Sysdig

Kubernetes Security Guide.

Adopting a cloud-native architecture and a DevOps approach to development demands changes to your IT architecture. Inside you'll find best practices for implementing Kubernetes security in your organization.

eBOOK

Kubernetes Security Guide.

Kubernetes Security Guide.

Introduction

In this Kubernetes security guide we cover the most significant aspects of implementing Kubernetes security best practices.

Kubernetes security, like monitoring or building a CI/CD pipeline is now a must as the Kubernetes platform is quickly becoming the defacto standard for modern containerized deployments.

To keep up with this rapid change, traditional security processes need to be updated for Kubernetes; legacy security tools that were not built for containers cannot keep up with cloud-native transformation. Containers provide immense benefit in the development and deployment of applications, enabling substantial portability and isolation. However, many of the same attributes that make containers so valuable also make it harder to understand what's happening inside, monitor, and secure them. At higher level, microservices allow for faster application development and orchestration tools like Kubernetes underpin deployment and enable dynamic scale of those apps. Of course, with all these moving parts, organizations must take a more dynamic security approach.

Many organizations are adopting new security best practices and implementing DevSecOps processes, where everyone is responsible for security and security is implemented from early development stages into production through the entire software supply chain. This also known as Continuous Security.

Moving Kubernetes to production comes with new infrastructure layers, new components, new procedures and therefore new security processes and tools. This security guide will help you to implement Kubernetes security, focused on Kubernetes-specific security features and configuration. We also highlight some additional tools that will go beyond what Kubernetes can do.

Kubernetes Security Guide.

Chapter 1

Understanding Kubernetes RBAC and TLS certificates

Two fundamental components of a Kubernetes security authentication and authorization involve RBAC and TLS certificates. In this chapter you'll learn how to enable and configure RBAC permissions, manage credentials for users and services, and rotate TLS certificates and security tokens.

Kubernetes RBAC Overview

Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users.

With RBAC you define the actions (i.e. get, update, delete) that Kubernetes subjects (human users, software, kubelets) are allowed to perform over Kubernetes entities (pods, secrets, nodes).

Kubernetes RBAC uses the "rbac.authorization.k8s.io" API group to drive authorization decisions. Before getting started, it is 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, an application, a team or a customer.

Kubernetes Security Guide.

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 will see on the RBAC examples below, but there is 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, 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 like 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 similar to a security profile. A Role is always confined to a single namespace, while a ClusterRole is cluster-scoped.

- Before designing your security policy, take into account that Kubernetes RBAC permissions are explicitly additive, 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 bounded to a certain namespace, ClusterRoleBindings are cluster-global.

Kubernetes Security Guide.

Getting Started

The first step is to define a Role that grants the verbs ["get", "watch", list"] over any pod resource, only in the "default" namespace. Then, create a RoleBinding that grants the permissions defined in "pod-reader" to the user "jane".

```
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
```

Kubernetes Security Guide.

Configuring Kubernetes RBAC Security

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

You need to look for this flag:
`--authorization-mode=Node,RBAC`

While the API server includes 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.

Creating Kubernetes Users and ServiceAccounts

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.

By using service-specific service accounts rather than default serviceAccounts you can have granular control the API access granted to any software entity inside your cluster.

When it comes to users, 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 need to carry out.
- Select Role and RoleBinding to their cluster counterparts, when possible. It is much easier to control security when is bounded to independent namespaces.
- Avoid the use of wildcards ["*"] when defining access to resources or actions over these resources.

Step-by-Step Instructions for Creating a Kubernetes serviceAccount

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 some updates using webhooks.

You just need 'read-only' access and just want 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 not able 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.

Kubernetes Security Guide.

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
```

Kubernetes Security Guide.

```
subjects:
- kind: ServiceAccount
   name: flask-backend
   namespace: flask
roleRef:
   kind: Role
   name: flask-backend-role
   apiGroup: rbac.authorization.k8s.io
---
kind: Deployment
apiVersion: extensions/v1betal
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
```

Next, you can check that the permissions are working as you expect with the `kubectl auth` command, which can query access for verbs, subjects and 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:flask:flask-backend
yes

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

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.

Step-by-Step Instructions for Creating a Kubernetes User

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 take a 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"
```

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

Now, you have to 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)
```

Kubernetes Security Guide.

Let's repeat this process for a second user, so we can show how to assign 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.

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, that's why we are going to just 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. This is often overlooked and never-expiring tokens are shared between administrators for convenience reasons.

We are going to cover 3 scenarios:

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

It is worth mentioning 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 RBAC. You cannot re-use the username until the certificate has expired
- recreate the certificate authority and issue new certificates to all active users
- consider OIDC (OpenID Connect) as an alternative authentication method

Kubernetes Kubelet TLS Certificate Rotation

The kubelet is a critical component from the security point of view since it serves as the bridge between the node operating system and the cluster logic.

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 Security Guide.

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
```

