**Getting Started with Secrets**

Learn an introduction about Kubernetes Secrets.

**We'll cover the following**

* [Understanding secrets](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3jNy1KOMwRr#Understanding-secrets)

**Understanding secrets**

We cannot treat all information equally. Sensitive data needs to be handled with additional care. Kubernetes provides an additional level of protection through Secrets.

A Secret is a relatively small amount of sensitive data. Some of the typical candidates for Secrets would be passwords, tokens, and SSH keys.

Kubernetes Secrets are very similar to ConfigMaps. If you compare the differences in the syntax, you’ll notice that there are only a few (if any). Conceptually, both ConfigMaps and Secrets are, more or less, the same. If you are familiar with ConfigMaps, you should have no trouble applying that knowledge to Secrets.

We already used Secrets without even knowing. Every Pod we created so far had a Secret mounted automatically by the system.

We’ll start by exploring auto-generated Secrets and proceed to produce some ourselves.

**Exploring Built-In Secrets**

Explore built-in Kubernetes Secrets.

**We'll cover the following**

* [Creating Jenkins objects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEzZ4wmAO4P#Creating-Jenkins-objects)
* [The Built-In Secrets](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEzZ4wmAO4P#The-Built-In-Secrets)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEzZ4wmAO4P#Try-it-yourself)

**Creating Jenkins objects**

We’ll create the same Jenkins objects we defined earlier.

kubectl create \

-f jenkins-unprotected.yml \

--record --save-config

kubectl rollout status deploy Jenkins

We created an Ingress, a Deployment, and a Service object. We also executed the kubectl rollout status command that will tell us when the deployment is finished and this may take some time.

The jenkins-unprotected.yml definition does not use any new feature so we won’t waste time going through the YAML file. Instead, we’ll open Jenkins UI in a browser.

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#Open the link beside the run button in the playground below

Upon closer inspection, you’ll notice that there is no login button. Jenkins is currently unprotected. The image does allow the option to define an initial administrative username and password.

If the files /etc/secrets/jenkins-user and /etc/secrets/jenkins-pass are present, the init script will read them, and use the content inside those files to define the username and the password.

Since we’re already familiar with ConfigMaps, we could use them to generate those files. However, since the user and the password should be better protected than other configuration entries, we’ll switch to Secrets.

If you’re interested in details, please explore the [jenkins/Dockerfile](https://github.com/vfarcic/docker-flow-stacks/blob/master/jenkins/Dockerfile" \t "_blank) from the [vfarcic/docker-flow-stack](https://github.com/vfarcic/docker-flow-stacks" \t "_blank) repository. The important part is that it expects /etc/secrets/jenkins-user and /etc/secrets/jenkins-pass files. If we can provide them, in a relatively secure manner, our Jenkins will be (more) secured by default.

## The Built-In Secrets

We’ll start by checking whether we already have some Secrets in the cluster.

kubectl get secrets

The **output** is as follows.

NAME TYPE DATA AGE

default-token-l9fhk kubernetes.io/service-account-token 3 32m

We did not create any Secret, and yet one is available in the system.

The “default-token-l9fhk” Secret was created automatically by Kubernetes. It contains credentials that can be used to access the API.

Moreover, Kubernetes automatically modifies the Pods to use this Secret. Unless we tweak Service Accounts, every Pod we create will have this Secret. Let’s confirm that is indeed true.

kubectl describe pods

The **output**, limited to the relevant sections, is as follows.

...

Mounts:

/var/jenkins\_home from jenkins-home (rw)

/var/run/secrets/kubernetes.io/serviceaccount from default-token-l9fhk (ro)

...

Volumes:

jenkins-home:

Type: EmptyDir (a temporary directory that shares a pods lifetime)

Medium:

default-token-l9fhk:

Type: Secret (a volume populated by a Secret)

SecretName: default-token-l9fhk

Optional: false

...

#my output

Volumes:

jenkins-home:

Type: EmptyDir (a temporary directory that shares a pod's lifetime)

Medium:

SizeLimit: <unset>

We can see that two volumes are mounted. The first one ("/var/jenkins\_home") was defined by us. It’s the same mount volume we used in the previous chapter, and it is meant to preserve Jenkins’ state by mounting its home directory.

