Jonathan

Follow 29 Followers About

Certified Kubernetes Security Specialist (CKS) Preparation Part 3— Cluster Setup

Jonathan Feb 17 · 9 min read

If you have not yet checked the previous parts of this series, please go ahead and check Part1 and Part2.

In this article, I would focus on the preparation around **cluster setup** in CKS certification exam. Some focus would be on CIS Kubernetes Benchmark, kube-bench and verify platform binaries as these 3 are new to me.

Network Policy

Let's first look at Kubernetes network policy. The concept is pretty similar to firewall, just that it could manage pod-to-pod, pod-to-service and external-to-service communication. Administrators identify what resources could access and could be accessed by Pod labels, namespaces or IP addresses. Pretty straightforward. Take the default example from official documentation, it basically means

- Pods with label "role: db" could be accessed with TCP, and port 6379
- Pods and Services with label "role: frontend", in namespace "project: myproject" and reside in 172.17.0.0/16 would be able to access Pods with label "role: db".
- Resources reside in 172.17.1.0/24 would not be able to access Pods with label "role: db".
- The same concept applies to egress rules.

```yaml
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

Reference: Network Policies | Kubernetes

With proper network security rules, administrators could ensure only the allowed resources are accessing the resources that made accessible with the accessible protocols and ports. Pay extra attention on default deny all and allow all rules.

Deny All Ingress

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```

```
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Allow All Ingress

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
  - {}
  policyTypes:
  - Ingress
```

Let's try to see this in practice. We do

- Allow ingress from Pod in namespace label "cks: allow" to Pod with label "cks: denymost"
- Allow ingress from Pod label "cks: allow" to Pod with label "cks: denymost"

Network Policy

```
jonw@CKS-Master:~$ cat networkpolicy-allowaccessors.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-accessors
  namespace: default
spec:
  podSelector:
    matchLabels:
      cks: denymost
```

```
  - from:
    - namespaceSelector:
        matchLabels:
          cks: allow
    - podSelector:
        matchLabels:
          cks: allow
```

Test accessing to Pod with label “cks: denymost” from Pod with label “cks: denymost”

```
jonw@CKS-Master:~$ kubectl get pods denymost -o wide --show-labels
NAME       READY   STATUS    RESTARTS   AGE   IP          NODE         NOMINATED NODE   READINESS GATES   LABELS
denymost   1/1     Running   0          19m   10.44.0.2   cks-worker   <none>           <none>            cks=denymost
jonw@CKS-Master:~$ kubectl get pods -A -o wide --show-labels | grep allow
allowingress   accessor2                       1/1     Running   0          25m   10.44.0.1   cks-worker   <none>           <none>            run=accessor2
default        accessor1                       1/1     Running   0          29m   10.44.0.4   cks-worker   <none>           <none>            cks=allow
jonw@CKS-Master:~$ kubectl get ns --show-labels
NAME              STATUS   AGE   LABELS
allowingress      Active   34m   cks=allow
default           Active   41h   <none>
kube-node-lease   Active   41h   <none>
kube-public       Active   41h   <none>
kube-system       Active   41h   <none>
```

```
jonw@CKS-Master:~$ kubectl exec accessor1 -- curl 10.44.0.2
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   612  100   612    0     0   597k      0 --:--:-- --:--:-- --:--:--  597k
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Pod denymost IP

```
jonw@CKS-Master:~$ kubectl exec accessor2 -n allowingress -- curl 10.44.0.2
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
```

```
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

What about trying to access Pod with label “cks: denymost” from elsewhere?Denied.

```
jonw@CKS-Master:~$ kubectl get pods accessor3 -o wide --show-labels
NAME        READY   STATUS    RESTARTS   AGE   IP           NODE         NOMINATED NODE   READINESS GATES   LABELS
accessor3   1/1     Running   0          17m   10.44.0.3    cks-worker   <none>           <none>            run=accessor3
jonw@CKS-Master:~$ kubectl exec accessor3 -- curl 10.44.0.2
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:--  0:00:20 --:--:--     0*C
```

Kubernetes Dashboard

Although most administrators would prefer to use Kubernetes CLI commands to administrate all resources within K8s clusters, some still prefer to have visual display. Kubernetes Dashboard is created for this reason.

By executing the command below (more details in this website), people easily setup the all Kubernetes Dashboard required resource under the namespace of “kubernetes-dashboard”.

- *kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.1.0/aio/deploy/recommended.yaml*