If you *describe* the new secret, you will be able to see the new token string. Please note that existing pods using this serviceAccount will continue using the old (invalid) token, 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=/ho
```

Summary

In this section we have learned how to enable and configure RBAC permissions in Kubernetes, use credentials for users and services and rotate TLS certificates and tokens, covering the authentication and authorization part of Kubernetes security. Next will continue with Kubernetes Security Context, Pod Security Policies and Kubernetes Network Policy.

Chapter 2

Implementing security at the pod level: Kubernetes Security Context, Kubernetes Security Policy and Kubernetes Network Policy

Configuring Kubernetes Security with 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 (like filling out empty parameters with default values), and can also be used to enforce further security checks.

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

```
--admission-control=Initializers, NamespaceLifecycle, LimitRanger, ServiceAccount,
PersistentVolumeLabel, DefaultStorageClass, DefaultTolerationSeconds, NodeRestriction, 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 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

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 Anchore scanned images. Check out this recommended set of admission controllers to run depending on your Kubernetes version.

Kubernetes Security Guide.

Configuring 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
    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.

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 as you see here:

```
$ 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 *fsGroupparameter*).

Security Enhanced Linux (SELinux)

You can assign SELinuxOptions objects using the *seLinuxOptions* field. Note that SELinux module needs to be loaded on the underlying Linux nodes for this 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.

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

```
$ 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.

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.

Kubernetes Security Guide.

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 Guide.

Kubernetes Pod Security Policy

Pod Security Policies or PSP are 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
Control Aspect	**Field Names**
The user and group IDs of the container	runAsUser, supplementalGroups
Restricting escalation to root privileges	allowPrivilegeEscalation, defaultAllow-PrivilegeEscalation
Linux capabilities	defaultAddCapabilities, requiredDrop-Capabilities, 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

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).

Kubernetes Security Guide.

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"
```

You need the *PodSecurityPolicy* admission controller enabled in your API server to enforce these policies.

If you plan to enable *PodSecurityPolicy*, make sure first that you 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
                                                                                    READONLY
NAME                          PRIV   CAPS    SELINUX    RUNASUSER FSGROUP    SUPGROUP    ROOTFS  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]
```

Kubernetes Security Guide.

In case 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:
  - '*'

$ kubectl create -f psp.yaml
podsecuritypolicy "example" created

$ kubectl get psp
                                                                READONLY
NAME      PRIV  CAPS  SELINUX   RUNASUSER  FSGROUP    SUPGROUP   ROOTFS     VOLUMES
example   true  []    RunAsAny  RunAsAny   MustRunAs  RunAsAny   false      [*]
```

If you try to create new pods without the *runAsUser* directive you will get:

```
$ kubectl create -f https://raw.githubusercontent.com/mateobur/kubernetes-securityguide/master/readonly/flask-ro.yaml
$ kubectl describe pod flask-ro
...
Failed       Error: container has runAsNonRoot and image will run as root
```

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.

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.

To show how these network policies work, let's use the Kubernetes example scenario guestbook:

```
$ 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 use these logical groupings to configure your network policy, abstracting away concepts like IP address or physical node, that wouldn't work here as Kubernetes can change those 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
```

That you can also find 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)
```

As we mentioned before, note that this policy will be enforced even if the pods migrate to another node or they are scaled up/down.

Kubernetes Resource Allocation Management

Resource limits are most typically established to avoid unintended saturation due to design limitations or software bugs, but they can also protect you from 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. Kubernetes makes sure that the actual resource consumption never goes over the configured limits.

You can run a quick example from the resources/flask-resources.yamlrepository 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

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 will 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 bounded to a namespace context.

Summary

This chapter explained how to configure security at the pod level using Kubernetes orchestration capabilities, as well as manage Kubernetes resource allocation. Now you know how to use Kubernetes security context, pod security policy and network policy resources to define the container privileges, permissions, capabilities and network communication rules.

The next chapter covers securing Kubernetes components.

Chapter 3

Securing Kubernetes components (kubelet, etcd or your registry)

This chapter explains how to secure sensitive Kubernetes components and common external resources such as the Docker registry. You will learn how to secure the Kubelet, the etcd cluster and pull from trusted Docker repositories.