The second mount is the more interesting one. We can see that it references the auto-generated Secret “default-token-l9fhk” and that it mounts it as “/var/run/secrets/kubernetes.io/serviceaccount”. Let’s take a look at that directory.

POD\_NAME=$(kubectl get pods \

-l service=jenkins,type=master \

-o jsonpath="{.items[\*].metadata.name}")

kubectl exec -it $POD\_NAME -- ls \

/var/run/secrets/kubernetes.io/serviceaccount

The **output** is as follows.

ca.crt namespace token

By auto-mounting that Secret, we got three files. They are required if we’d like to access the API server from within the containers. “ca.crt” is the certificate, the namespace contains the namespace the Pod is running in, and the last one is the token we’d need to establish communication with the API.

We won’t go into examples that prove those files can be used to access the API server securely. Just remember that if you ever need to do that, Kubernetes has you covered through that auto-generated Secret.

Let’s get back to the task at hand. We want to make Jenkins more secure by providing it with an initial username and password.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create \

-f jenkins-unprotected.yml \

--record --save-config

kubectl rollout status deploy jenkins

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#Open the link beside the run button in the playground below

kubectl get secrets

kubectl describe pods

POD\_NAME=$(kubectl get pods \

-l service=jenkins,type=master \

-o jsonpath="{.items[\*].metadata.name}")

kubectl exec -it $POD\_NAME -- ls \

/var/run/secrets/kubernetes.io/serviceaccount

**Creating Generic Secrets**

Learn to create and explore generic Secrets.

**We'll cover the following**

* [Creating secrets](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVKZZkn3noy#Creating-secrets)
* [Working with secrets](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVKZZkn3noy#Working-with-secrets)
  + [JSON Representation](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVKZZkn3noy#JSON-Representation)
  + [Decoding the values](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVKZZkn3noy#Decoding-the-values)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVKZZkn3noy#Try-it-yourself)

**Creating secrets**

The commands to create Secrets are almost the same as those we used to create ConfigMaps. We can, for example, generate Secrets based on literal values.

kubectl create secret \

generic my-creds \

--from-literal=username=jdoe \

--from-literal=password=incognito

The major difference is that we specified the type of the Secret as generic.

It could also be docker-registry or tls. We won’t explore those two, but only say that the former can be used to provide kubelet with credentials it needs to pull images from private registries. The latter is used for storing certificates.

In this chapter, we’ll focus on the generic type of secrets which happen to use the same syntax as ConfigMaps.

Just as with ConfigMaps, generic Secrets can use --from-env-file, --from-file, and --from-literal as sources. They can be mounted as files, or transformed into environment variables. Since creating Secrets is so similar to creating ConfigMaps, we won’t go into all the permutations we can do.

For now, we created a Secret called my-creds which holds two literal values.

## Working with secrets

Let’s take a look at the Secrets we now have in the cluster.

kubectl get secrets

The **output** is as follows.

NAME TYPE DATA AGE

default-token-n6fs4 kubernetes.io/service-account-token 3 33m

my-creds Opaque 2 6s

We can see that the newly created Secret is available and that it has two pieces of data.

### JSON Representation

Let’s see the json representation of the Secret and try to find out how to retrieve it.

kubectl get secret my-creds -o json

The **output** is as follows (metadata is removed for brevity).

{

"apiVersion": "v1",

"data": {

"password": "aW5jb2duaXRv",

"username": "amRvZQ=="

},

"kind": "Secret",

"metadata": {

...

},

"type": "Opaque"

}

We can see that the “data” field contains the “password” and the “username”. They coincide with the literal values we specified in the command that created the Secret.

### Decoding the values

You’ll notice that the values are “strange”. They are encoded. If we’d like to see the original values we stored as secrets, we’ll need to decode them.

kubectl get secret my-creds \

-o jsonpath="{.data.username}" \

| base64 –decode

We used jsonpath to filter the output so that only the username data is retrieved. Since the value is encoded, we piped the output to base64 command that decoded it for us. The result is jdoe.