```
jonw@CKS-Master:~$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.1.0/aio/deploy/recomme
nded.yaml
```

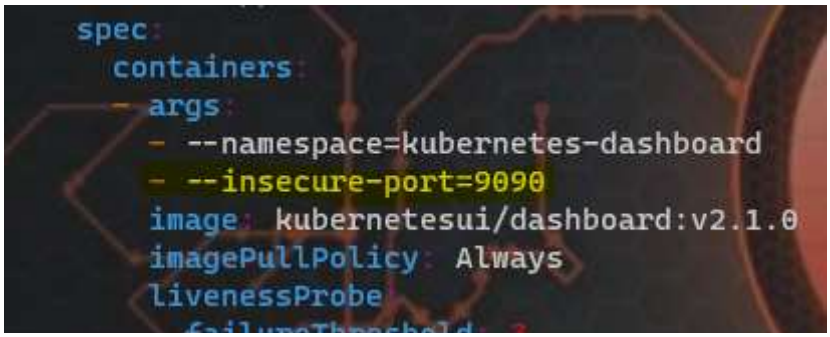
We could see all the resources being deployed in the one liner.

- *kubectl -n kubernetes-dashboard get pod,deploy,svc*

As most of you notice, default Kubernetes Dashboard service is exposed as Cluster IP and it would not be possible for administrators to access this IP address without getting inside a shell inside a Pod. For most cases, administrators use "*kubectl proxy*" to proxy an endpoint within the working machine to the actual Kubernetes Dashboard service. For more information on how to proxy the service, check here.

In some testing environments in less security concern, we could make Kubernetes Dashboard deployments and services to be exposed with Node Port, so administrators could use nodes' IP address, public or private, and assigned port to access the service. We edit the actual running deployment YAML.

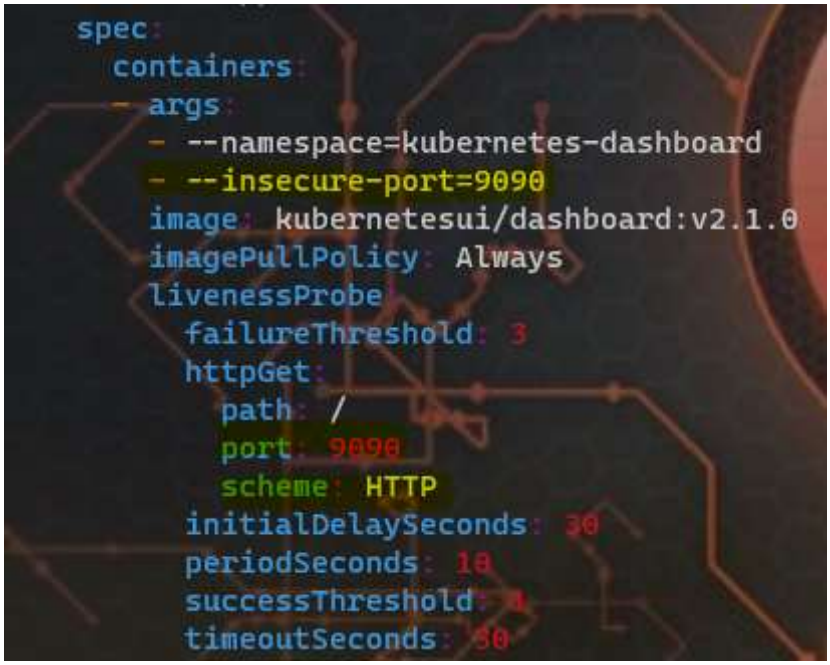
- *kubectl edit deployment kubernetes-dashboard -n kubernetes-dashboard*



- Remove the argument of auto generate certificates
- Add the argument of insecure port 9090
- Change livenessProbe to listen on port 9090
- Change scheme to listen on HTTP

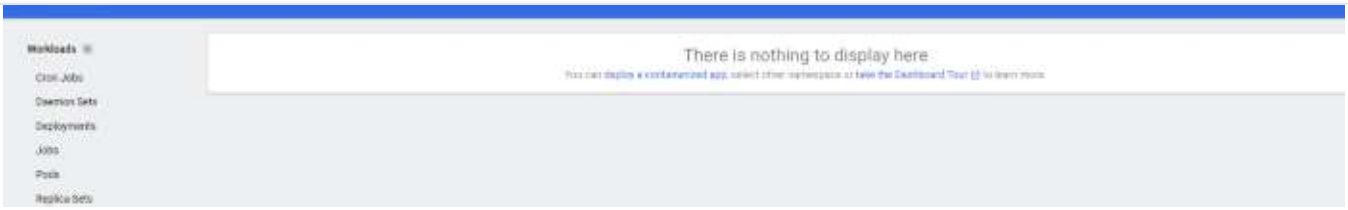
After that, we make changes on Kubernetes Dashboard services.

- *kubectl edit service kubernetes-dashboard -n kubernetes-dashboard*



- Change port to 9090
- Change targetPort to 9090
- Change type to NodePort

Right now, try to access Kubernetes Dashboard with nodes' IP address with assigned port.



Since Kubernetes Dashboard is leveraging service account “default” in namespace “kubernetes-dashboard” for accessing each resource, binding the right permission to this service account would allow the dashboard to show more information in the corresponding namespaces.