Kubelet Security

The kubelet is a fundamental piece of any Kubernetes deployment. It is 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 then 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 -
- NodeRestriction admission controller
- RBAC to access the kubelet API resources

Below are descriptions of each and tips for verifying they are working as expected.

Kubelet Security: Access to the Kubelet API

The kubelet security configuration parameters are usually 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 remain 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).
- The kubelet has a `` `--client-ca-file `` flag, providing a CA bundle to verify client certificates.
- The `` `--authorization-mode `` is NOT set to *AlwaysAllow*, as the more secure *Webhook* mode will delegate authorization decisions to the Kubernetes API server.
- The `--read-only-port` is set to 0 to avoid unauthorized connections to the read-only endpoint (optional).

Kubelet Security: Kubelet Access to Kubernetes API

As mentioned in Chapter #1, the level of access granted to a kubelet is determined by the NodeRestriction Admission Controller for 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. 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,ServiceAccount,Persistent
VolumeLabel,DefaultStorageClass,DefaultTolerationSeconds,NodeRestriction,ResourceQuota
```

Accessing the Kubelet API with Curl

Typically, just the Kubernetes API server will need to use the kubelet REST API. 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.

```
# 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.

Securing Kubernetes etcd

etcd is a key-value distributed database that persists Kubernetes state. The *etcd* configuration and upgrade 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."

These restrictions can be enforced using regular Linux firewalling (iptables/netfilter, etc), with run-time access protection, or with PKI-based authentication and parameters.

Run-time Access Protection

An example of run-time access protection could be making sure that the *etcd* binary only reads and writes from a set of configured directories or network sockets, any run-time access that is not explicitly whitelisted will raise an alarm.

Using Sysdig Falco it will look similar to this:

```
- macro: etcd_write_allowed_directories
  condition: evt.arg[1] startswith /var/lib/etcd

- rule: Write to non write allowed dir (etcd)
  desc: attempt to write to directories that should be immutable
  condition: open_write and not etcd_write_allowed_directories
  output: "Writing to non write allowed dir (user=%user.name command=%proc.cmdline file=%fd.name)"
  priority: ERROR
```


```
- macro: etcd_write_allowed_directories
  condition: evt.arg[1] startswith /var/lib/etcd

- rule: Write to non write allowed dir (etcd)
  desc: attempt to write to directories that should be immutable
  condition: open_write and not etcd_write_allowed_directories
  output: "Writing to non write allowed dir (user=%user.name command=%proc.cmdline file=%fd.name)"
  priority: ERROR
```

Kubernetes Security Guide.

PKI-based authentication for etcd

Ideally, you should create 2 sets of certificate and key pairs that are going to be used exclusively for etcd. One pair will verify member to member connections and the other one Kubernetes API to etcd connections. Conveniently, the etcd project provides these scripts to help you generate the certificates.

Once you have all 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=<path>`
  - `--key-file=<path>`
  - `--client-cert-auth`
  - `--trusted-ca-file=<path>` (can be the same you used for peer to peer)
- Kubernetes API server parameters:
  - `--etcd-certfile=k8sclient.cert`
  - `--etcd-keyfile=k8sclient.key`

Using a Trusted Docker Registry

If you don't specify otherwise, Kubernetes will pull Docker images from the public registry Docker Hub. This is fine for testing or learning environments, but not ideal for production, since most organizations want to keep images and content private.

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. Appart 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": "1",
   "data" : {
       ".dockercfg": "eyJyZWdpc3RyeS5sb2NhbCI6eyJ1c2VybmFtZ
              SI6ImpvaG5kb3ciLCJwYXNzd29yZCI6InNlY3JldHBh
   }
   "kind": "Secret",
   "metadata": {
       "creation Timestamp":"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.

Using 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.

Summary

This chapter provided tips on how to secure sensitive Kubernetes components and common external resources such as the Docker registry. Now you know how to secure the Kubelet, the etcd cluster and pull from trusted Docker repositories.

In the next chapter will offer plenty of practical examples and use case scenarios covering Kubernetes runtime threat detection.

Chapter 4

Hardening Kube-system Components with Sysdig Secure Security Policies

In this chapter we cover best practices for implementing run-time security on the kube-system components (kubelet, api-server, scheduler, kubedns, etc.) deployed in Docker containers.

One of the main sources of concern for companies approaching the container paradigm has traditionally been **security**. It's a radical infrastructure switch after all, and certain level of caution is perfectly healthy.

Kubernetes devs are aware of this and the platform has improved leaps and bounds in this respect. Work on the RBAC API, integrated secrets vault or certificate rotation mechanisms are the latest examples of this effort.

