Kubernetes: Certificates, Tokens, Authentication and Service Accounts

 Daniel Albuschat May 19 '19 · 9 min read

#kubernetes

Mostly for personal/learning experiences, I have created quite a few Kubernetes clusters, such as the one on my Raspberry Pi rack. I also created two clusters for a production and a staging environment on ultra-cheap cloud servers from Hetzner Cloud. Luckily, none of those environments where serious business.

Disclaimer: I'm not a Kubernetes expert, nor am I a security expert, so make sure that you second-source the information you find on this post before you rely on them. I just wanted to publish the experience and insights that I made during this trip - thanks!

Why was that lucky?

Because **I accidentally leaked the certificates** for my admin access to the staging cluster. I was trying to set up a CI/CD pipeline for an open source project using CircleCI. While I was testing out the steps one by one, I dumped the content of `${HOME}/.kube/config` that has been created from a BASE64-encoded environment variable, like described on this blog post. That was fatal, though, since a) the job logs of open source projects are **publicly visible** and b) jobs and their logs **can not**

be deleted manually, I had to reach out to the support for this. *Ouch!*

So let's dig into what happened here.

Creating a cluster

First of all, I created the cluster manually using `kubeadm`, following the official docs. Doing so, I created a cluster with RBAC enabled and a `kube-config` has been created for me that includes a user that is identified by a certificate.

Accessing the cluster

After `kubeadm` created the cluster successfully, it instructs you what to do to access your cluster:

```
"Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:"

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

The `admin.conf` that kubeadm creates includes a user identified by a certificate:

```
- name: kubernetes-admin
  user:
    client-certificate-data: <BASE64 ENCODED X509 CERTIFICATE>
    client-key-data:  <BASE64 ENCODED PRIVATE KEY FOR THE CERTIFICATE>
```

Following these instructions is your only way to access this cluster using `kubectl` as of now, so you should go ahead and do this now. After you copied the `admin.conf`, you have `cluster-admin` access. You are `root`, so to say.

How is that?

What `kubeadm` did is that it created a new CA (Certificate Authority) root certificate that is the master certificate for your cluster. It looks something like this:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0 (0x0)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = kubernetes
        Validity
            Not Before: May 19 11:11:04 2019 GMT
            Not After : May 16 11:11:04 2029 GMT
        Subject: CN = kubernetes
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    <REDACTED>
                Exponent: 65537 (0x10001)
        X509v3 extensions:
```

```
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment, Certificate Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
         <REDACTED>
```

So... it doesn't really contain much besides the info that it is a CA and it's CN (= Common Name) is Kubernetes. That's because this cert only acts as a root for other certs that are used for different purposes on the cluster. You can have a look at `/etc/kubernetes/pki` to take a peek at some of the certs that are used in your cluster and have been signed by the CA:

```
daniel@kube-box:~# ls /etc/kubernetes/pki/ -1
apiserver.crt
apiserver-etcd-client.crt
apiserver-etcd-client.key
apiserver.key
apiserver-kubelet-client.crt
apiserver-kubelet-client.key
ca.crt
ca.key
etcd
front-proxy-ca.crt
front-proxy-ca.key
front-proxy-client.crt
front-proxy-client.key
sa.key
sa.pub
```

It is possible to allow access to clients that authenticate themselves using certificates that are trusted by the CA. This is

enabled by passing this `ca.crt` to `kube-controller-manager` in the `--client-ca-file` parameter. This is what the docs have to say about it:

```
--client-ca-file string
If set, any request presenting a client certificate signed by one of the
authorities in the client-ca-file is authenticated with an identity
corresponding to the CommonName of the client certificate.
```

Back to your `kube-config` : The certificate that is included in BASE64 in your `admin.conf` is signed by that exact CA. This is why it is trusted by the cluster. Let's have a look at the certificate:

```
grep 'client-certificate-data: ' ${HOME}/.kube/config | \
   sed 's/.*client-certificate-data: //' | \
   base64 -d | \
   openssl x509 --in - --text

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 3459994011761527671 (0x30045e38cc064b77)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = kubernetes
        Validity
            Not Before: May 12 10:54:39 2019 GMT
            Not After : May 11 10:54:42 2020 GMT
        Subject: O = system:masters, CN = kubernetes-admin
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    <REDACTED>
```

```
            Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Client Authentication
    Signature Algorithm: sha256WithRSAEncryption
         <REDACTED>
```

What does this cert tell us (and the cluster)?

a) It is issued and trusted by our `kubernetes` cluster
b) It identifies the Organisation (`O`) `system:masters` , which is interpreted as a group by kubernetes
c) It identifies the Common Name (`CN`) `kubernetes-admin` , which is interpreted as a user by kubernetes

In other words: This certificate logs in as the user `kubernetes-admin` with the group `system:masters` . This is the reason why you don't need to provide the group name in the `kube-config` , and why you can change the user's name at will in the `kube-config` , without this changing the actual user that is being logged in.

