TLS implementation in the Kubernetes API has no way to verify a certificate besides checking the origin. Neither CRL (Certificate Revocation List) nor OCSP (Online Certificate Status Protocol) are implemented. This means that a lost or exposed certificate will be able to authenticate to the API as long as it hasn't expired.

There are a few ways to mitigate the impact:

- issue (very) short lived certificates to keep the period of potential exposure small.
- remove the permissions in Kubernetes RBAC. You cannot reuse the username until the certificate has expired.
- recreate the certificate authority and issue new certificates to all active users.
- consider OIDC (OpenID Connect) as a alternative authentication method.

Kubernetes kubelet TLS certificate rotation

The kubelet serves as the bridge between the node operating system and the cluster logic, and thus is a critical security component.

By default, the kubelet executable will load its certificates from a regular directory that is passed as argument:

```
--cert-dir=/var/lib/kubelet/pki/

/var/lib/kubelet/pki# ls
kubelet-client.crt  kubelet-client.key  kubelet.crt  kubelet.key
```

You can regenerate the certs manually using the root CA of your cluster. However, starting from Kubernetes 1.8, there is an automated approach at your disposal.

You can instruct your kubelets to renew their certificates automatically as the expiration date approaches using the config flags:

- `--rotate-certificates`

and

- `--feature-gates=RotateKubeletClientCertificate=true`

By default, the kubelet certificates expire in one year. You can tune this parameter passing the flag `--experimental-cluster-signing-duration` to the kube-controller-manager binary.

Kubernetes serviceAccount token rotation

Every time you create a serviceAccount, a Kubernetes secret storing its auth token is automatically generated.

```
 $ kubectl get serviceaccounts
NAME             SECRETS   AGE
default          1         26d
falco-account    1         18d
sysdig-account   1         12d

$ kubectl get secrets
NAME                           TYPE
DATA      AGE
default-token-f2lmn            kubernetes.io/service-account-token
3         26d
falco-account-token-jvgtz      kubernetes.io/service-account-token
3         18d
sysdig-account-token-9sjgd     kubernetes.io/service-account-token
3         12d
```

You can request new tokens from the API and replace the old ones:

```
$ kubectl delete secret falco-account-token-jvgtz
$ cat > /tmp/rotate-token.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: falco-account-token
  annotations:
    kubernetes.io/service-account.name: falco-account
type: kubernetes.io/service-account-token
EOF
$ kubectl create -f /tmp/rotate-token.yaml
```

If you describe the new secret, you will be able to see the new token string. Note that existing pods using this serviceAccount will continue using the old (invalid) token, so you may want to plan a rolling update over the affected pods to start using the new token.

Updating serviceAccount tokens is not as common as updating user certs, passwords, etc. There is no fully automated way of doing this other than using the Kubernetes API at the moment. Consider whether or not this security artifact rotation makes sense for your use cases.

Kubernetes user TLS certificate rotation

As we have seen in the 'How to create a Kubernetes user' example, you can assign a certificate to a user, but there is no User API object per se.

When you sign the user certificate using Kubernetes root CA, you can assign an expiration date using the -days parameter to enforce routinary rotation:

```
openssl x509 -req -in john.csr -CA /etc/kubernetes/pki/ca.crt -CAkey
/etc/kubernetes/pki/ca.key -CAcreateserial -days 365 -out john.crt

$ openssl x509 -in john.crt -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 11309651818125161149 (0x9cf3f46850b372bd)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not Before: Apr  3 10:43:25 2018 GMT
            Not After : Apr  3 10:43:25 2019 GMT
```

Then, you can replace the old user certificate using the config set-credentials command:

```
$ kubectl config set-credentials john --client-certificate=/home/
newusers/john.crt --client-key=/home/newusers/john.key
```

Securing Kubernetes hosts

While the scope of this guide is securing Kubernetes, a lot of people tend to forget that Kubernetes clusters run in hosts. A security breach in a host will compromise everything you are running there, including Kubernetes and the workloads inside.

There is a lot of literature about securing hosts: secure networking, security tools like SELinux or AppArmor, IPtables, etc.

We are not trying to cover everything here, but there are some important things related to Kubernetes that can make it much easier to defend your hosts against undesired intrusions.

Using a minimal host OS

Your hosts only mission should be to run Kubernetes. Everything else is only increasing the chances of being attacked. Each additional library, program, service or process running is increasing the attack surface.

In order to limit risk and reduce attack vectors, it is recommended to use one of the different options of minimal OS designed to run Kubernetes. There are several options like CoreOS, Red Hat Atomic or Rancher. These distributions have specific libraries and dependencies required to run containers, and nothing else.

Update system patches

Old versions of libraries, kernels and applications are some of the main vulnerabilities a host can have. Once a version is known vulnerable, the exploits can be easily automatized and used at great scale. Updates and security patches must be applied in a systematic and consistent way to ensure that all of your hosts are as safe as possible.

You should also check that the host OS you are using is receiving the latest security updates. Sometimes distributions stop supporting old versions and your systems can become vulnerable.

Node recycling

In cloud native environments, hosts are born to be ephemeral. A good strategy to recycle your nodes has several advantages:

- It is a good way to clean hosts that could have been compromised.
- It helps you keep your system up to date if you update the image when you spawn new nodes.
- It will help prevent node failing due to fatigue, like disks filling with logging, problems in updates, etc.
- It will prepare your infrastructure to kill hosts and spawn new ones. This will be a good test of fault tolerance.

Running CIS benchmark security tests

Even when you have meticulously configured your Kubernetes cluster following security best practices, there are a lot of factors you may have overlooked. CIS (Center for Internet Security) has published its benchmarks that you can pass to your clusters and ensure that they comply with the set of rules of the CIS recommendations.

There are several benchmarks that can be of help:

- CIS Kubernetes Benchmark: Checks Kubernetes setup and best security practices to ensure the cluster is properly configured to be safe.
- CIS Docker Benchmark: Assesses the configuration of docker service running in the machine (assuming you are using docker, of course).
- There are different OS security benchmarks for different Linux distributions so you can check the security of the system running below your Kubernetes clusters.

These benchmarks have a lot of tests and not all will apply to every cluster. They are recommendations, so you can opt-out in some particular tests. Anyway it is a very good starting point to measure the security level of your clusters.

CHAPTER 3

Understanding Kubernetes RBAC

Kubernetes RBAC security context is a fundamental part of your Kubernetes security best practices. We will learn how to create a user in Kubernetes and set Kubernetes permissions using RBAC.

Kubernetes role-based access control (RBAC)

Kubernetes RBAC is essentially an authorization and access control specification where you define the actions (GET, UPDATE, DELETE, etc) that Kubernetes subjects (i.e. human users, software, kubelets) are allowed to perform over Kubernetes entities (i.e. pods, secrets, nodes).

RBAC uses the "rbac.authorization.k8s.io" API group to drive authorization decisions. Before getting started, it's important to understand the API group building blocks:

- Namespaces: Logical segmentation and isolation, or "virtual clusters". Correct use of Kubernetes namespaces is fundamental for security, as you can group together users, roles and resources according to business logic without granting global privileges for the cluster. Typically you use a namespace to group a project, application, team or customer.
- **Subjects:** The security "actors".
  - **Regular users:** Humans or other authorized accesses from outside the cluster. Kubernetes delegates the user creation and management to the administrator. In practice, this means that you can "refer" to a user, as we can see on the Kubernetes RBAC examples below, but there's no User API object per se.
  - **ServiceAccounts:** Used to assign permissions to software entities. Kubernetes creates its own default serviceAccounts and you can create additional ones for your pods/deployments. Any pod run by Kubernetes gets its own privileges through its serviceAccount, and they're applied to all processes run within the containers of that pod.
  - **Groups of users:** Kubernetes user groups are not explicitly created; instead, the API can implicitly group users using a common property, like the prefix of a serviceAccount or the organization field of a user certificate. As with regular Linux permissions, you can assign RBAC privileges to entire groups.
- **Resources:** The entities that will be accessed by the subjects.
  - Resources can refer to a generic entity ("pod", "deployment", etc), subresources such as the logs coming from a pod ("pod/log") or the particular resource name like an Ingress: "ingress-controller-istio", including custom resources your deployment defines.

- Resources can also refer to Pod Security Policies or PSP.