Much of Kubernetes security content available focuses tuning cluster configuration parameters, restricting user privileges, and secret management. But what if somebody finds a way to bypass those? Or a software component does something unexpected because it suffers from a bug or security vulnerability that your static scanning will never catch?

Run-time security provides an extra layer of protection for those times malicious users or software behave in a way you didn't prepare for. Now let's create and test Kubernetes run-time security policies using Sysdig Secure, our container native security and forensics product.

kube-system Security: Core Components

Below is a list of the relevant kube-system components discussed in this chapter:

kube-apiserver

The central communications hub of the cluster. Provides REST endpoints to interact with the other cluster entities and stores the distributed state in the etcd backend.

etcd

The database backend where the cluster configuration, state, and related information persists.

kube-controller-manager

This component implements the main control loo. In other words, it observes the differences between the current and desired cluster states and performs the changes needed to move towards desired state. When you launch a *ReplicationController*, it is included in this component.

kube-scheduler

Watches newly created pods that have no node assigned, and selects a node for them to run on. From version 1.6 onwards, you can plug in your own custom Kubernetes scheduler.

kube-dns

An internal cluster DNS server. Kube DNS will automatically configure the registers for Kubernetes namespaces, services, and pods. It enables the pods to easily locate other services in the cluster.

kubelet

The cluster agent that runs on every Kubernetes node. The kubelet launches the pods using the available container engine (Docker, rkt, etc) and periodically checks and reports pod status.

kube-proxy

Another service that runs on every node, providing the necessary network translation between service endpoints and pods.

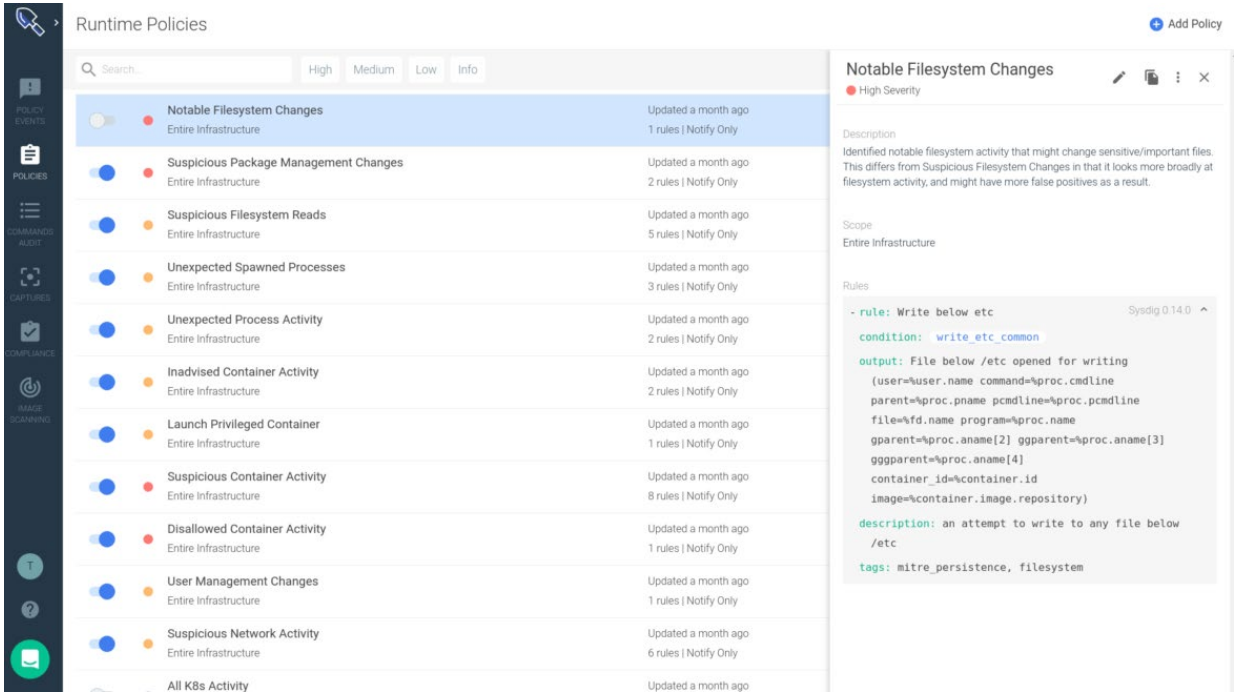
The kubelet and kube-proxy run as processes directly in the Kubernetes nodes, while the other components typically run as cluster Docker containers inside the kube-system namespace.

Kubernetes Security by Default with Sysdig Secure

We are going to produce a security ruleset for the components listed previously, using a whitelisting approach (explicitly declaring valid entities, banning everything else). The container/microservice paradigm goes very well with this approach because containers are already minimal and predictable by design.