Where are the permissions defined?

In RBAC-enabled clusters, permissions are defined in `Roles` (per namespace) or `ClusterRoles` (for all namespaces). These permissions are then granted to objects using `RoleBindings` and `ClusterRoleBindings` . So what you have to look for are `RoleBindings` and `ClusterRoleBindings` that grant permissions

to the group `system:masters` or the user `kubernetes-admin`. You can do this by having a look at the output of

```
kubectl -A=true get rolebindings && kubectl -A=true get clusterrolebindings
```

The default setup that `kubeadm` created for me yielded one hit for that search, the `ClusterRoleBinding` named `cluster-admin`, which grants permissions to a `ClusterRole` with the same name. Here's the definition:

```
kind: ClusterRole
metadata:
  name: cluster-admin
rules:
- apiGroups:
  - '*'
  resources:
  - '*'
  verbs:
  - '*'
- nonResourceURLs:
  - '*'
  verbs:
  - '*'
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
```

```
kind: Group
name: system:masters
```

So there we have it! The group `system:masters`, which the certificate authorizes as, grants '*' permissions to all resources using all verbs, hence full access.

Unvalidated assumption: I think that users and groups in this case are only defined in the certificates, and new users can be `created` by issuing a new certificate with the CommonName set to the desired username and the Organisation set to the desired group. This username and group can then, without further ado, be used in `ClusterRoleBindings` and `RoleBindings`. I did not take the time to validate this, though, which would be possible by issuing a new certificate using `openssl`, signed with the cluster's CA. It'd be great if someone could confirm or debunk this assumption in a comment!

The mystery

So I got this far and found out how the user and group are identified and how permissions are granted to this user and group. My guess then was that when I delete the `ClusterRoleBinding`, or rather remove the group `system:masters` from it, that the certificate should not have access to the cluster anymore. If I did that, and it had the expected result, I would lose all access to the cluster and would have successfully logged me out for good. So I first added a `serviceaccount` and created a `kube-config` that logged in using a token for that `serviceaccount` and verified that the access

worked. We will see later how to do this. Then, after setting the safety net in place, I removed the `system:masters` subject from the `ClusterRoleBinding`. To my surprise, this did not lock the user out. I could still fully access the cluster using the old `kube-config` … maybe someone can explain this behaviour in a comment?

Alternative 1: Replace the CA

One sure-as-hell way to make the leaked certificate useless is to replace the CA in the cluster. This would require a restart of the cluster, though. And it would require to re-issue all the certificates that we have seen above, and maybe some more. I rated the possibility to totally fuck everything up and waste multiple hours on the trip at about 99%, so I abandoned the plan. :-)

Alternative 2: Rebuild the whole cluster

Luckily, it was a staging cluster, so I had plenty of freedom. Before starting my investigations, I powered off all nodes. Then, after not finding a proper solution to only make the leaked certificate useless, I killed the whole cluster using

```
kubeadm reset
rm -rf /etc/kubernetes
rm -rf /var/lib/kubelet
```

And recreated from scratch with kubeadm. (Which is so great, by the way!!)

Then I went ahead and made a few dozen more commits reading 'NOT printing the content of the kube-config anymore', 'Getting CI/CD to work', 'Maybe now it works', 'Uhm what?', 'That gotta work', 'fuck CI/CD', … :-)

Lessons learned: Use service-accounts with tokens

(Or other authentication methods like OpenID, as recommended in this awesome post.)