Similarly, the command that will retrieve and decode the second Secret data is as follows.

kubectl get secret my-creds \

-o jsonpath="{.data.password}" \

| base64 –decode

The **output** is “incognito”.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create secret \

generic my-creds \

--from-literal=username=jdoe \

--from-literal=password=incognito

kubectl get secrets

kubectl get secret my-creds -o json

kubectl get secret my-creds \

-o jsonpath="{.data.username}" \

| base64 --decode

kubectl get secret my-creds \

-o jsonpath="{.data.password}" \

| base64 –decode

**Mounting Generic Secrets**

Learn to mount the generic Secret to secure the deployed Jenkins.

**We'll cover the following**

* [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/x1wXXQnDLlq#Looking-into-the-definition)
* [Applying the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/x1wXXQnDLlq#Applying-the-definition)
* [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/x1wXXQnDLlq#Verification)
* [Destroying everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/x1wXXQnDLlq#Destroying-everything)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/x1wXXQnDLlq#Try-it-yourself)

**Looking into the definition**

Let’s see how we could mount the Secret we created. For this let us see an updated definition of jenkins.yml. The **Definition**, limited to the relevant parts, is as follows.

apiVersion: apps/v1

kind: Deployment

metadata:

name: jenkins

spec:

...

template:

...

spec:

containers:

- name: jenkins

image: vfarcic/jenkins

env:

- name: JENKINS\_OPTS

value: --prefix=/jenkins

volumeMounts:

- name: jenkins-home

mountPath: /var/jenkins\_home

- name: jenkins-creds

mountPath: /etc/secrets

volumes:

- name: jenkins-home

emptyDir: {}

- name: jenkins-creds

secret:

secretName: my-creds

defaultMode: 0444

items:

- key: username

path: jenkins-user

- key: password

path: jenkins-pass

...

**Line 19-20:** We added jenkins-creds that mounts the /etc/secrets directory.

**Line 24-26:** The jenkins-creds Volume references the Secret named my-creds.

**Line 27:** Since we want the process inside the container to only read the Secret, we set the defaultMode to 0444. That will give read permissions to everyone. Typically, we’d set it to 0400, thus giving the read permissions only to the root user. However, since the Jenkins image uses the jenkins user, we gave read permissions to everyone instead of only to the root user.

**Line 29-32:** Finally, since the image expected files named jenkins-user and jenkins-pass, we made explicit paths. Otherwise, Kubernetes would create files username and password.

## Applying the definition

Let’s apply the new definition.

kubectl apply -f jenkins.yml

kubectl rollout status deploy Jenkins

We applied the definition and waited until the new objects were rolled out.

## Verification

Now we can check whether the correct files are indeed stored in the /etc/secrets directory.

POD\_NAME=$(kubectl get pods \

-l service=jenkins,type=master \

-o jsonpath="{.items[\*].metadata.name}")

kubectl exec -it $POD\_NAME \

-- ls /etc/secrets

The **output** of the latter command is as follows.

jenkins-pass jenkins-user

The files we need are indeed injected. To be on the safe side, we’ll also check the content of one of them.

kubectl exec -it $POD\_NAME \

-- cat /etc/secrets/jenkins-user

The **output** is “jdoe”, the username of our newly deployed Jenkins.

Finally, let’s confirm that the application is indeed secured.

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

#Click the link beside the run button

You’ll see that, this time, the link to create new jobs is gone.

Please use jdoe and incognito if you’d like to login to your newly deployed and (more) secured Jenkins.

## Destroying everything

For now, we’ll destroy the cluster we used in this chapter.

k3d cluster delete mycluster –all

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl apply -f jenkins.yml

kubectl rollout status deploy jenkins

POD\_NAME=$(kubectl get pods \

-l service=jenkins,type=master \

-o jsonpath="{.items[\*].metadata.name}")

kubectl exec -it $POD\_NAME \

-- ls /etc/secrets

kubectl exec -it $POD\_NAME \

-- cat /etc/secrets/jenkins-user

kubectl port-forward service/jenkins 3000:8080 --address 0.0.0.0

k3d cluster delete mycluster –all

**Secrets Compared to ConfigMaps**

Compare Kubernets Secrets and ConfigMaps.

**We'll cover the following**

* [The Similarities](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JQRG0BON2Ql#The-Similarities)
* [The Differences](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JQRG0BON2Ql#The-Differences)
* [Conclusion](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JQRG0BON2Ql#Conclusion)

**The Similarities**

So far, Kubernetes Secrets do not seem to differ from ConfigMaps. From a **functional perspective**, they are, indeed, the same.