```
jonw@CKS-Master:~$ kubectl get sa -n kubernetes-dashboard
NAME                   SECRETS   AGE
default                1         100m
kubernetes-dashboard   1         100m
```

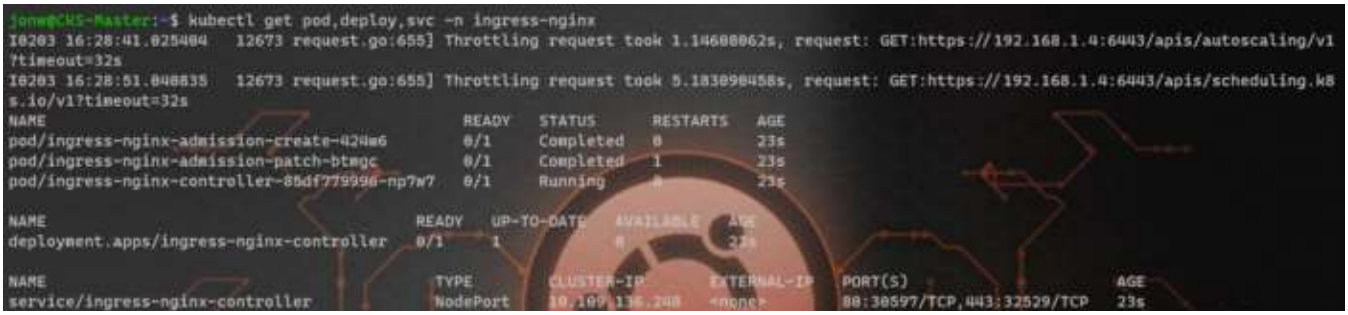
Secure Ingress

As most of the readers here already have some practical experience on K8s, one of the most popular ingress being used nowadays in K8s is NGINX Ingress, so we would use it as the example. First thing first, we could install NGINX Ingress with one liner.

- *kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.43.0/deploy/static/provider/baremetal/deploy.yaml*

The detailed instruction is here. Let’s take a look what have been created. One thing to note down is that NGINX Ingress is now

- Being exposed as Node Port service
- Assigned **high port 30597 over HTTP**
- Assigned **high port 32529 over HTTPS**.



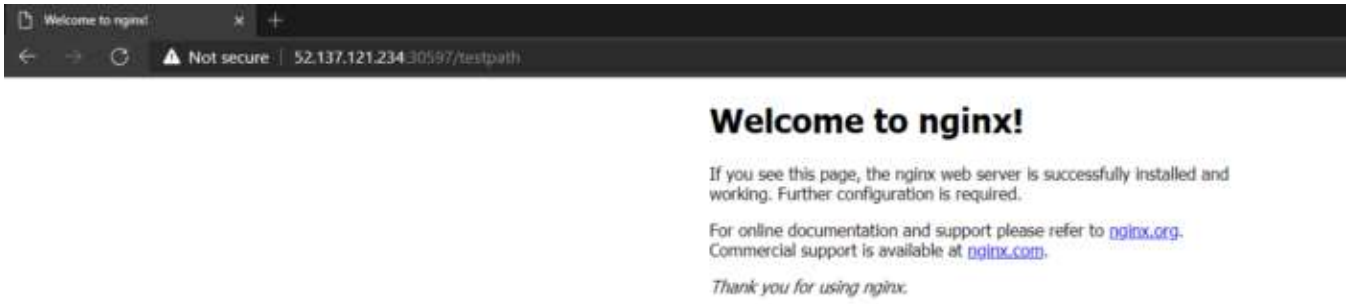
After installation, our goal would be to expose a couple of services to test ingress functionalities. As you see below, 2 Pods (service1, service2) has been created and both being exposed as ClusterIP service (service1, service2).

With that, we could head over to official documentation and check the section about how to set up Ingress rules. Basically, what it means in the template is that Ingress would be directing traffic to a service named “test” with the path of “http://<node IP>:<assigned node port>/testpath”. So, let’s give it a try.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

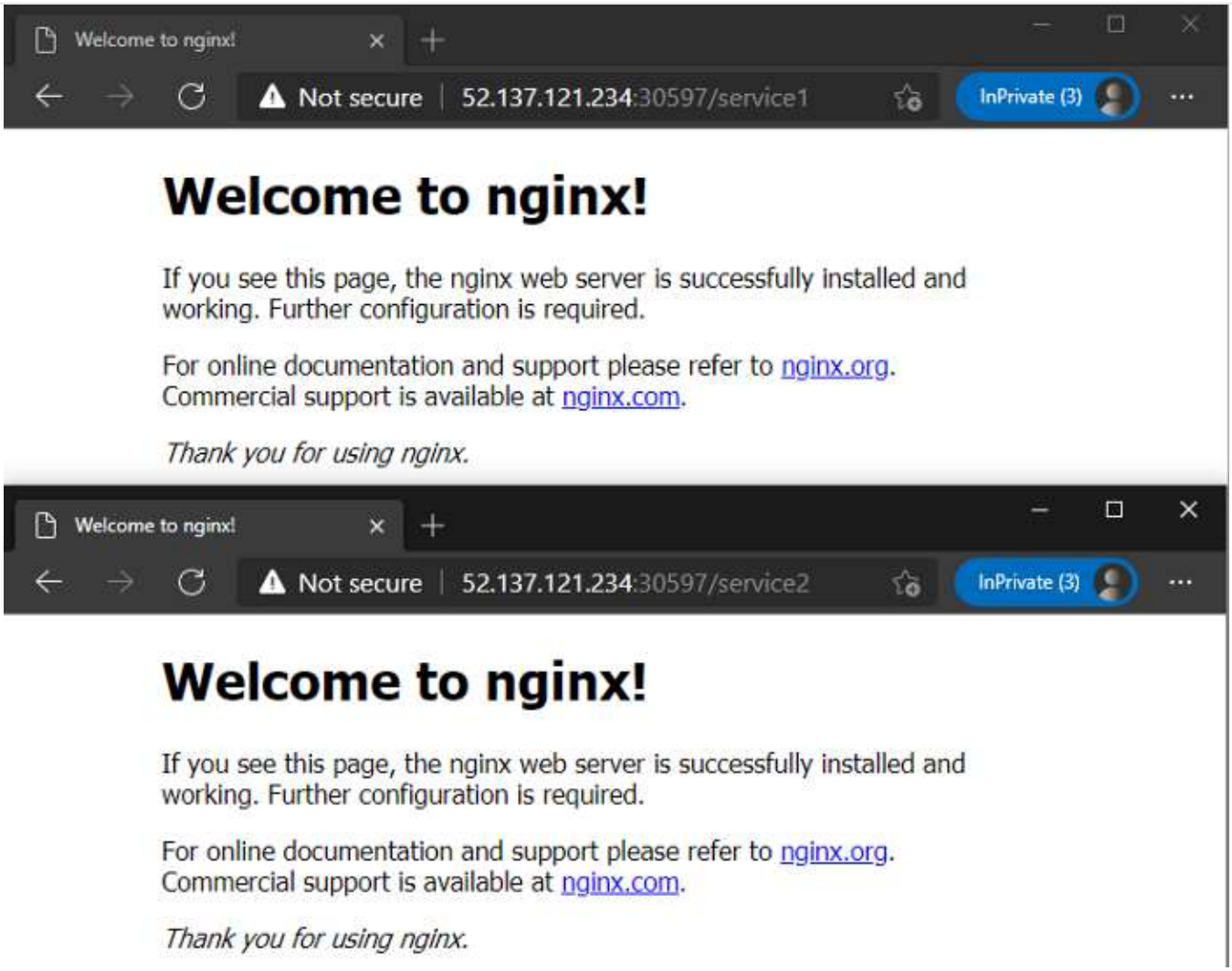
**Note: since we expose service with names as “service1” and “service2” earlier, we would need to make sure the Ingress YAML is actually pointing to service with those

We see we could access “http://<node IP>:<assigned node port>/testpath” and view the default NGINX content.



We could make Ingress to point to 2 different services with different paths. Since we are making Ingress to point to more than 1 service hosted inside K8s cluster, it makes sense to use path name to correctly show which service we are trying to reach.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /service1
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 80
      - path: /service2
        pathType: Prefix
        backend:
          service:
            name: service2
            port:
              number: 80
```