- **Role and ClusterRole:** A set of permissions over a group of resources. Think of it as a "security profile"; a Role is always confined to a single namespace, a ClusterRole is cluster-scoped.
  - Before designing your security policy, take into account that Kubernetes RBAC permissions are explicitly additive and there are no "deny" rules.
  - Some resources only make sense at the cluster level (i.e. nodes); you need to create a ClusterRole to control access in this case.
  - Roles define a list of actions that can be performed over the resources or **verbs**: GET, WATCH, LIST, CREATE, UPDATE, PATCH, DELETE.
- **RoleBindings and ClusterRoleBindings:** Grants the permissions defined in a Role or ClusterRole to a subject or group of subjects. Again, RoleBindings are bound to a certain namespace while ClusterRoleBindings are cluster-global.

Let's start looking at these Role and RoleBinding definitions:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [“”] # “” indicates the core API group
  resources: [“pods”]
  verbs: [“get”, “watch”, “list”]

# This role binding allows “jane” to read pods in the “default”
namespace.
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader

  apiGroup: rbac.authorization.k8s.io
```

We define a Role that grants the verbs [“get”, “watch”, “list”] over any pod resource, but only in the "default" namespace. Then, we create a RoleBinding that grants the permissions defined in "pod-reader" to the user "jane".

RBAC configuration: API server flags

Start by making sure your cluster configuration supports RBAC. The location of the configuration file is your kube-apiserver manifest. This depends on the deployment method, but it's usually inside `/etc/kubernetes/manifests` in either the master node(s) or the apiserver pod.

Look for this flag: `--authorization-mode=Node,RBAC`

Node authorization is used by the kubelets. We discuss kubelet permissions in the 'Securing Kubernetes components' chapter of this guide.

While the API server has plenty of flag options, some are best avoided when taking a best practices approach to security:

- `--insecure-port:` Opens up access to unauthorized, unauthenticated requests. If this parameter is equal to 0, it means no insecure port.
- `--insecure-bind-address:` Ideally, you should avoid insecure connections altogether, but in case you really need them, you can use this parameter to just bind to localhost. Make sure this parameter is not set, or at least not set to a network-reachable IP address.
- `--anonymous-auth:` Enables anonymous requests to the secure port of the API server.

How to create Kubernetes users and serviceAccounts

When it comes to Kubernetes users and permissions, the best approach is one that applies the principle of least privilege, which promotes minimal user profile privileges based on users' job necessities:

- Grant the minimum required access privileges for the task that a user or pod needs to carry out.
- Prefer Role and RoleBinding to their cluster counterparts, when possible. It's much easier to control security when it is bound to independent namespaces.
- Avoid the use of wildcards ["*"] when defining access to resources or verbs over these resources. Be specific.

ServiceAccounts are used to provide an identity to the processes that run in your pods (similar concept to the sshd or www-data users in a Linux system). If you don't specify a serviceAccount, these pods will be assigned to the default service account of their namespace.

Using default serviceAccounts can be vague and prone to oversights. Try to use service-specific service accounts instead. This way, you can granularly control the API access that you grant to any software entity inside your cluster.

How to create a Kubernetes serviceAccount step by step

Imagine, for example, that your app needs to query the Kubernetes API to retrieve pod information and state changes because you want to notify and send updates using webhooks.

You just need 'read-only' access to monitor one specific namespace. Using a serviceAccount, you can grant these specific privileges (and nothing else) to your software agent.

The default serviceAccount (the one you will get if you don't specify any) is unable to retrieve this information:

```
$ kubectl auth can-i list pods -n default
--as=system:serviceaccount:default:default
no
```

We have created an example deployment to showcase this Kubernetes security feature.

Take a look at the rbac/flask.yaml file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: flask
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: flask-backend
  namespace: flask
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: flask-backend-role
  namespace: flask
rules:
  - apiGroups: [“”]
    resources: [“pods”]
    verbs: [“get”, “list”, “watch”]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: flask-backend-role-binding
  namespace: flask
subjects:
```

```
  - kind: ServiceAccount
    name: flask-backend
    namespace: flask
roleRef:
  kind: Role
  name: flask-backend-role
  apiGroup: rbac.authorization.k8s.io
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: flask
  namespace: flask
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: flask
    spec:
      serviceAccount: flask-backend
      containers:
      - image: mateobur/flask:latest
        name: flask
        ports:
        - containerPort: 5000
```

This will create a serviceAccount ("flask backend"), a Role that grants some permissions over the other pods in this "flask" namespace, a RoleBinding associating the serviceAccount and the Role and finally, a deployment of pods that will use the serviceAccount:

```
$ kubectl create -f flask.yaml
```

If you query the secrets for the flask namespace, you can verify that an API access token was automatically created for your serviceAccount:

```
$ kubectl get secrets -n flask
NAME                        TYPE
DATA      AGE
default-token-fjfgn         kubernetes.io/service-account-token
3         5m
flask-backend-token-68b6q   kubernetes.io/service-account-token
3         5m
```

Then, you can check that the permissions are working as expected with the **kubectl auth** command that can query access for verbs and subjects, as well as impersonate other accounts:

```
$ kubectl auth can-i list pods -n default --as=system:serviceaccount:
flask:flask-backend
no

$ kubectl auth can-i list pods -n flask --as=system:serviceaccount:fl
ask:flask-backend
yes

$ kubectl auth can-i create pods -n flask --as=system:serviceaccount:
flask:flask-backend
no
```

To summarize, you will need to configure a serviceAccount and its related Kubernetes RBAC permissions if your software needs to interact with the hosting Kubernetes cluster. Other examples might include the kubelet agents or a Kubernetes Horizontal Pod Autoscaler.

How to create a Kubernetes user step by step

As we mentioned before, Kubernetes users do not have an explicit API object that you can create, list or modify.

Users are bundled as a parameter of a configuration context that defines the cluster name, (default) namespace and username:

```
$ kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO
NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
```

If you look at the current context, you will note that the user has *client-certificate-data* and *client-key-data* attributes (omitted in the output by default for security reasons).

```
$ kubectl config view
…
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

If you have access to the Kubernetes root certification authority, you can generate a new security context that declares a new Kubernetes user.

So, in order to create a new Kubernetes user, let's start creating a new private key:

```
$ openssl genrsa -out john.key 2048
```

Then, you need to create a certificate signing request containing the public key and other subject information:

```
$ openssl req -new -key john.key -out john.csr -subj "/CN=john/
O=examplegroup"
```

Note that Kubernetes will use the *Organization* (O=examplegroup) field to determine user group membership for RBAC.

You will sign this CSR using the root Kubernetes CA, found in /etc/kubernetes/pki for this example. The file location in your deployment may vary:

```
# openssl x509 -req -in john.csr -CA /etc/kubernetes/pki/ca.crt
-CAkey /etc/kubernetes/pki/ca.key -CAcreateserial -out john.crt
Signature ok
subject=/CN=john/O=examplegroup
Getting CA Private Key
```

You can inspect the new certificate:

```
# openssl x509 -in john.crt -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 11309651818125161147 (0x9cf3f46850b372bb)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not Before: Apr  2 20:20:54 2018 GMT
            Not After : May  2 20:20:54 2018 GMT
        Subject: CN=john, O=examplegroup
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
```

Let's repeat this process for a second user so we can show how to assign Kubernetes RBAC permissions to a group:

```
$ openssl genrsa -out mary.key 2048
$ openssl req -new -key mary.key -out mary.csr -subj "/CN=mary/
O=examplegroup"
# openssl x509 -req -in mary.csr -CA /etc/kubernetes/pki/ca.crt
-CAkey /etc/kubernetes/pki/ca.key -CAcreateserial -out mary.crt
```

You can now register the new credentials and config context:

```
$ kubectl config set-credentials john --client-certificate=/home/
newusers/john.crt --client-key=/home/newusers/john.key
$ kubectl config set-context john@kubernetes --cluster=kubernetes
--user=john
Context "john@kubernetes" created.