So my lesson learned is to do what I've seen at the big managed kubernetes providers: Use a service-account and it's access token for authorization. Here I'll show how to set up a super-user that uses a token instead of a cert:

```
kubectl -n kube-system create serviceaccount admin
```

To grant super-user permissions, the easiest way is to create a new `ClusterRoleBinding` to bind this service-account to the `cluster-admin` `ClusterRole`:

```
kubectl create clusterrolebinding add-on-cluster-admin \
  --clusterrole=cluster-admin \
  --serviceaccount=kube-system:admin
```

Use your new service-account

Your admin user is now ready and armed. Now we need to log in with this user. I assume that you have the `admin.conf` in `${HOME}/.kube/conf`. We now want to add the new user, identified by it's token, and add a new context that uses this user:

```
TOKENNAME=`kubectl -n kube-system get serviceaccount/admin -o jsonpath='{.secr
TOKEN=`kubectl -n kube-system get secret $TOKENNAME -o jsonpath='{.data.token}
kubectl config set-credentials admin --token=$TOKEN
kubectl config set-context admin@kubernetes --cluster kubernetes --user admin
```

Now go ahead and try your new, shiny service-account:

```
kubectl config use-context admin@kubernetes
kubectl -n kube-system get all
```

If this went well, you should go ahead and delete the certificate-based user and the corresponding context:

```
kubectl config unset users.kubernetes-admin
kubectl config delete-context kubernetes-admin@kubernetes
```

Yay! Now we have a `kube-conf` that only includes token-based access. This is great, because it is very easy to revoke that token if this config might be leaked or published.

How to invalidate a leaked token

This is easy! Just delete the secret that corresponds to the user's token. We already saw how to find out which is the correct secret:

```
kubectl -n kube-system get serviceaccount/admin -o yaml
```

You will see a field "name" in the "secrets" array. This is a name of a secret that holds this service-account's token. Now go ahead and simply delete it:

```
kubectl -n kube-system delete secrets/token-admin-xyz123
```

Then wait a few seconds, and try to access your cluster:

```
dainel@kube-box:~# kubectl -n kube-system get all
error: You must be logged in to the server (Unauthorized)
```

Wohoo!

But how do you regain access? Well, if you're on your master node, simply copy the `admin.conf` back to your `${HOME}/.kube/conf` and repeat the steps from "Use your new service-account". Kubernetes will have created and assigned a new token by now.

I hope that this helped, and I'd love to hear feedback, errata, etc. in the comments!

Also make sure to read the VERY comprehensible and awesome post "Kubernetes Security Best-Practices" by Peter Benjamin.

And a big thank you to Andreas Antonsson, vaizki and Alan J Castonguay, who have helped me on the official Kubernetes Slack channel to get a better understanding of what is going on.

Daniel Albuschat + FOLLOW

Have had many hats on in my life: Developer, Team Lead, Scrum Master, Architect and most recently Product Owner. Interested in product discovery, quality assurance and language design.

@danielkun dalbuschat daniel-kun

You're Unvalidated Assumption is correct. Kubernetes does not have a database to store usernames, so you can refer to any arbitrary username you want in the Subject of your certificate, k8s will make authorization decisions based on role/bindings given that username.

1 REPLY

jialin

Oct 16 '19

'-A' unknown flag

1 REPLY

Daniel Albuschat

Oct 19 '19

Hey jialin,

the -A flag was introduced in a recent kubectl version. I guess you are using a previous version, so you'd need to update to use kubectl -A.

Greetings!

1 REPLY

qbaze

Sep 9 '19

Very useful! Thanks.

1 REPLY

Amruta Ranade

May 30 '19

Oh this is so helpful! I am experimenting with Kubernetes - trying out different auth/custom CA cert scenarios. Thanks for sharing your experience :)

1 REPLY

Daniel Albuschat

Jun 1 '19

Thanks!
I have been told by multiple sources, however, that using Service Account tokens isn't a silver bullet and not recommended, either O_o

The reason is that the tokens are "ephemeral", whatever that means. I have yet to find out when/why they will be recreated. I personally don't see the disadvantage to certs, though, since you should totally periodically roll your credentials anyways, so I'd suggest to do this with certs, too. But it turns out, as described in the article, that rolling (and therefore invalidating the old) certs is a huge PITA.

It's all still a mystery to me.

2 REPLY

Closing the Kubernetes Skills Gap with Developer-First Learning


katjuell

How do you begin learning Kubernetes? This is an important question for any organization building, hosting, or managing applications today.

142 2

TIL: The path to CKA Week 5.

Jace - Jan 5

10 Best Kubernetes Courses Online

Digital Defynd - Jan 7

Apache Camel K on AKS quick setup

Haris Secic - Jan 5

Do 0 ao CKA pt 1/??

Marcelo Freire - Jan 2