Now we know the service works with HTTP. We have reached the point of making the website to be only accessible over HTTPS, a secure way of accessing the content. Of course, we would first need a certificate to bind the website.

Create self-signed certificate and key with one liner.

- *openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes*

- *kubectl create secret tls secure-ingress — cert=cert.pem — key=key.pem*

Bind the TLS secret to the existing Ingress. For detailed template, please refer to this site.

From the screenshot above, we learn Ingress service is listening on 32529 for HTTPS . On Linux environment, use commands like below to visit websites using self-signed certificate

- *curl https://secure-ingress.com:32529 — resolve secure-ingress.com:32529:52.137.121.234 -k*

Resolved public IP is needed since the FQDN resolution would not work in this situation and “-k” is needed for insecure access as it is leveraging self-signed certificate.

Add "*-v*" at the end of the curl command to get verbose of the whole process of clients accessing HTTPS website and we would see the access is using self-signed certificate for certain.

- *curl https://secure-ingress.com:32529 — resolve secure-ingress.com:32529:52.137.121.234 -kv*

Node Metadata Protection

Every virtual machines (nodes) hosted in the cloud would need to access node metadata endpoint for various reasons. However, allowing the access to every resource is not ideal. To enhance the security of this access is to apply network policies, so only designated resources within K8s could have the ability of contacting node metadata endpoints. As this is more or less the same concept as section "**Network Policy**", we would just quick go to the next section.

CIS Benchmarks

target is to see whether administrators could make running K8s cluster to reach that goal. With that in mind, we could head to this site and download the CIS Benchmarks PDF for K8s.