Sysdig Secure

Sysdig Secure, part of the Sysdig Container Intelligence Platform, is designed to provide container run-time security and forensics for enterprises. Sysdig Secure's deep container visibility provides insight into what's happening inside the containers and leverages key orchestration technologies such as Kubernetes, Docker, OpenShift, Amazon ECS to bring in metadata and apply rules from a service and application perspective.



The Sysdig Secure rule syntax is the same used in Sysdig Falco, and is documented here.

kube-system Process Security

One of the easiest and most effective whitelists that you can configure is the list of allowed processes. This task that would be somewhat tedious and error prone in a classical all-purpose server, but it's quite straightforward for microservices.

For example, let's see the processes on our etcd service container:

```
$ kubectl exec -it etcd --namespace=kube-system sh
/ # ps aux
PID   USER  TIME   COMMAND
    1 root  34:42  etcd --listen-client-urls=http://127.0.0.1:2379 --advertise-client-urls=http://127.0.0.1:2379 --data-dir=/var/lib/etcd
   41 root  0:00   sh
   45 root  0:00   ps aux
```

Note that anything beyond the etcd process would be extremely suspicious.

Let's take a look to the API server:

```
$ kubectl exec -it kube-apiserver --namespace=kube-system sh
/ # ps aux
PID   USER        TIME     COMMAND
    1 root        117:34   kube-apiserver --insecure-port=0 --allow-privileged=true --requestheader-username
                           -headers=X-Remote-User --service-cluster-ip-range=10.96.0.0/12 --proxy-client
   40 root        0:00     sh
   44 root        0:00     ps aux
```

Similarly, we see a single process running here.

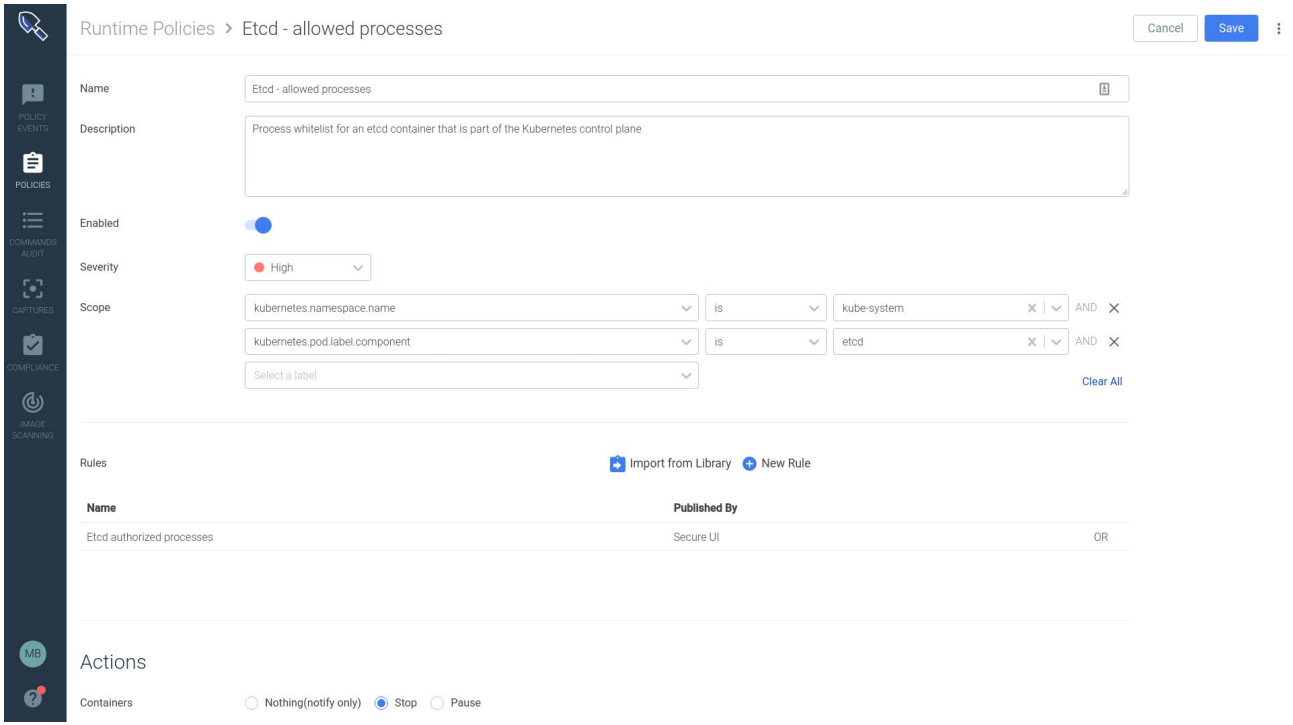
Kubernetes Security Guide.

