ATTACKING KUBERNETES

WITH SECURITY BEST PRACTICE

WWW.DEVSECOPSGUIDES.COM

Attacking Kubernetes

• Apr 15, 2024 • 31 min read

Table of contents

- KUBECONFIG FILE
- VULNERABLE APPLICATION
- EXPOSED DASHBOARD

- Execution
  - EXEC INTO CONTAINER:
  - NEW CONTAINER:
  - APPLICATION EXPLOIT:
  - SSH SERVER RUNNING INSIDE CONTAINER:
- Persistence
  - BACKDOOR CONTAINER:
  - WRITABLE HOSTPATH MOUNT:
  - KUBERNETES CRONJOB:
- Privilege escalation
  - PRIVILEGED CONTAINER
  - CLUSTER-ADMIN BINDING
  - HOSTPATH MOUNT
  - ACCESSING CLOUD RESOURCES:
- Defense evasion
  - CLEAR CONTAINER LOGS:
  - DELETE KUBERNETES EVENTS:
  - POD/CONTAINER NAME SIMILARITY:
  - CONNECT FROM PROXY SERVER
- Credential access
  - LIST KUBERNETES SECRETS:
  - MOUNT SERVICE PRINCIPAL:
  - ACCESS CONTAINER SERVICE ACCOUNT:
  - APPLICATION CREDENTIALS IN CONFIGURATION FILES:
- Discovery

Show less

In the ever-evolving landscape of cybersecurity, Kubernetes has emerged as a dominant force in managing containerized applications. However, its widespread adoption has also made it a prime target for attackers seeking to exploit vulnerabilities and gain unauthorized access. As organizations increasingly rely on Kubernetes for orchestrating their cloud-native environments, safeguarding these clusters against malicious actors becomes paramount. Implementing robust security best practices is essential to fortify Kubernetes deployments and mitigate potential risks.

Securing Kubernetes begins with a comprehensive understanding of its architecture and potential attack vectors. From ensuring proper authentication and authorization mechanisms to implementing network policies and encryption protocols, a layered approach to defense is indispensable. Moreover, continuous monitoring, timely updates, and adherence to industry standards such as CIS benchmarks are fundamental in maintaining the integrity and resilience of Kubernetes clusters. By integrating these security measures into the development and operational workflows, organizations can bolster their defenses and thwart potential threats aimed at compromising Kubernetes environments.

Restrict Kubernetes API access to specific IP ranges

Attackers seeking to bypass restrictions on Kubernetes API access to specific IP ranges may exploit vulnerabilities in the cluster's configuration. By manipulating the `spec.loadBalancerSourceRanges` parameter of the Kubernetes service, attackers can potentially evade IP-based restrictions and gain unauthorized access to the API server. The following command demonstrates how an attacker might attempt to modify this parameter using `kubectl`:

```
kubectl edit svc/kubernetes
```

Upon executing this command, the attacker gains access to the Kubernetes service configuration in the default editor. Within the configuration, they can locate and modify the `spec.loadBalancerSourceRanges` field to allow traffic from unauthorized IP addresses. By removing or adding IP ranges to this field, the attacker can effectively bypass the intended security controls, granting themselves unrestricted access to the Kubernetes API.

```
spec:
  loadBalancerSourceRanges:
```

```
  - 0.0.0.0/0  # Allow all IP addresses (example of bypassing security
control)
```

Use Role-Based Access Control (RBAC)

Attackers aiming to bypass Role-Based Access Control (RBAC) in Kubernetes may exploit misconfigurations or weaknesses in the RBAC setup to escalate privileges and gain unauthorized access to sensitive resources. By leveraging the `kubectl create serviceaccount` and `kubectl create clusterrolebinding` commands, attackers can create new service accounts and bind them to cluster-wide roles, potentially granting themselves elevated privileges beyond their intended scope.

COPY

```
kubectl create serviceaccount <name>
kubectl create clusterrolebinding <name> --clusterrole=<role> --
serviceaccount=<namespace>:<name>
```

In executing these commands, attackers create a new service account named `<name>` and bind it to a specified cluster role using a cluster role binding named `<name>`. This effectively associates the new service account with the permissions defined by the specified cluster role, granting the account access to resources and capabilities within the Kubernetes cluster according to the role's permissions.