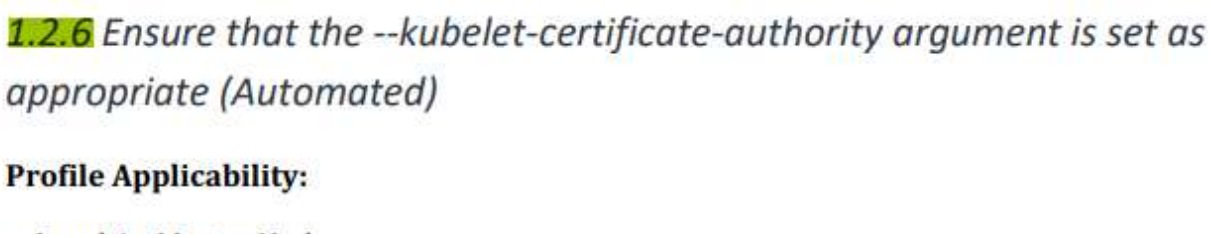
The PDF file is essentially a reference to let you know what needs to be modified within K8s clusters, but how would we know what needs to be changed? That is where kube-bench come into play. From this site, run

- *docker run — pid=host -v /etc:/etc:ro -v /var:/var:ro -t aquasec/kube-bench:latest [master|node] — version 1.13*

You might need to replace the content of "*master|node*" and "*running K8s version*".

Then, you would get a result similar to below.

We could ignore the passed ones and focus the ones with [FAIL]. Using [FAIL] 1.2.6 as an example, we first head to the CIS Benchmarks PDF for K8s and search for 1.2.6. Then, we see something like below.



Verify kubelet's certificate before establishing connection.

Rationale:

The connections from the apiserver to the kubelet are used for fetching logs for pods, attaching (through kubectl) to running pods, and using the kubelet's port-forwarding functionality. These connections terminate at the kubelet's HTTPS endpoint. By default, the apiserver does not verify the kubelet's serving certificate, which makes the connection subject to man-in-the-middle attacks, and unsafe to run over untrusted and/or public networks.

Audit:

Run the following command on the master node:

```
ps -ef | grep kube-apiserver
```

Verify that the `--kubelet-certificate-authority` argument exists and is set as appropriate.

Remediation:

Follow the Kubernetes documentation and setup the TLS connection between the apiserver and kubelets. Then, edit the API server pod specification file `/etc/kubernetes/manifests/kube-apiserver.yaml` on the master node and set the `--kubelet-certificate-authority` parameter to the path to the cert file for the certificate authority.

```
--kubelet-certificate-authority=<ca-string>
```

Impact:

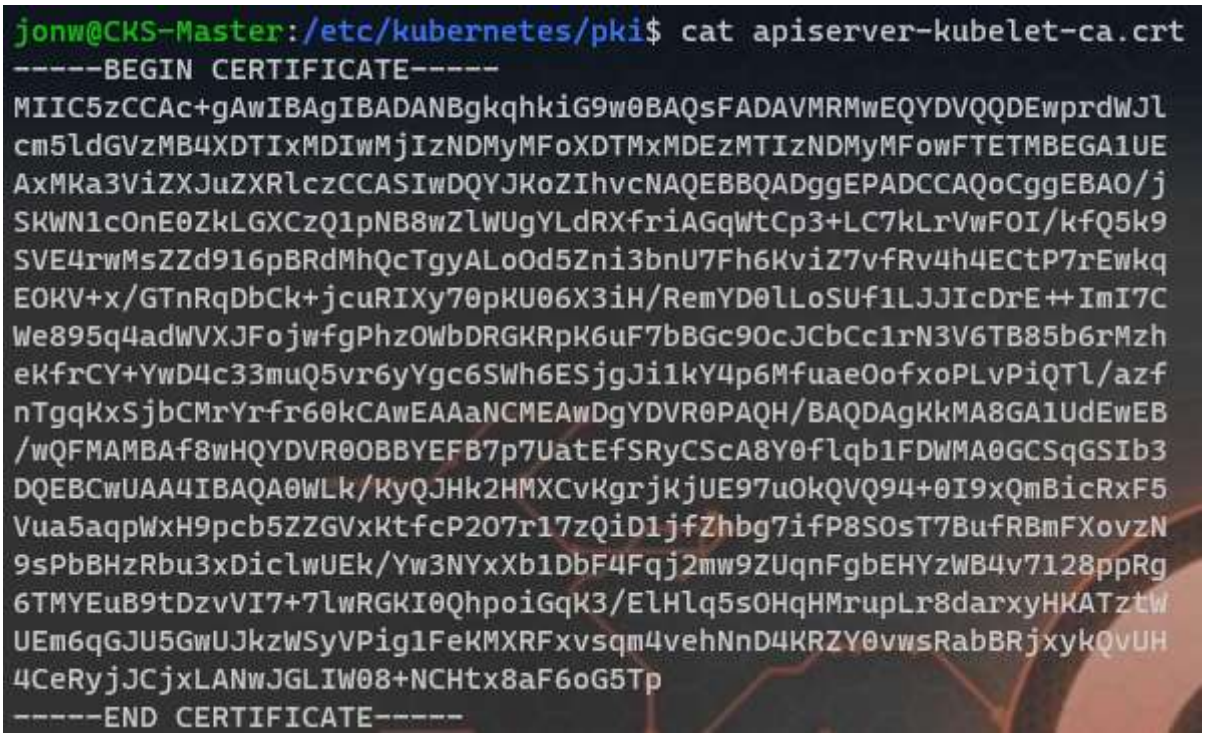
You require TLS to be configured on apiserver as well as kubelets.