Thus, you can write a Sysdig Secure list and rule similar to this one:

```
- list: etcd_authorized_processes
  items: [etcd]

- rule: Etcd allowed processes
  desc: Whitelist of authorized etcd processes
  condition: spawned_process and not proc.name in etcd_authorized_processes
  output: Unauthorized process (%proc.cmdline) running in (%container.id)
  priority: ERROR
```

And the following Kubernetes security policy:



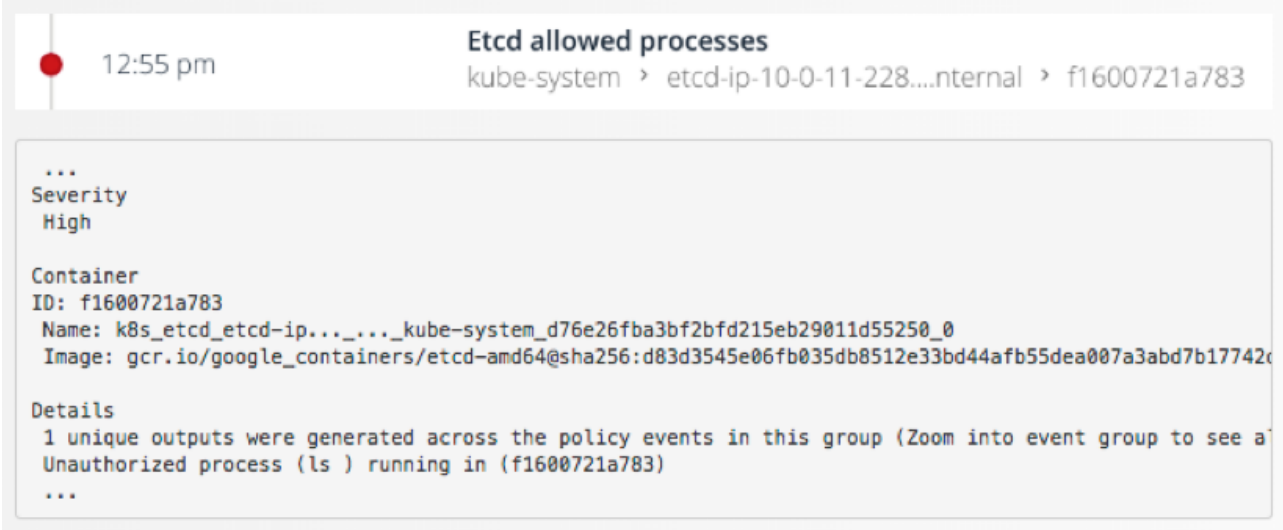
Here you are restricting the scope of this rule to `kubernetes.namespace.name=kube-system` and Docker containers with the label `component=etcd`. This security violation is critical enough to stop the container immediately (see Actions) and of course, in a real scenario you would configure several notification channels such as email or Slack, and you can also go for webhooks, VictorOps, PagerDuty, etc.

Let's trigger it the policy we just created, opening a shell in the etcd container and running any process:

```
/ # ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
/ # user@localhost:~/kubernetes$
```

Notice the automatic expulsion from container. The container where the command was running has been inmediately killed by Sysdig.

You will get the event in your Sysdig Secure stream and clicking on it you will be able to see the detail (just including the sections relevant for this example below):



For the sake of brevity, we are only going to include the YAML rules for the next sections, as you have seen, creating and testing the corresponding Sysdig Secure policies is a straightforward process.

Trusted containers in Kubernetes kube-system

In conjunction with other tools that run your own images registry, Sysdig Secure offers an additional layer of run-time security against the use of untrusted containers. This is especially important for the kube-system namespace where the allowed list of pods is pretty small and immutable.

The following rule will automatically detect (and kill) any container that doesn't come from any of the allowed images:

```
- macro: allowed_containers
  condition:  (container.image startswith gcr.io/google_containers/etcd-amd64 or
              container.image startswith gcr.io/google_containers/k8s-dns-dnsmasq-nanny-amd64 or
              container.image startswith gcr.io/google_containers/k8s-dns-kube-dns-amd64 or
              container.image startswith gcr.io/google_containers/k8s-dns-sidecar-amd64 or
              container.image startswith gcr.io/google_containers/kube-apiserver-amd64 or
              container.image startswith gcr.io/google_containers/kube-controller-manager-amd64 or
              container.image startswith gcr.io/google_containers/kube-proxy-amd64 or
              container.image startswith gcr.io/google_containers/kube-scheduler-amd64)

- rule: Unallowed container running in kube-system namespace
  desc: Unallowed container running in kube-system namespace
  condition: container and not allowed_containers
  output: Unallowed container running in kube-system namespace (%container.info)
  priority: ERROR
```

Kubernetes Security Guide.