$ kubectl config get-contexts
CURRENT   NAME                          CLUSTER
AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes
kubernetes-admin
          john@kubernetes               kubernetes
john
```

If you want this file to be portable between hosts, you need to embed the certificates inline. You can do this automatically appending the --embed-certs=true parameter to the kubectl config set-credentials command.

Let's use this new user / context:

```
$ kubectl config use-context john@kubernetes
$ kubectl get pods
Error from server (Forbidden): pods is forbidden: User "john" cannot
list pods in the namespace "default"
```

Ok, this is expected because we haven't assigned any RBAC permissions to our "john" user.

Let's go back to our root admin user and create a new clusterrolebinding:

```
$ kubectl config use-context kubernetes-admin@kubernetes
$ kubectl create clusterrolebinding examplegroup-admin-binding
--clusterrole=cluster-admin --group=examplegroup
clusterrolebinding "examplegroup-admin-binding" created

$ kubectl config use-context john@kubernetes
$ kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
flask-cap   1/1       Running   0          1m
```

Please note that we have assigned these credentials to the group rather than the user, so the user 'mary' should have exactly the same access privileges.

Using an external user directory

Instead of managing your users via manual certificates and contexts, you can delegate authentication to an external backend, like a company-local LDAP directory or cloud-based identity manager.

There are several authentication strategies and protocols supported by the Kubernetes client. Additionally, kubectl can be configured to use an external binary to retrieve user credentials, thus making it compatible with any directory software that has written an integration layer.

Another popular option that is relatively easy to configure and is supported by the major cloud providers is OpenID Connect(OIDC).

To use OpenID, first, you need to instruct the Kubernetes API service about the external endpoint and client ID. Here you have the complete list of OIDC parameters. These are the two mandatory ones (Google Cloud example):

```
- --oidc-issuer-url=https://accounts.google.com
- --oidc-client-id=someuser.apps.googleusercontent.com
```

Once the API backend is configured, the Kubernetes operator can obtain a valid context (an actual set of Kubernetes credentials) using the set-credentials subcommand and a set of ID tokens and URL provided by the authentication backend:

```
kubectl config set-credentials USER_NAME
   --auth-provider=oidc
   --auth-provider-arg=idp-issuer-url=( issuer url )
   --auth-provider-arg=client-id=( your client id )
   --auth-provider-arg=client-secret=( your client secret )
   --auth-provider-arg=refresh-token=( your refresh token )
   --auth-provider-arg=idp-certificate-authority=( path to your ca
certificate )
   --auth-provider-arg=id-token=( your id_token )
```

You have a complete step-by-step tutorial for the Google Cloud user backend here.

CHAPTER 4

Security at the pod level: K8s security context, PSP, Network Policies

Once you have defined Kubernetes RBAC: users and services credentials and permissions, we can start leveraging Kubernetes orchestration capabilities to configure security at the pod level. In this part, we will learn how to configure security at the pod level using Kubernetes orchestration capabilities: Kubernetes Security Context, Kubernetes Security Policy and Kubernetes Network Policy resources to define the container privileges, permissions, capabilities and network communication rules. We will also discuss how to limit resource starvation with allocation management.

Kubernetes admission controllers

An admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized. Admission controllers pre-process the requests and can provide utility functions (such as filling out empty parameters with default values), but can also be used to enforce security policies and other checks.

Admission controllers are found on the kube-apiserver conf file:

```
--enable-admission-plugins=NamespaceLifecycle,LimitRanger,Service
Account,TaintNodesByCondition,Priority,DefaultTolerationSeconds,
DefaultStorageClass,StorageObjectInUseProtection,PersistentVolume
ClaimResize,MutatingAdmissionWebhook,ValidatingAdmissionWebhook,
RuntimeClass,ResourceQuota
```

Here are the admission controllers that can help you strengthen security:

DenyEscalatingExec: Forbids executing commands on an “escalated” container. This includes pods that run as privileged, have access to the host IPC namespace and have access to the host PID namespace. Without this admission controller, a regular user can escalate privileges over the Kubernetes node by just spawning a terminal on these containers.

NodeRestriction: Limits the node and pod objects a kubelet can modify. Using this controller, a Kubernetes node will only be able to modify the API representation of itself and the pods bound to this node.

PodSecurityPolicy: This admission controller acts on creation and modification of the pod and determines if it should be admitted based on the requested Security Context and the

available Pod Security Policies. The PodSecurityPolicy objects define a set of conditions and security context that a pod must declare in order to be accepted into the cluster. We will cover PSP in more detail below.

ValidatingAdmissionWebhooks: Calls any external service that is implementing your custom security policies to decide if a pod should be accepted in your cluster. For example, you can pre-validate container images using Grafeas, a container-oriented auditing and compliance engine, or validate scanned images.

There is a recommended set of admission controllers to run depending on your Kubernetes version. After version 1.10 all the recommended controllers are enabled by default.

Kubernetes security context

When you declare a pod/deployment, you can group several security-related parameters, like SELinux profile, Linux capabilities, etc, in a Security context block:

```
…
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup
    fsGroup: 2000

…
```

You can configure the following parameters as part of your security context:

Privileged: Processes inside of a privileged container get almost the same privileges as those outside of a container, such as being able to directly configure the host kernel or host network stack.

Other context parameters that you can enforce include:

User and Group ID for the processes, containers and volumes: When you run a container without any security context, the 'entrypoint' command will run as root. This is easy to verify:

```
$ kubectl run -i --tty busybox --image=busybox --restart=Never -- sh
/ # ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 sh
```

Using the runAsUser parameter you can modify the user ID of the processes inside a container. For example:

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: gcr.io/google-samples/node-hello:1.0
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

If you spawn a container using this definition, you can check that the initial process is using *UID 1000*.

```
USER   PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
1000     1  0.0  0.0   4336   724 ?     Ss   18:16   0:00 /bin/sh -c
node server.js
```

And any file you create inside the /data/demo volume will use GID 2000 (due to the fsGroup parameter).

Security Enhanced Linux (SELinux): You can assign SELinuxOptions objects using the seLinuxOptions field. Note that the SELinux module needs to be loaded on the underlying Linux nodes for these policies to take effect.

Capabilities: Linux capabilities break down root full unrestricted access into a set of separate permissions. This way, you can grant some privileges to your software, like binding to a port < 1024, without granting full root access.

There is a default set of capabilities granted to any container if you don't modify the security context. For example, using chown to set file permissions or net_raw to craft raw network packages.

Using the pod security context, you can drop default Linux capabilities and/or add non-default Linux capabilities. Again, applying the principle of least-privilege, you can greatly reduce the damage of any malicious attack taking over the pod.

As a quick example, you can spawn the flask-cap pod:

```
$ kubectl create -f flask-cap.yaml

apiVersion: v1
kind: Pod
metadata:
  name: flask-cap
  namespace: default
spec:
  containers:
  - image: mateobur/flask
    name: flask-cap
    securityContext:
      capabilities:
        drop:
          - NET_RAW
          - CHOWN
```

Note that some *securityContext* should be applied at the pod level, while other labels are applied at container level.

If you spawn a shell, you can verify that these capabilities have been dropped:

```
$ kubectl exec -it flask-cap bash
root@flask-cap:/# ping 8.8.8.8
ping: Lacking privilege for raw socket.
root@flask-cap:/# chown daemon /tmp
chown: changing ownership of '/tmp': Operation not permitted
```

AppArmor and Seccomp: You can also apply the profiles of these security frameworks to Kubernetes pods. This feature is in beta state as of Kubernetes 1.9, profile configurations are referenced using annotations for the time being.

AppArmor, Seccomp or SELinux allow you to define run-time profiles for your containers, but if you want to define run-time profiles at a higher level with more context, Sysdig Falco and Sysdig Secure can be better options. Sysdig Falco monitors the run-time security of your containers according to a set of user-defined rules. It has some similarities and some important differences with the other tools we just mentioned (reviewed in the "SELinux, Seccomp, Sysdig Falco, and you" article).

AllowPrivilegeEscalation: The execve system call can grant a newly-started program privileges that its parent did not have, such as the *setuid or setgid Linux flags*. This is controlled by the *AllowPrivilegeEscalation* boolean and should be used with care and only when required.

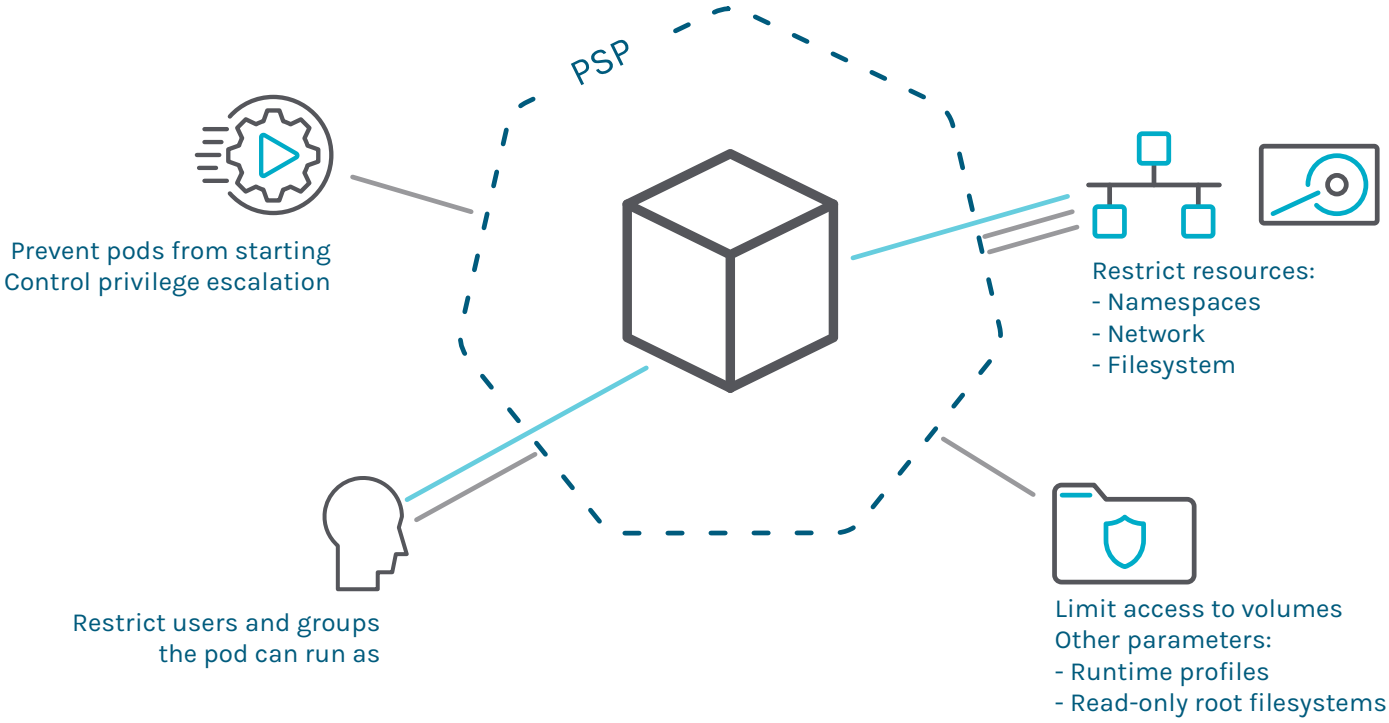
ReadOnlyRootFilesystem: This controls whether a container will be able to write into the root filesystem. It is common that the containers only need to write on mounted volumes that persist the state, as their root filesystem is supposed to be immutable. You can enforce this behavior using the *readOnlyRootFilesystem* flag:

```
$ kubectl create -f https://raw.githubusercontent.com/mateobur/
kubernetes-securityguide/master/readonly/flask-ro.yaml
$ kubectl exec -it flask-ro bash
root@flask-ro:/# mount | grep "/ "
none on / type aufs (ro,relatime,si=e6100da9e6227a70,dio,dirperm1)
root@flask-ro:/# touch foo
touch: cannot touch 'foo': Read-only file system
```

Kubernetes security policy

By default, Kubernetes is quite lenient about who or what can create pods and the privileges of containers - one could easily create a privileged container that could cause harm, or jeopardize the security of the system. Pod Security Policies provides a mechanism to lock down the system. They define a set of conditions that a pod must run with in order to be accepted into the cluster.

Kubernetes Pod Security Policy (PSP), often shortened to Kubernetes Security Policy, is implemented as an admission controller. Using security policies, you can restrict the pods that will be allowed to run on your cluster, only if they follow the policy we have defined.



You have different control aspects that the cluster administrator can set:

Control Aspect	Field Names
Running of privileged containers	privileged
Usage of the root namespaces	hostPID, hostIPC
Usage of host networking and ports	hostNetwork, hostPorts
Usage of volume types	volumes
Usage of the host filesystem	allowedHostPaths
White list of FlexVolume drivers	allowedFlexVolumes
Allocating an FSGroup that owns the pod's volumes	fsGroup
Requiring the use of a read only root file system	readOnlyRootFilesystem
The user and group IDs of the container	runAsUser, supplementalGroups
Restricting escalation to root privileges	allowPrivilegeEscalation, defaultAllowPrivilegeEscalation
Linux capabilities	defaultAddCapabilities, requiredDropCapabilities, allowedCapabilities
The SELinux context of the container	seLinux
The AppArmor profile used by containers	annotations
The seccomp profile used by containers	annotations
The sysctl profile used by containers	annotations

PSP functionality is enabled by adding `PodSecurityPolicy` to the kubeapi-server configuration, i.e.

```
--enable-admission-plugins=...,PodSecurityPolicy
```

Pod Security Policies are quite complex, therefore it is critically important to understand exactly how they work before attempting to implement them.

One thing to bear in mind is that this is a global setting and you cannot just enable it for a particular part of your cluster. Once enabled, it affects every deployment and namespace in your cluster, including the **`default`**, **`kube-public`** and **`kube-system`** namespaces. Also, the default policy is to disallow anything that's not explicitly allowed.

Once PSP is enabled, then everything is locked down and you must set policy rules to explicitly allow. Individual PSP policies are implemented as declarative YAML objects. For example:

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false
  runAsUser:
    rule: MustRunAsNonRoot
  seLinux:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
  - 'nfs'
  hostPorts:
  - min: 100
    max: 100
```

This PSP definition implements the following security rules:

- Disallow containers running in privileged mode.
- Disallow containers that require root privileges.
- Disallow containers that access volumes **apart from** NFS volumes.
- Disallow containers that access host ports **apart from** port 100.

There is a direct relation between the Kubernetes Pod Security Context labels and the Kubernetes Pod Security Policies. Your Security Policy will filter allowed pod security contexts defining:

- Default pod security context values (i.e. **`defaultAddCapabilities`**).
- Mandatory pod security flags and values (i.e. **`allowPrivilegeEscalation: false`**).
- Whitelists and blacklists for the list-based security flags (i.e. list of allowed host paths to mount).

For example, to define that container can only mount a specific host path, you would do:

```
allowedHostPaths:
  # This allows "/foo", "/foo/", "/foo/bar" etc., but
  # disallows "/fool", "/etc/foo" etc.
  # "/foo/../" is never valid.
  - pathPrefix: "/foo"
```

PSP policies are applied upon pod creation, and the default policy is aggressively secure. This means you could turn PSP functionality on with no immediately obvious effect. However, as pods disappear over time and Kubernetes tries to start new ones, these may be disallowed causing major problems with your system. Therefore, it is common, and much easier, to apply a reasonably liberal default policy once PSP is enabled, then systematically secure certain aspects, rather than enable it with no default policy and manually enable secured functionality.

PSP and RBAC

When a user accesses the cluster using **`kubectl`**, for example, to manually create a pod, they are authenticated by the apiserver as a particular 'User Account', e.g. 'admin'. However, in a production environment, a user typically creates a Deployment, StatefulSet, Job, or Daemonset, and these in turn use a controller to create the pod. In this case, the controller is authenticated as a particular 'Service Account', e.g. 'default'.

There is a complex RBAC engine that determines the privileges a particular controller has. See 'Kubernetes Role-Based Access Control (RBAC)' on page 32 for a description of RBAC and role binding.