The description matches and it also provides us the way to remediate. Let's try the steps in the cluster. First, we show kubelet configuration with

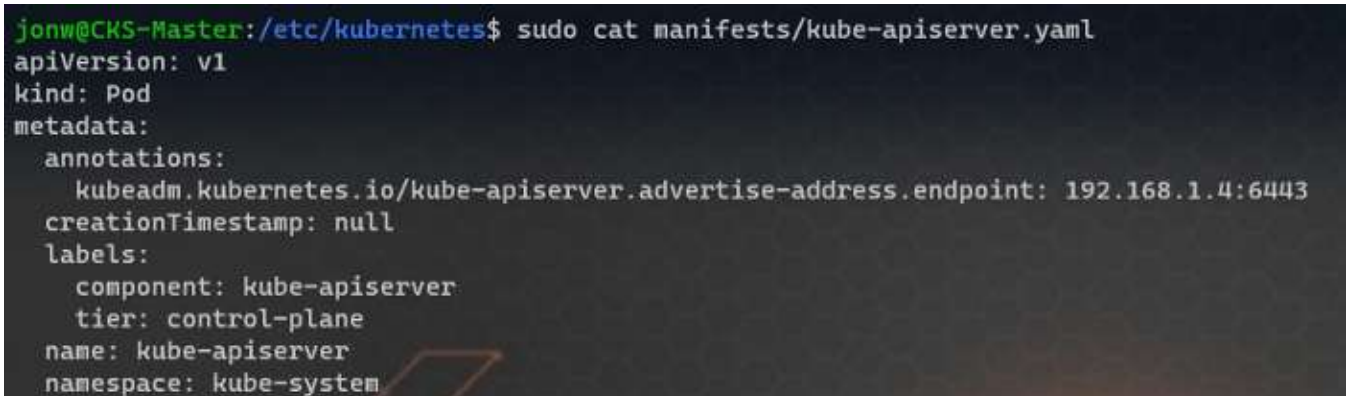
- *sudo cat /etc/kubernetes/kubelet.conf*

We copied the content in “certificate-authority-data” and come outside of the text editor to decode the base 64 encoding on this content. Also, we save the content to /etc/kubernetes/pki/apiserver-kubelet-ca.crt. The command should be similar to

- *echo <copied base64-encoded content> | base64 -d > /etc/kubernetes/pki/apiserver-kubelet-ca.crt*



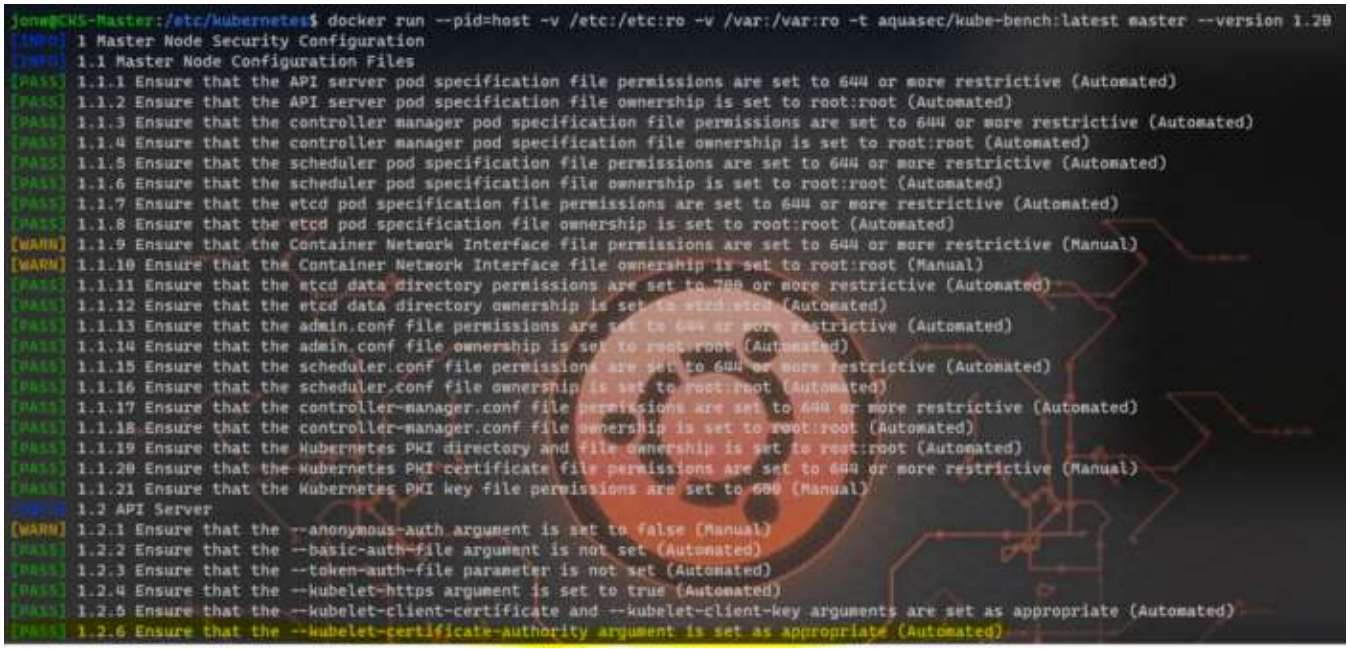
Now, we head over to modify kube-apiserver.yaml in /etc/kubernetes/manifests/kube-apiserver.yaml, so the communication between kube-apiserver and kubelet could leaverage the newly added CA information.