Detect Processes Attempting to Access a Secret File After Boot

Kubernetes has a secrets mechanism to securely initialize pods with artifacts like private keys, passwords, tokens, etc. Generally, a pod will need to access its secrets during start up. You can detect a long running process that attempts to access a secret file.

Assuming you mount your secrets under `/etc/secrets` (replace with your configured path) this rule will detect and react when a process in a container reads those sensible files:

```
- macro: proc_is_new
  condition: proc.duration <= 5000000000

- rule: Read secret file after startup
  desc: >
    an attempt to read any secret file (e.g. files containing user/password/authentication
    information) Processes might read these files at startup, but not afterwards.
  condition: fd.name startswith /etc/secrets and open_read and not proc_is_new
  output: >
    Sensitive file opened for reading after startup (user=%user.name
    command=%proc.cmdline file=%fd.name)
  priority: WARNING
```

Detect Pods Attempting to Connect to Their Local Kubelet

In a Kubernetes cluster, the API server talks to the kubelets and these set up the pods in each node. Usually, the pods shouldn't connect to the kubelet or try to scrape its metrics unless you have a service that explicitly needs that. Let's fire an alarm to see if we can detect this behavior.

The API server connects to the kubelet service using port 10250. 10255 and 10248 (now deprecated) are read-only health check and stats ports. To detect any pod connecting to the local kubelet, we will use a rule like this:

```
- macro: kubelet_ports
 condition: fd.sport in (10248, 10250, 10255)

- rule: Pod connecting to kubelet
 desc: A pod is opening an outbound network connection to the local kubelet
 condition: outbound and fd.sip = "127.0.0.1" and kubelet_ports
 output: Pod connecting to local kubelet (command=%proc.cmdline %container.info connection=%fd.name)
 priority: WARNING
```

Kube-system User Security Policies

Another common attack symptom could be the modification of attributes and permissions of the default users to gain access to privileged information (forging an URL for example).

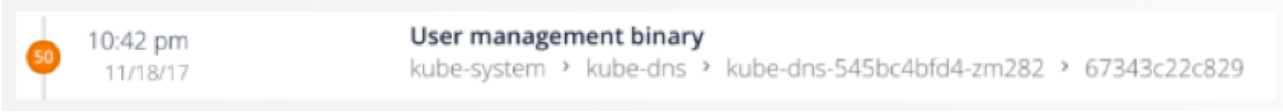
These kube-system components do not need to modify or create any new user, you can detect and alert if any container tries to perform these actions.

Conveniently enough, there are already lists and macros for the user management tools. The following rule makes it quite easy:

```
- rule: User management operations
  desc: User created or modified inside a pod
  condition: proc.name in (user_mgmt_binaries) and not proc.pname in (cron_binaries, systemd, run-parts)
  output: User management binary run on container (command=%proc.cmdline %container.info)
  priority: ERROR
```

Let's test this policy:

```
$ kubectl exec -it kube-dns-545bc4bfd4-zm282 --namespace=kube-system sh
Defaulting container name to kubedns.
Use 'kubectl describe pod/kube-dns-545bc4bfd4-zm282' to see all of the containers in this pod.
/ # passwd postgres
```



In this case the default ruleset of Sysdig Secure already detects any write below /etc, offering extra protection in the event an attacker tries to change the valid sources of users, for example modifying */etc/nsswitch.conf* ;-).

Kubernetes Security Guide.

Kubernetes File System Security: Write-allowed Directories

Most container directories are generally static and read-only. The list of write-allowed directories should be easy to define and any violation immediately detected. This is an example to detect writes outside the allowed path:

```
- macro: etcd_write_allowed_directories
  condition: evt.arg[1] startswith /var/lib/etcd

- rule: Write to non write allowed dir (etcd)
  desc: attempt to write to directories that should be immutable
  condition: open_write and not etcd_write_allowed_directories
  output: "Writing to non write allowed dir (user=%user.name command=%proc.cmdline file=%fd.name)"
  priority: ERROR
```