We saw earlier how *Roles and Rolebindings* are implemented using Yaml objects to define who has permissions to perform specific actions on certain resources., i.e.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [“”]
  resources: [“pods”]
  verbs: [“get”, “watch”, “list”]
```

When implementing PSP the Role definition may also specify a PSP object using the `resourceNames` directive, as illustrated below:

```
kind: RoleBinding
apiVersion:
rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.
authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.
authorization.k8s.io
```

```
kind: Role
apiVersion:
rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [“”]
  resources: [“pods”]
  verbs: [“get”, “watch”,
“list”]
  resourceNames: [no_priv_psp]
```

```
kind:
PodSecurityPolicy
apiVersion: extensions/
v1beta1
metadata:
  name: no_priv_psp
spec:
  privileged: false
 runAsUser:
    rule: MustRunAsNonRoot
 seLinux:
    rule: RunAsAny
```

This allows you to define much more granular rules that set which users and service accounts may create within the cluster.

Implementing PSPs

As mentioned previously, you can only use PSPs if they are enabled in your Kubernetes cluster's admission controller. You can easily tell if they're enabled by executing the following command:

```
$ kubectl get psp
the server doesn't have a resource type "podSecurityPolicies".
```

If the command returns the error message illustrated above, then PSP has not been enabled in the cluster. However, if it has been enabled, but no policies have yet been loaded, then you will see the following:

```
$ kubectl get psp
No resources found,
```

If you plan to enable *PodSecurityPolicy*, first configure (or have present already) a default PSP and the associated RBAC permissions, otherwise the cluster will fail to create new pods.

If your cloud provider / deployment design already supports and enables PSP, it will come pre-populated with a default set of policies, for example:

```
$ kubectl get psp
NAME                          PRIV     CAPS    SELINUX    RUNASUSER  FSGROUP    SUPGROUP
READONLYROOTFS   VOLUMES
gce.event-exporter             false   []      RunAsAny   RunAsAny   RunAsAny   RunAsAny false
[hostPath secret]
gce.fluentd-gcp               false    []      RunAsAny   RunAsAny   RunAsAny   RunAsAny false
[configMap hostPath secret]
gce.persistent-volume-binder  false    []      RunAsAny   RunAsAny   RunAsAny   RunAsAny false
[nfs secret]
gce.privileged                true     [*]     RunAsAny   RunAsAny   RunAsAny   RunAsAny false
[*]
gce.unprivileged-addon        false    []      RunAsAny   RunAsAny   RunAsAny   RunAsAny false
[emptyDir configMap secret]
```

In the event you enabled PSP for a cluster that didn't have any pre-populated rule, you can create a permissive policy to avoid run-time disruption and then perform iterative adjustments over your configuration:

For example, this policy below will prevent the execution of any pod that tries to use the root user or group, allowing any other security context:

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: true
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  volumes:
  - '*'
```

Working with PSPs by example

Here, we will go through a number of steps to illustrate the PSP functionality. If you have a Kubernetes system available, then you can perform these steps. If not, then you can follow along in the text.

Note: These steps will enable PSP in your clusters, so do not perform these on a working environment.

Firstly, set up a namespace and a service account.

```
$ kubectl create namespace example-psp
kubectl create serviceaccount -n example-psp foo-user
```

Before we start, we need to set a basic Role Based Access Control (RBAC) configuration in order to be able to run our PSP.

```
$ kubectl create rolebinding -n example-psp fake-editor
--clusterrole=edit --serviceaccount=example-psp:foo-user
```

This command creates a rolebinding called fake-editor between the service account called foo-user and gives it the role of edit.

Before we implement PSP, let's verify that we have unlimited access to create resources. We'll test this by creating a privileged container named privileged01.

```
$ kubectl -n example-psp --as=system:serviceaccount:example-psp:foo-
user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name:      privileged01
spec:
  containers:
    - name:  pause
      image: k8s.gcr.io/pause
      securityContext:
        privileged: true
EOF
pod/privileged01 created
```

The output `pod/privileged01 created` shows that it worked.

We can also see that the order was successful, as follows:

```
$ kubectl get pods -n example-psp
NAME           READY    STATUS     RESTARTS    AGE
privileged01   1/1      Running    0           1m16s
```

Unlimited access to resource creation may be the source of privilege escalation or other threats to our cluster. Enabling PSPs can help prevent this.

Before we can define PSP policies, we need to enable PSP in the cluster. To do so, edit the file `kube-apiserver.yaml`.

Note: The location of this file depends upon your installation.

Ensure PodSecurityPolicy is included in the list of Admission Controllers, e.g.

```
--enable-admission-plugins=NodeRestriction,PodSecurityPolicy
```

Now PSP is enabled, but if we try to create a privileged pod, then we get an error stating there are no policies to check it against:

```
$ kubectl -n example-psp --as=system:serviceaccount:example-psp:foo-
user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name:      privileged02
spec:
  containers:
    - name:  pause
      image: k8s.gcr.io/pause
      securityContext:
        privileged: true
EOF
Error from server (Forbidden): error when creating “STDIN”: pods
“privileged02” is forbidden: no providers available to validate pod
request
```

For this example, we will define a PSP policy explicitly blocking privileged containers.

We shall create a the following PSP policy file, called example-psp.yaml:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - ‘*’
```

We can view the policy as follows:

```
$ kubectl get psp
NAME      PRIV   CAPS    SELINUX    RUNASUSER   FSGROUP    SUPGROUP   READONLYROOTFS   VOLUMES
example   false          RunAsAny   RunAsAny    RunAsAny   RunAsAny   false            *
```

Now we will apply the policy:

```
$ kubectl -n example-psp create -f example-psp.yaml
```

We have enabled PSP on our cluster and also have a PSP defined. Let's try to create the same privileged container again. We'll name it privileged02:

```
$ kubectl -n example-psp --as=system:serviceaccount:example-psp:foo-
user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  Name:  privileged02
spec:
  containers:
    - name:  pause
      image: k8s.gcr.io/pause
      securityContext:
        privileged: true
EOF
Error from server (Forbidden): error when creating "STDIN": pods
"privileged02" is forbidden: unable to validate against any pod
security policy: [spec.containers[0].securityContext.privileged:
Invalid value: true: Privileged containers are not allowed]
```

This time the error indicates that the creation of privileged containers is explicitly forbidden by PSP. Our policy is working as we expected.

We can once again enable privileged containers at our cluster to check that this change of behaviour was produced by the PSP.

We shall edit example-psp.yaml and change privileged: false to privileged: true.

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: true
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - '*'
```

We need to re-apply the policy:

```
$ kubectl -n example-psp apply -f example-psp.yaml
```

Now if we try again to create a privileged container, we'll see that it succeeds:

```
kubectl -n example-psp --as=system:serviceaccount:example-psp:foo-
user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name:      privileged03
spec:
  containers:
    - name:  pause
      image: k8s.gcr.io/pause
      securityContext:
        privileged: true
EOF
pod/privileged03 created
```

Looking at the policy again you will see that privileged containers are explicitly allowed:

```
$ kubectl get psp -n example-psp
```

```
NAME      PRIV   CAPS   SELINUX    RUNASUSER   FSGROUP    SUPGROUP   READONLYROOTFS   VOLUMES
example   true          RunAsAny   RunAsAny    RunAsAny   RunAsAny   false            *
```

Practical considerations

You must take into consideration the ordering of your PSP policies. When multiple policies are available, the pod security policy controller selects them according to the following criteria:

- PodSecurityPolicies which allow the pod as-is without changing defaults or mutating the pod, are preferred. The order of these non-mutating PodSecurityPolicies doesn't matter.
- If the pod must be defaulted or mutated, the first PodSecurityPolicy (ordered by name) to allow the pod is selected.

Please refer to the Kubernetes documentation for further details.

Deploying usually involves the following pseudo process:

1. Apply your PSPs *before* enabling them
2. Enable the PodSecurityPolicy Admission Controller in the kubeAPI configuration
3. Recycle your Pods that are under the control of the PSP
4. Monitor closely and fix/add PSPs as needed

Kubernetes network policies

Kubernetes also defines security at the pod networking level. A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.

You can compare Kubernetes network policies with classic network firewalling (ala iptables) but with one important advantage: using Kubernetes context like pod labels, namespaces, etc.

Kubernetes supports several third-party plugins that implement pod overlay networks. You need to check your provider documentation (these for Calico or Weave) to make sure that Kubernetes network policies are supported and enabled, otherwise, the configuration will show up in your cluster but will not have any effect.

Let's use the Kubernetes example scenario guestbook to show how these network policies work:

```
kubectl create -f https://raw.githubusercontent.com/fabric8io/
kansible/master/vendor/k8s.io/kubernetes/examples/guestbook/all-in-
one/guestbook-all-in-one.yaml
```

This will create 'frontend' and 'backend' pods:

```
$ kubectl describe pod frontend-685d7ff496-7s6kz | grep tier
        tier=frontend
$ kubectl describe pod redis-master-7bd4d6ccfd-8dnlq | grep tier
        tier=backend
```

You can configure your network policy with these logical resources. Abstracting concepts such as IP addresses or physical nodes won't work because Kubernetes can change them dynamically.