After kube-apiserver restarts, check by running

- *kubectl get pods -n kube-system | grep kube-apiserver*

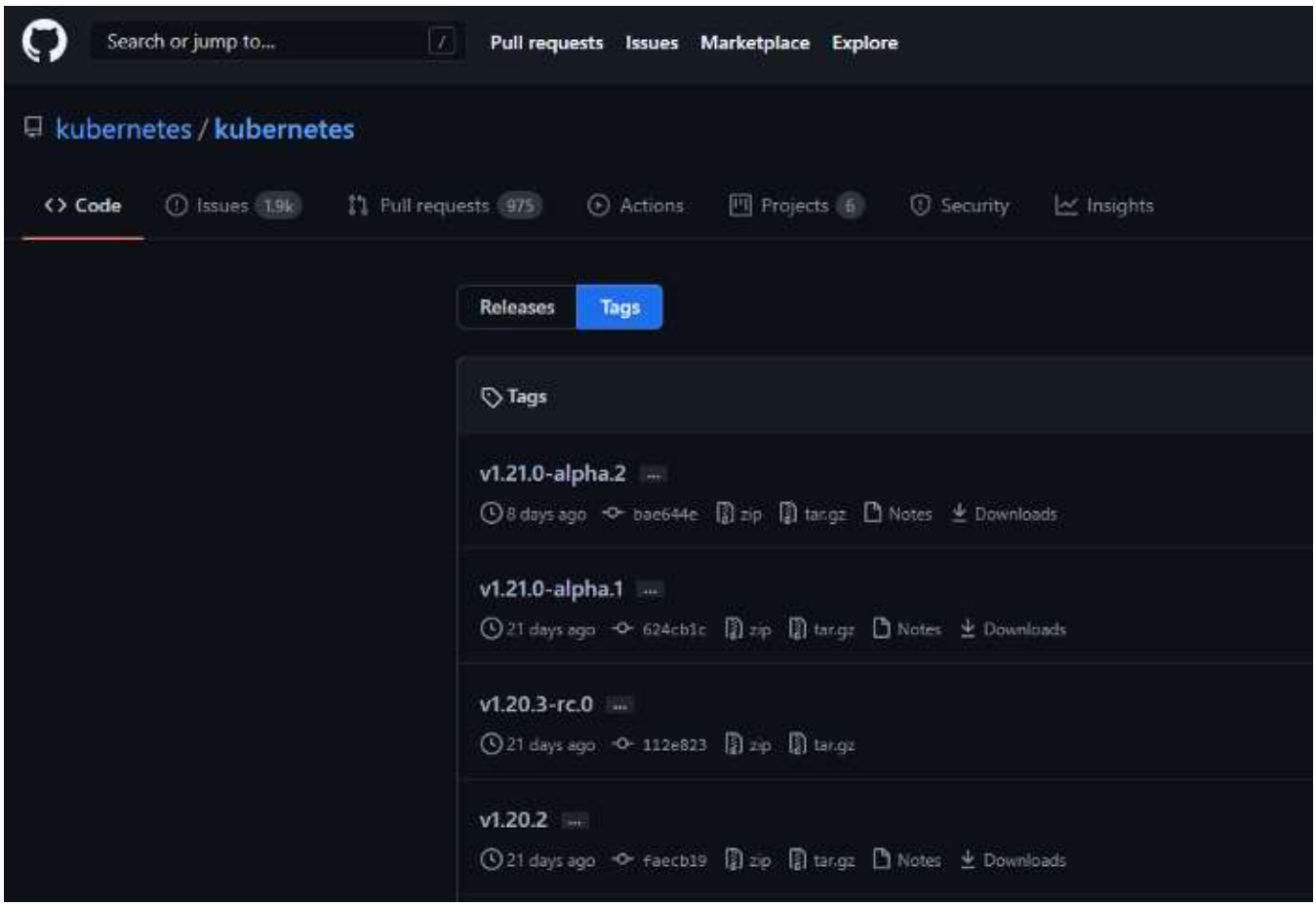
We rerun the kube-bench testing to ensure now [FAIL] 1.2.6 turns to [PASS] 1.2.6.