* Both allow us to inject some content. Both can use files, literal values, and files with environment variables as data sources.
* Both can output data into containers as files or as environment variables.
* Even the syntax for using Secrets is almost the same as the one used for ConfigMaps.

**The Differences**

The only significant difference between ConfigMaps and Secrets is that the latter creates files in a *tmpfs* (temporary file storage).

Secrets are constructed as in-memory files, thus leaving no trace on the host’s files system. That, in itself, is not enough to call Secrets secure, but it is a step towards it. We’d need to combine them with *Authorization Policies* to make the passwords, keys, tokens, and other never-to-be-seen-by-publicly types of data secure. Even then, we might want to turn our attention towards third-party Secret managers like [HashiCorp Vault](https://www.vaultproject.io/" \t "_blank).

**Conclusion**

Secrets are almost the same as ConfigMaps. The main difference is that the secret files are created in *tmpfs*. Kubernetes secrets do not make your system secure. They are only a step towards such a system.

**Not so Secretive Secrets**

Explore the insecurities associated with using Secrets and discuss the actions we can perform to secure the environment.

**We'll cover the following**

* [The Insecurities](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N78NBlw2Qqz#The-Insecurities)
* [How to Secure?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N78NBlw2Qqz#How-to-Secure?)

**The Insecurities**

Almost everything Kubernetes needs is stored in [etcd](https://github.com/coreos/etcd" \t "_blank). That includes Secrets. The problem is that they are stored as plain text. Anyone with access to etcd has access to Kubernetes Secrets. We can limit the access to etcd, but that’s not the end of our troubles.

*etcd* stores data to disk as plain text. Restricting the access to etcd still leaves the Secrets vulnerable to who has access to the file system. That, in a way, diminishes the advantage of storing Secrets in containers in tmpfs. There’s not much benefit of having them in tmpfs used by containers, if those same Secrets are stored on disk by etcd.

Even after securing the access to etcd and making sure that unauthorized users do not have access to the file system partition used by etcd, we are still at risk. When multiple replicas of etcd are running, data is synchronized between them. By default, etcd communication between replicas is not secured. Anyone sniffing that communication could get a hold of our secrets.

Kubernetes Secrets are a step in the right direction. It is, without a doubt, better to use Secrets than to expose confidential information as environment variables or other less secure methods. Still, Secrets can give us a false sense of security.

**How to Secure?**

We need to take additional precautions to protect ourselves. That might include, but is not limited to, the following actions:

* Secure the communication between etcd instances with SSL/TLS.
* Limit the access to etcd and wipe the disk or partitions that were used by it.
* Do not define Secrets in YAML files stored in a repository. Create Secrets through ad-hoc kubectl create secret commands. If possible, delete commands history afterward.
* Make sure that the applications using Secrets do not accidentally output them to logs or transmit them to other applications.
* Create policies that allow only trusted users to retrieve secrets. However, you should be aware that even with proper policies in place, any user with permissions to run a Pod could mount a Secret and read it.

We did not yet explore etcd configuration, nor did we learn how to set up authorization policies. For now, just remember that Secrets are not as secured as one might think. At least, not those provided by Kubernetes community. We do encourage you to use them, as long as you’re aware of their shortcomings.

**Comparison with Docker Swarm**

Comparison between Kubernetes Secrets and the Docker Swarm equivalent.

**We'll cover the following**

* [The Similarities](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/m2AKZvM2nA3#The-Similarities)
* [The Differences](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/m2AKZvM2nA3#The-Differences)
* [Conclusion](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/m2AKZvM2nA3#Conclusion)

**The Similarities**

Secrets are very similar to Kubernetes ConfigMaps and Docker Swarm configs. Everything we said for configurations applies to Secrets, with a few additional features.

Both Kubernetes and Docker Swarm stores Secrets in *tmpfs* inside containers. From that aspect, they are equally secure.

Docker swarm vs Kubernetes

**The Differences**

The significant difference is in the way Secrets are stored internally.

Kubernetes stores Secrets in etcd. By default, they are exposed, and we need to take extra precautions to protect them.