Let's apply the following network policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-backend-egress
  namespace: default
  spec:
    podSelector:
      matchLabels:
        tier: backend
        policyTypes:
          - Egress
          egress:
            - to:
              - podSelector:
                matchLabels:
                  tier: backend
```

You can also find that in the repository:

```
$ kubectl create -f netpol/guestbook-network-policy.yaml
```

Then you can get the pod names and local IP addresses using:

```
$ kubectl get pods -o wide
[...]
```

In order to check that the policy is working as expected, you can 'exec' into the 'redis-master' pod and try to ping first a 'redis-slave' (same tier) and then a 'frontend' pod:

```
$ kubectl exec -it redis-master-7bd4d6ccfd-8dnlq bash
$ ping 10.28.4.21
PING 10.28.4.21 (10.28.4.21) 56(84) bytes of data.
64 bytes from 10.28.4.21: icmp_seq=1 ttl=63 time=0.092 ms
$ ping 10.28.4.23
PING 10.28.4.23 (10.28.4.23) 56(84) bytes of data.
(no response, blocked)
```

This policy will be enforced even if the pods migrate to another node or are scaled up/down.

You can also use namespace selectors and CIDR ip blocks for your ingress and egress rules, like in the example below:

```
ingress:
 - from:
   - ipBlock:
       cidr: 172.17.0.0/16
       except:
       - 172.17.1.0/24
   - namespaceSelector:
       matchLabels:
         project: myproject
   - podSelector:
       matchLabels:
         role: frontend
```

Kubernetes resource allocation management

Resource limits are usually established to avoid unintended saturation due to design limitations or software bugs, but can also protect against malicious resource abuse. Unauthorized resource consumption that tries to remain undetected is becoming much more common due to cryptojacking attempts.

There are two basic concepts: **requests** and **limits**.

Requests: The Kubernetes node will check if it has enough resources left to fully satisfy the request before scheduling the pod. This value will be mainly used for scheduling. The requested resources are not guaranteed

You can run a quick example from the resources/flask-resources.yaml repository file

```
apiVersion: v1
kind: Pod
metadata:
  name: flask-resources
  namespace: default
spec:
  containers:
  - image: mateobur/flask
    name: flask-resources
    resources:
      requests:
        memory: 512Mi
      limits:
        memory: 700Mi

$ kubectl create -f resources/flask-resources.yaml
```

Limits: Are the top resource consumption the container can make. Kubernetes ensures that the actual resource consumption never goes over the configured limits.

Let's use the stress load generator to test the limits:

```
root@flask-resources:/# stress --cpu 1 --io 1 --vm 2 --vm-bytes 800M
stress: info: [79] dispatching hogs: 1 cpu, 1 io, 2 vm, 0 hdd
stress: FAIL: [79] (416) <-- worker 83 got signal 9
```

The resources that you can reserve and limit by default using the pod description are:

- CPU
- Main memory
- Local ephemeral storage

There are some third party plugins and cloud providers that can extend the Kubernetes API to allow defining requests and limits over any other kind of logical resources using the Extended Resources interface. You can also configure resource quotas bound to a namespace context.

CHAPTER 5

Securing workloads at runtime

Although image scanning is the first step in managing security risk, runtime security is a critical component of a secure DevOps workflow.

Several security threats, by their very nature, only manifest during runtime:

- Zero-day vulnerabilities
- Misconfigurations
- Software bugs causing erratic behavior or resource leaking
- Internal privilege escalation attempts

Runtime security will protect you against these threats. At the same time, it will also help you verify that the other security barriers are effective, let you configure them, and provide a last line of defense.

The key aspects of runtime security are:

- **Continuous scanning:** By continuously tracking your running images, you can detect which ones are affected by newly discovered vulnerabilities, or aren't compliant after a change in your policies.
- **Threat detection:** Is your container doing what it's supposed to do? By monitoring your container activity, you can immediately identify suspicious behaviour.
- **Automatic incident response:** Once a policy violation is detected, action needs to be taken as soon as possible. An automatic response will block security threats right when they are detected by killing or pausing the affected containers, notifying the relevant people about the incident.
- **Capturing forensics data:** After an incident, you need to find the source to prevent it from happening again. Comprehensive forensics data is crucial for your investigation, as well as being able to correlate events from several sources with concrete Kubernetes resources (namespaces, deployments, containers, etc.).

Let's see how you can implement **container runtime security** when securing your Kubernetes cluster to prevent threats and remediate security incidents when they happen.

How to implement runtime security

Choosing the right tools is vital for an effective implementation. With so many security products available, all with their distinct methods, advantages and weaknesses, you'll need to know about each one before making a decision.

Overall, these products can be grouped into those focused on enforcement vs. auditing. Both groups use **policies** to describe what is considered allowed or disallowed behavior for a process in terms of privileges or resources accessed, like files or network.

Enforcement tools use policies to limit the behavior of a process. This is done by preventing system calls from succeeding and even killing the process in some cases. Seccomp, SELinux and AppArmor are examples of enforcement tools.

Auditing tools use policies to monitor the behavior of a process and notify when its actions step outside the policy. Falco is primarily an auditing tool, although it has some enforcement capabilities.

Let's examine those tools in more detail!

Falco

Falco is the de facto Kubernetes threat detection engine. Falco detects unexpected application behavior and alerts on threats at runtime. An open source tool originally created by us at Sysdig, Falco is the first runtime security project to join the CNCF Incubating stage.

Falco uses eBFP (among other sources) to capture system calls, with Kubernetes application context, to gain visibility into runtime system activity of containers and hosts. By tapping into our Sysdig open-source libraries through Linux system calls, it can run in high performance production environments. Falco also ingests Kubernetes API audit events to provide runtime detection and alerting for orchestration activity.

The security events generated by Falco can be used by other tools, like kubernetes-response-engine, to perform the actual enforcement.

Additionally, you can set up alerts and get notified of those events matching a filter expression.

Installing Falco

Helm is one of the preferred methods for installing Falco on Kubernetes. Using the Falco Helm chart, you can install Falco in a few seconds with a simple command. It provides an extensive set of configuration values to start Falco with different configurations.

To deploy Falco with default configuration on a cluster where Helm is deployed, run:

```
helm install --name falco stable/falco
```

To remove Falco from your cluster run:

```
helm delete falco
```

This method provides several advantages:

- All of the benefits of installing Falco as a Kubernetes DaemonSet, so you don't need to repeat the deployment on new nodes that will get Falco automatically.
- Easier, faster and automated.
- Configuration and rulesets managed by the chart, creating portable and repeatable configuration.
- Bundles some integrations out-of-the-box, like Kubernetes RBAC permissions.

You have more details about this installation method on "Automate Sysdig Falco Deployment Using Helm Charts". You can check other installation methods and options in the Falco documentation.

Anatomy of a Falco rule

Let's inspect a Falco rule to learn how they are built.

```
- rule: Example rule (nginx). This is the human name for the rule.
  desc: Detect any listening socket outside our expected one.
  condition: evt.type in (accept,listen) and (container.
image!=myregistry/nginx or proc.name!=nginx or k8s.ns.name!="load-
balancer")
  output: This is where I write the alert message and I provide some
extra information (command=%proc.cmdline connection=%fd.name).
  priority: WARNING
```

In the **condition** field, we define our abnormal behavior:

```
evt.type in (accept,listen) and (container.image!=myregistry/nginx or
proc.name!=nginx or k8s.ns.name!="load-balancer")
```

This rule detects any listening socket outside of our expected one, which is:

- The container image is myregistry/nginx.
- The listening process inside that container is nginx.
- The Kubernetes namespace is load-balancer.

Anything else should trigger an alert.

This is a good example, as it combines conditions from different sources:

- **System call events:** evt.type = listen, evt.type = mkdir, evt.type = setns, etc.
- **Docker metadata:** container.image, container.privileged, container.name, etc.
- **Process tree information:** proc.pname, proc.cmdline, etc.
- **Kubernetes namespace metadata:** k8s.ns.name, k8s.pod.name, etc.

Falco rules provide the flexibility and expressiveness needed to create accurate security rules that fully understand your operational entities.

Securing NGINX with Falco

Falco ships with a comprehensive set of out-of-the-box rules. Let's see how to enable them to detect abnormal behavior in NGINX, in particular, the execution of an `ls` command:

First, we copy the rule to our Falco installation:

```
# cp ~/falco-extras/rules/rules-nginx.yaml /etc/falco/falco_rules.
local.yaml
```

Then, restart Falco for the changes to take effect:

```
# service falco restart
```

If you run a docker image and execute a command inside of it, an alert will trigger:

```
$ docker run -d -P --name mynginx nginx
$ docker exec mynginx ls
$ cat /var/log/falco.log
11:03:13.464648222: Notice Unexpected process spawned in nginx
container (command=ls  pid=26942 user=root mynginx (id=4f94bdd87187)
image=nginx)
```

The `ls` command is not a whitelisted binary in this template.

You will probably save a considerable amount of time using these default Falco security rulesets. However, keep in mind that every version or even tag of a Docker container image is unique, and may have differences in user-defined data directories, binary paths, scripts that need to access some external port or device, or configuration. You will need to adapt the templates to your specifics before actually using them in production.

If you want to use these rulesets from the library, plus you own customization with the Helm chart, use this script to generate the Helm configuration:

```
$ git clone https://github.com/draios/falco-extras.git
$ cd falco-extras
$ ./scripts/rules2helm rules/rules-traefik.yaml rules/rules-redis.
yaml > custom-rules.yaml
$ helm install --name sysdig-falco-1 -f custom-rules.yaml stable/
falco
```

There are complete instructions, practical examples and more information about the library of Falco default rulesets in the "Protect your Docker containers using Falco security rules".

Seccomp

Seccomp enforces security by sandboxing the processes, reducing the potential attack surface by limiting the actions that a process can perform. Seccomp is a mechanism in the Linux kernel that allows a process to make a one-way transition to a restricted state where it can only perform a limited set of system calls.

While in the restricted state, if a process attempts any other system calls, it is killed via a `SIGKILL` signal. In its most restrictive mode, seccomp prevents all system calls other than `read()`, `write()`, `_exit()` and `sigreturn()`. This would allow a program to initialize and then drop into a restricted mode where it could only read from/write to already-opened files.

Here's an example of using seccomp() in strict mode:

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>

int main(int argc, char **argv)
{
        int output = open("output.txt", O_WRONLY);
        const char *val = "test";

        printf("Calling prctl() to set seccomp strict mode...\n");
        prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);

        printf("Writing to an already open file...\n");
        write(output, val, strlen(val)+1);

        printf("Trying to open the file for reading...\n");
        int input = open("output.txt", O_RDONLY);

        printf("You will not see this message--the process will be
killed first\n");
}
```

When executed, this program generates the following output:

```
$ ./seccomp_strict
Calling prctl() to set seccomp strict mode...
Writing to an already open file...
Trying to open the file for reading...
Killed
```

You can see that after the program enables strict seccomp mode, it can write to stdout, which was already open, but an attempt to open a second file results in the process being killed.

Although restrictive and undoubtedly very secure, seccomp's strict mode is... strict. You can't do much other than read/write to already open files. Network activity, starting threads and even getting the current time via `gettimeofday()` are all blocked.

What if you want to combine application sandboxing with flexible policies or per-application profiles to allow for a richer (but still limited) set of actions? For example, instead of preventing all file opens, you wanted to allow them only for specific files?

Seccomp-bpf

seccomp-bpf is an extension to seccomp that allows specifying a filter that is applied to every system call. The filter is written using BPF, which had its origins in tcpdump, but has become essentially a virtual machine implementation in the Linux kernel. The BPF program is loaded into the kernel and its execution is triggered by a system call. This execution results in a filtering decision. Based on the results of the filter, the system call can be allowed, blocked or the process can be killed.

Here's an example program using seccomp with a policy that adds `open()` to the set of allowed system calls. This example is heavily inspired and uses the `seccomp-bpf.h` header file from this very useful seccomp tutorial page.

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <linux/seccomp.h>
#include <sys/prctl.h>
#include “seccomp-bpf.h”

void install_syscall_filter()
{
        struct sock_filter filter[] = {
                /* Validate architecture. */
                VALIDATE_ARCHITECTURE,
                /* Grab the system call number. */
```

```
                EXAMINE_SYSCALL,
                /* List allowed syscalls. We add open() to the set of
                   allowed syscalls by the strict policy, but not
                   close(). */
                ALLOW_SYSCALL(rt_sigreturn),
#ifdef __NR_sigreturn
                ALLOW_SYSCALL(sigreturn),
#endif
                ALLOW_SYSCALL(exit_group),
                ALLOW_SYSCALL(exit),
                ALLOW_SYSCALL(read),
                ALLOW_SYSCALL(write),
                ALLOW_SYSCALL(open),
                KILL_PROCESS,
        };
        struct sock_fprog prog = {
                .len = (unsigned short)(sizeof(filter)/
sizeof(filter[0])),
                .filter = filter,
        };

        assert(prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0) == 0);

        assert(prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog) == 0);
}

int main(int argc, char **argv)
{
        int output = open("output.txt", O_WRONLY);
        const char *val = "test";

        printf("Calling prctl() to set seccomp with filter...\n");

        install_syscall_filter();

        printf("Writing to an already open file...\n");
        write(output, val, strlen(val)+1);

        printf("Trying to open the file for reading...\n");
        int input = open("output.txt", O_RDONLY);

        printf("Note that open() worked. However, close() will
not\n");
        close(input);

        printf("You will not see this message--the process will be
killed first\n");
}
```

When executed, this program generates the following output:

```
$ ./seccomp_policy
Calling prctl() to set seccomp with filter…
Writing to an already open file…
Trying to open the file for reading…
Note that open() worked. However, close() will not
Bad system call
```

In strict mode, seccomp kills a process when it violates the policy. However, seccomp-bpf allows a number of actions to be taken based on the results of running the policy:

- Killing the process.
- Sending the process a **`SIGSYS`** signal.
- Failing the system call and returning a (filter-provided) **`errno`** value.
- Notifying an attached process tracer (see **`ptrace()`**), if one is attached. In turn, the process tracer can skip or even change the system call.
- Allowing the system call.

Probably the most widespread use of seccomp-bpf is by docker to isolate containerized applications. Docker launches processes with a seccomp profile that disables 44 system calls, preventing their use. Examples of disabled system calls are **`mount`** (mounting filesystems), **`reboot`** (reboot the host) and **`setns`** (change namespaces to try to escape the container).

AppArmor

In its most comprehensive form, if you add policies to sandboxing, the result is Mandatory Access Control systems, like AppArmor and SELinux. These strive for system-wide enforcement of policies that control the actions and resources that each program on a system can perform. Activities outside the policies can result in logged warnings, failing the system call, or killing the process.

In AppArmor, policies are defined in profiles. A profile will apply to a given program, and will define the allowed operations and the resources it can access. Here's an example of an AppArmor profile, taken from the Ubuntu AppArmor wiki:

```
# From /etc/apparmor.d/usr.sbin.tcpdump on Ubuntu 9.04 and
https://wiki.ubuntu.com/AppArmor#Example_profile

#include <tunables/global>

/usr/sbin/tcpdump {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  #include <abstractions/user-tmp>

  capability net_raw,
  capability setuid,
  capability setgid,
  capability dac_override,
  network raw,
  network packet,

  # for -D
  capability sys_module,
  @{PROC}/bus/usb/ r,
  @{PROC}/bus/usb/** r,

  # for -F and -w
  audit deny @{HOME}/.* mrwkl,
  audit deny @{HOME}/.*/ rw,
  audit deny @{HOME}/.*/** mrwkl,
  audit deny @{HOME}/bin/ rw,
  audit deny @{HOME}/bin/** mrwkl,
  @{HOME}/ r,
  @{HOME}/** rw,

  /usr/sbin/tcpdump r,
}
```

This profile begins by stating that the rules will apply to `/usr/bin/tcpdump`, and then declares what `tcpdump` will be able to do, including:

Permitted linux capabilities: `net_raw`, `setuid`, `setgid`, `dac_override`

Permitted network operations: `raw`, `packet`

Allowed files: `/proc/bus/usb` and its children, files below `$HOME`, and `/usr/bin/tcpdump` itself.

Disallowed files: any dot-files or dot-directories below `$HOME` and anything below `$HOME/bin`. Audit deny also indicates that attempts to access these files should be logged.

SELinux

Although conceptually similar to AppArmor, SELinux significantly differs on its implementation. While AppArmor profiles are oriented around processes, SELinux policies are much more complex. They apply separately to actors, actions and targets, with a whole middleware of types defining the policies for each in different places. It's because of this that SELinux uses a combination of policy files and file attributes.

It's not within the scope of this guide to explain all of those SELinux concepts here, but we invite you to check this tutorial for a longer but more thorough introduction.

Having said that, let's briefly look at the set of policies for /usr/sbin/tcpdump to see how it's allowed to access raw sockets (the equivalent to the **net_raw** capability in the previous AppArmor example).

As said before, SELinux uses file attributes. Once configured in a system, you can use **ls -Z** to show the security context information of a file. For tcpdump, it would be something like this:

```
$ ls -Z /usr/sbin/tcpdump
-rwxr-xr-x. root root system_u:object_r:netutils_exec_t:s0 /usr/sbin/
tcpdump
```

Take a look at the fourth column. There, the security context information includes a user (**system_u**), a role (**object_r**), a type (**netutils_exec_t**) and a level (**s0**).

In this case, it means that when someone runs tcpdump, the running process changes to a security context with the **netutils_t** type.

We can use the SELinux search tool **sesearch**, to show the domain transitions involving a given entrypoint (**netutils_exec_t** for tcpdump), when related to files:

```
$ sesearch -t netutils_exec_t -c file -p entrypoint -Ad
Found 1 semantic av rules:
   allow netutils_t netutils_exec_t : file { ioctl read getattr lock
execute execute_no_trans entrypoint open } ;
```

If we want to see what linux capabilities tcpdump has, or any other program with the context **netutils_t**, we can use sesearch again:

```
# sesearch -t netutils_t -c capability -Ad
Found 1 semantic av rules:
   allow netutils_t netutils_t : capability { chown dac_read_search
setgid setuid net_admin net_raw sys_chroot } ;
```

Seccomp vs AppArmor and SELinux

The distinction between AppArmor, SELinux and Seccomp can be fuzzy at times. They all rely on the same mechanisms–kernel-level interception/filtering of system calls, driven by per-process policies. However, there are some distinctions.

The policy languages used by AppArmor and SELinux differ from each other with respect to ease of use and specific terminology, but are generally richer and more complex than seccomp. AppArmor and SELinux allow for defining actors (generally processes), actions (reading files, network operations), and targets (files, IPs, protocols, etc.), where seccomp is limited to a simple list of system calls and arguments.

Additionally, seccomp is voluntary. Processes have to willingly drop into a restricted state by calling `prctl(PR_SET_SECCOMP, …)`, while in Mandatory Access Control systems, the policy is defined and loaded before a process is run.

Challenges implementing abnormal behavior detection

Whatever runtime security solution you choose, there are several issues you'll eventually face. For example, defining what is considered abnormal behavior for each of your containers is going to take some time. And that work will require some experimenting and tweaking to ensure they aren't missing edge cases of triggering false positives.

Here are some key resources and approaches that can save you a lot of work, and help you be ready for production sooner.

Out of the box rules

Why write each policy from scratch? Most of them are built using the same base rules, like "Don't access files below $HOME" or "Don't spawn terminals in containers".

When evaluating a runtime security solution, check how many of those rules are available out-of-the-box. Are they well maintained? Can you customize them? Is it easy to find what you need?

The cloud native security hub is a community-driven repository for Falco rules, among other security-related resources, for the most common cloud applications.

Our Sysdig Secure has a comprehensive library of customizable Falco rules. They are tagged by type of resource, application and compliance standard, so you can easily find what you need. Most importantly, they are actively maintained and updated to protect you from newly discovered attacks.

Automated policy creation using machine learning

Most containerized applications have a predictable behavior, as they always read the same files or connect to the same network endpoints. Machine learning is great at detecting these kinds of patterns.

When evaluating your runtime security solution, look for machine learning features. Will they help you save time, or are they just a publicity stunt?

For example, Sysdig Secure offers image profiling. After some time learning the expected behavior of a given image, it can generate a base security policy for the image that you can later fine tune.

Avoiding false positives, tuning Falco rules.

False positives are unavoidable. You might have been too restrictive, forgotten some edge cases, or a new version of a container changed its behavior and you have to update your policies accordingly.

Although they aren't harmful, it's considered best practice to mitigate these events if possible. After all, they can cause considerable noise on the notification channels and steal attention away from the real incidents.

Identifying false positives generally involves the following steps:

1. Identify the most commonly occurring policy events.
2. Determine if the events are unique to the environment.
3. Address the false positives via rule changes.
4. Address the false positives via policy scope.
5. Disable the policy/policies that trigger the events.

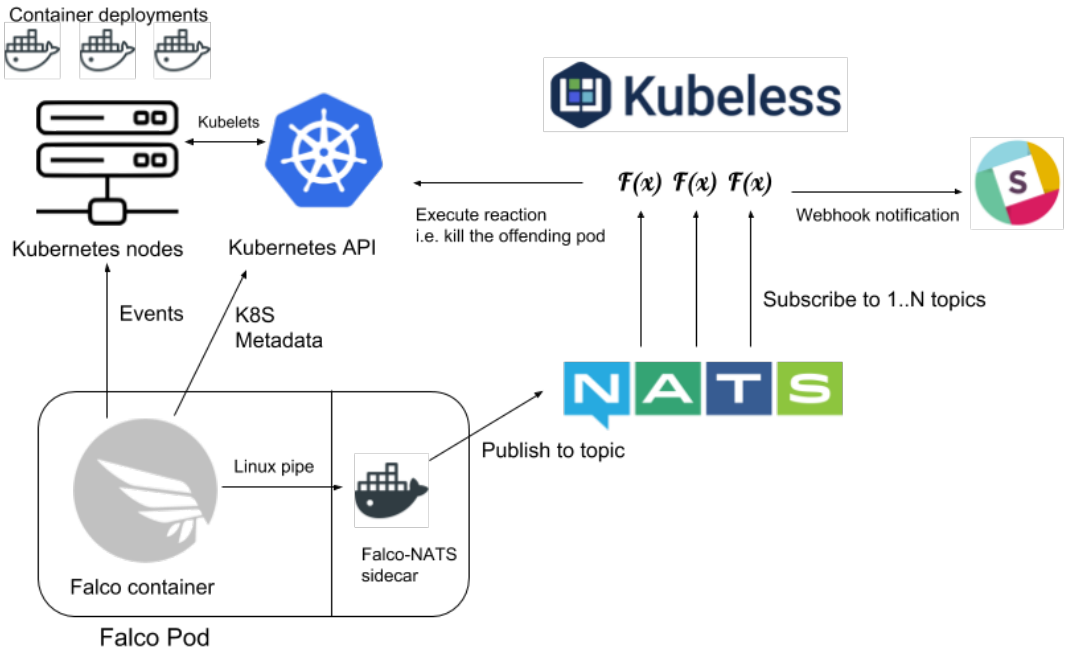
To get a concrete example of the process, check out this guide on how to identify false positives on Falco rules with Sysdig Secure.

Threat blocking and Incident remediation with open source tools

We've seen how Falco provides deep visibility into what's happening inside containers. It also has Docker and Kubernetes native support, so it's able to tell not only when a security incident happened, but also provide the appropriate metadata to investigate the issue.

However, beyond detection, you also need to react fast. Because containers have short lifespans, it's essential to find the culprit before the container causes any real damage, simply disappears or moves somewhere else. Automation is more important than ever, and writing response procedures as code playbooks is the best approach.

There are open source components that can be tied together to build a Kubernetes security stack:



- Falco: Provides visibility and **abnormal behavior detection**. Its default ruleset library, the ones you can find in the cloud native security hub, and other resources offered by its community, provide you with a good starting point to protect the most popular images including kube-system components, Nginx, HAproxy, Apache, Redis, MongoDB, Elastic, PostgreSQL, etc.
- Nats: A **messaging broker** that can receive Falco security alerts and make them available to other parties.
- Kubeless: A Function as a Service framework for Kubernetes. It will subscribe and listen for events on NATS, executing different **playbooks for incident response and attack mitigation, as well as automated actions written as code**. For example, Kubeless will post a notification to Slack, connect to the Kubernetes API server to stop the pod, or create a Network Policy that will isolate the pod from the network.

Additionally, you could implement **logging, audit and reporting** integrating Falco with third party tools like Fluentd, Elastic and Kibana.

For installation and setup instructions, as well as examples on creating Kubeless functions, check out this "How to implement an open source container security stack" article.

Find out how the Sysdig Secure DevOps Platform can help you and your teams confidently run cloud-native apps in production. Contact us for additional details about the platform, or to arrange a personalized demo.

www.sysdig.com