Verify Platform Binaries

comparing the downloaded file sha512sum with the string provided in GitHub page.

First, we head to this site and search for the version we would desire to install in K8s clusters.



For me, I would need to know v1.20.2 details, so I click on *v1.20.2*.

Then, click on *CHANGELOG* and find *Server binaries*.

server-linux-amd64.tar.gz	65abf178782e43bc21e8455ffbfdadf6064dbeae3ff704ccf9e13e8acee18235c280b06778e5de4bd702f5507e1870fe38c561366c
kubernetes-server-linux-arm.tar.gz	4cac058ade494999292fa4fff0f46c2f3f614d8a6f63d3f0991ee942c0d0aacf6af9ae8b091ffae23809d5281fb4ae0d0743a94a96d8
kubernetes-server-linux-arm64.tar.gz	f5ef9bc1013b638cdc17071e60c987572185d9aea796c44ddd2c791770debe1f7225898c8704f3484c4802e452bb25590bf0523
kubernetes-server-linux-ppc64le.tar.gz	1d9ad46458137e5490832465c6ed997e6d986746e02985c525a0780c9e91857024d7c7fc2fb6f1dfbabd4613214e3e9abf2c560
kubernetes-server-linux-s390x.tar.gz	d9c907cd4e75441970409a9801cff97529af8517c85d2a1c5e9ab055c1886b0e0fd62457644c1c52998e84e7c91920ee09040be5

Depends on the processor K8s clusters are running on, different server binaries would have different sha512sum. The next step is to download the binaries into server and compare it to the string shown on the site.

- *echo "<copied sha512sum>" > compare*
- *wget <v1.20.2 server binaries>*
- *sha512sum <downloaded file> >> compare*
- *cat compare | uniq*

```
jonw@CKS-Master:~$ echo "f5ef9bc1013b638cdc17071e60c987572185d9aea796c44ddd2c791770debe1f7225898c8704f3484c4802e452bb25590bf05232dfff50da741e7897b6907bfe" > compare
jonw@CKS-Master:~$ wget https://dl.k8s.io/v1.20.2/kubernetes-server-linux-arm64.tar.gz
--2021-02-03 18:39:54--  https://dl.k8s.io/v1.20.2/kubernetes-server-linux-arm64.tar.gz
Resolving dl.k8s.io (dl.k8s.io) ... 34.107.204.206, 2600:1901:0:26f3::
Connecting to dl.k8s.io (dl.k8s.io)|34.107.204.206|:443 ... connected.
HTTP request sent, awaiting response ... 302 Moved Temporarily
Location: https://storage.googleapis.com/kubernetes-release/release/v1.20.2/kubernetes-server-linux-arm64.tar.gz [following]
--2021-02-03 18:39:54--  https://storage.googleapis.com/kubernetes-release/release/v1.20.2/kubernetes-server-linux-arm64.tar.gz
Resolving storage.googleapis.com (storage.googleapis.com) ... 74.125.195.128, 74.125.20.128, 74.125.142.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.195.128|:443 ... connected.
HTTP request sent, awaiting response ... 200 OK
Length: 291791433 (278M) [application/x-tar]
Saving to: 'kubernetes-server-linux-arm64.tar.gz'

kubernetes-server-linux-arm64.tar.g 100%[=========================================================>] 278.27M   114MB/s    in 2.4s

2021-02-03 18:39:57 (114 MB/s) - 'kubernetes-server-linux-arm64.tar.gz' saved [291791433/291791433]

jonw@CKS-Master:~$ sha512sum kubernetes-server-linux-arm64.tar.gz >> compare
jonw@CKS-Master:~$ cat compare
f5ef9bc1013b638cdc17071e60c987572185d9aea796c44ddd2c791770debe1f7225898c8704f3484c4802e452bb25590bf05232dfff50da741e7897b6907bfe
f5ef9bc1013b638cdc17071e60c987572185d9aea796c44ddd2c791770debe1f7225898c8704f3484c4802e452bb25590bf05232dfff50da741e7897b6907bfe  kubernetes-server-linux-arm64.tar.gz
```

This articles briefly covers the main topics within the CKS certificate exam domain, cluster setup. We would be talking about the next domain, cluster hardening, in the next article.

Ck Kubernetes Cluster Preparation Exam