Docker Swarm secrets are, on the other hand, more secure by default. They are synchronized between managers using SSL/TLS, and they are encrypted at rest. We prefer “secured by default” approach behind Docker Swarm secrets.

In Kubernetes, we need to take extra steps to reach a similar level of security as with Docker Swarm.

On the other hand, Kubernetes integration with third-party solutions for secrets is much better. For example, plugging in [HashiCorp Vault](https://www.vaultproject.io/" \t "_blank) into a Kubernetes workflow is much smoother than if we’d try to integrate it with Docker Swarm. Using Vault is a better solution than what Kubernetes and Swarm offer.

**Conclusion**

Even though Kubernetes can be made more secure with Vault and similar products, for now, we are evaluating secrets management that comes with Kubernetes and Docker Swarm.

If we exclude third-party solutions, Docker Swarm has a clear advantage over Kubernetes. Its secrets are more secure by default. Even after tweaking Kubernetes (especially etcd), Docker Swarm is still more secure.

That does not mean that secrets management with both products does not have a lot to be desired. Both have their shortcomings.

We must proclaim **Docker Swarm as a winner** in this round. Its secrets are more secretive.

**What's Next?**

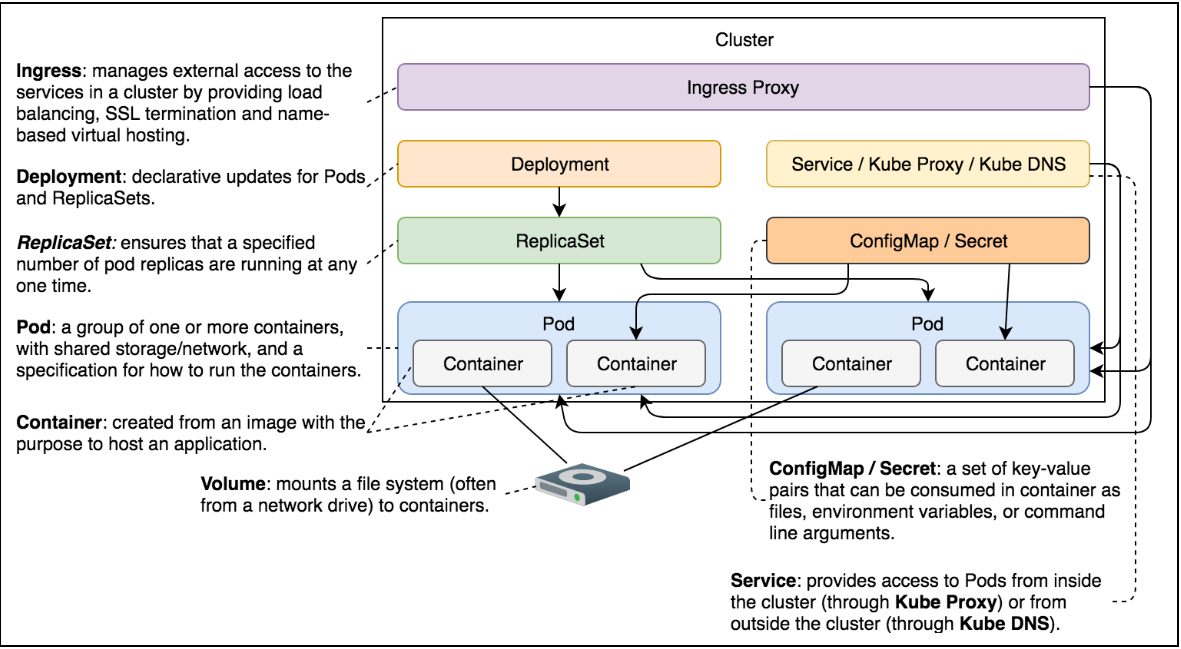
Recap what we have learned so far and what we are going to explore next.

**We'll cover the following**

* [Summary](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qA6Q0YAoZW7#Summary)

**Summary**

In this chapter, we started off with understanding secrets, explored the inbuilt Kubernetes Secrets, created and mounted generic Secrets and compared Secrets with ConfigMaps. Finally, we went through the shortcomings associated with using Secrets and proposed the solution to overcome these.



ℹ️ If you’d like to know more about Secrets, please explore [Secret v1 core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.10/#secret-v1-core) API documentation.