Kubernetes Network Security: Processes Opening a Listening Port

Another interesting whitelist is the list of processes that can open a listening connection. For example, in this rule we only let kube-proxy to open listening ports:

```
- list: kube_proxy_port_processes
  items: [kube-proxy]

- rule: Unauthorized process opened a port
  desc: A kube_proxy process tried to open a port and is not whitelisted
  condition: evt.type=listen and not proc.name in (kube_proxy_port_processes)
  output: Non-whitelisted process opened a port (command=%proc.cmdline connection=%fd.name)
  priority: WARNING
```

Kubernetes Network Security: Processes Opening an Outbound Connection

The list of processes that can initiate a connection from your kube-system pods should be fairly limited as well. For example here, only kube-apiserver can initiate an outbound connection:

```
- list: kube_apiserver_outbound_processes
  items: [kube-apiserver]

- rule: Unauthorized process opened an outbound connection
  desc: A kube-apiserver process tried to open an outbound connection and is not whitelisted
  condition: outbound and not proc.name in (kube_apiserver_outbound_processes)
  output: Non-whitelisted process opened an outbound connection (command=%proc.cmdline connection=%fd.name)
  priority: WARNING
```

Let's try it:

```
$ kubectl exec -it kube-apiserver --namespace=kube-system sh / # wget www.google.com
```

Alert description text:

```
Non-whitelisted process opened an outbound connection (command=wget www.google.com
connection=10.0.11.228:41636->172.217.13.68:80)
```

Kubernetes Network Security: Detecting NodePort Endpoints

A human error or maybe a malicious user can configure a service as type *Nodeport*, thus bypassing the firewalls and other security measures that you have configured for your load balancers:

30000 to 32767 is the default port range in Kubernetes for NodePort services. This is a rule to detect that:

```
- rule: Unexpected NodePort connection
  desc: A service has been declared using type NodePort
  condition: (outbound or inbound) and fd.sport >= 30000 and fd.sport <= 32767
  output: A service is using a NodePort connection (command=%proc.cmdline connection=%fd.name)
  priority: WARNING
```

Kubernetes Security Guide.

Default Sysdig Secure Rules and Policies

In addition to the security rules and policies we have listed here, the default Sysdig Secure policies had been designed to protect containers from several common threats and attacks. Most of them will be useful to protect your kube-system out of the box, or with some minor customizations.

You will find policies including:

Modify binary dirs

an attempt to modify any file below a set of binary directories

Change thread namespace

an attempt to change a program/thread namespace (commonly done as a part of creating a container) by calling setns

Launch Sensitive Mount Container

detect the initial process started by a container that has a mount from a sensitive host directory (i.e. /proc). Exceptions are made for known trusted images.

Launch Privileged Container

detect the initial process started in a privileged container. Exceptions are made for known trusted images.

System procs network activity

any network activity performed by system binaries that are not expected to send or receive any network traffic.

This input file will auto generate the mentioned whitelisting rules for processes, write-allowed directories, process allowed to start outbound connections and process allowed to open listening ports.

This simple python script automatically generate a ruleset that you can directly copy and paste as custom rules for Sysdig Secure

```
$ ./generate-secure-rules.py input_rules.yaml > output_rules.yaml
```

Summary

Strict run-time security for your kube-system pods is an effective mechanism against any attack that has already bypassed your existing security measures or that managed to exploit a new vulnerability.

Get a free Sysdig Secure trial and protect your containers and microservices with run-time security or learn more about Sysdig Falco (single host, command line only), the open source relative of Sysdig Secure.

Kubernetes Security Guide.

Auto Generate Kubernetes Security Rules and Policies

Creating all these whitelisting rules by hand can be laborious, Fortunately, you can auto-generate the rules that will be used as base for your policies using the following YAML format:

```
- podname: etcd
  proc: [etcd]
  write_dir: [/var/lib/etcd]
  outbound_proc: [etcd]
  listen_proc: [etcd]

- podname: kube_apiserver
  proc: [kube-apiserver]
  write_dir: false
  outbound_proc: [kube-apiserver]
  listen_proc: [kube-apiserver]

- podname: kube_dns
  proc: [dnsmasq, dnsmasq-nanny, sidecar, kube-dns]
  write_dir: [/var/run/dnsmasq.pid, /dev/null]
  outbound_proc: [kube-dns]
  listen_proc: [kube-dns, sidecar, dnsmasq]

- podname: kube_controller
  proc: [kube-controller-manager]
  write_dir: false
  outbound_proc: [kube-controller-manager]
  listen_proc: [kube-controller-manager]

- podname: kube_scheduler
  proc: [kube-scheduler]
  write_dir: false
  outbound_proc: [kube-scheduler]
  listen_proc: [kube-scheduler]
```