COPY

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <name>
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: <role>
subjects:
- kind: ServiceAccount
```

```
  name: <name>
  namespace: <namespace>
```

In the resulting RBAC configuration, the cluster role binding `<name>` associates the service account `<name>` in the specified namespace `<namespace>` with the cluster role `<role>`, effectively granting the associated permissions to the service account.

Enable PodSecurityPolicy (PSP)

Attackers aiming to bypass PodSecurityPolicy (PSP) controls in Kubernetes may exploit vulnerabilities or misconfigurations to escalate privileges and execute malicious actions within the cluster. By utilizing the `kubectl create serviceaccount` and `kubectl create clusterrolebinding` commands, attackers can create a new service account and bind it to a cluster role associated with PSP, potentially granting themselves elevated privileges beyond the intended restrictions.

```
kubectl create serviceaccount psp-sa
kubectl create clusterrolebinding psp-binding --
clusterrole=psp:vmxnet3 --serviceaccount=default:psp-sa
```

In executing these commands, attackers create a new service account named `psp-sa` and bind it to a cluster role named `psp:vmxnet3` using a cluster role binding named `psp-binding`. This effectively associates the new service account with permissions defined by the specified PSP cluster role, potentially granting it escalated privileges related to the use of VMXNET3 network interfaces.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: psp-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
```

```
  kind: ClusterRole
  name: psp:vmxnet3
subjects:
- kind: ServiceAccount
  name: psp-sa
  namespace: default
```

In the resulting RBAC configuration, the cluster role binding `psp-binding` associates the service account `psp-sa` in the default namespace with the PSP cluster role `psp:vmxnet3`, potentially granting the associated permissions to the service account.

Use Network Policies

Attackers may attempt to bypass Network Policies in Kubernetes to gain unauthorized access to network resources or to compromise the security of the cluster. By employing the `kubectl apply -f` command, attackers can apply a maliciously crafted Network Policy YAML file (`networkpolicy.yml`) to potentially circumvent existing network restrictions and carry out malicious activities within the cluster.

COPY

```
kubectl apply -f networkpolicy.yml
```

In executing this command, the attacker applies the Network Policy defined in the YAML file to the Kubernetes cluster. The content of the `networkpolicy.yml` file may include rules that allow traffic from unauthorized sources or permit communication between pods that should be restricted.

COPY

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector: {}
  ingress:
```

```
  - from:
    - namespaceSelector:
        matchLabels:
          name: default
    - podSelector:
        matchLabels:
          app: example-app
  policyTypes:
  - Ingress
```

In the above example, the Network Policy named `allow-all` allows all ingress traffic from pods labeled with `app: example-app` in the default namespace. Attackers can exploit such misconfigured or overly permissive Network Policies to bypass intended security controls and communicate with pods or services that they should not have access to.

Enable Audit Logging

Attackers aiming to bypass audit logging controls in Kubernetes may exploit vulnerabilities or misconfigurations to conceal their actions and evade detection. By utilizing commands such as `kubectl apply` and `kubectl edit`, attackers can attempt to modify the audit logging configuration to disable or manipulate the logging of their activities within the cluster.

```
COPY
kubectl apply -f audit-policy.yaml
kubectl edit cm/kube-apiserver -n kube-system
```

In executing these commands, attackers apply a custom audit policy defined in the YAML file `audit-policy.yaml` and edit the ConfigMap `kube-apiserver` in the `kube-system` namespace, where the audit logging configuration is stored.

The `audit-policy.yaml` file may contain a custom audit policy defining which events should be logged:

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
```

The `kubectl edit` command allows attackers to modify parameters related to audit logging, such as `--audit-log-path` and `--audit-policy-file`, potentially altering the destination and content of audit logs or disabling auditing altogether.

```
apiVersion: v1
data:
  audit-log-path: /var/log/kubernetes/audit.log
  audit-policy-file: /etc/kubernetes/audit-policy.yaml
```

By manipulating these parameters, attackers may attempt to divert audit logs to unauthorized locations, suppress logging of their activities, or modify the audit policy to exclude events associated with their malicious actions, thereby obscuring their presence and evading detection by security monitoring systems.

Use Secure Service Endpoints

Attackers may attempt to bypass secure service endpoint controls in Kubernetes to exploit vulnerabilities or gain unauthorized access to services within the cluster. By leveraging the `kubectl patch` command, attackers can potentially manipulate the configuration of a service, such as enabling the publishing of not ready addresses and setting session affinity to `ClientIP`, in an attempt to circumvent security measures.

```
kubectl patch svc <svc-name> -p '{"spec":
{"publishNotReadyAddresses": true, "sessionAffinity": "ClientIP"}}'
```

In executing this command, the attacker patches the configuration of the specified service (`<svc-name>`) with the provided JSON patch. This patch instructs Kubernetes to publish not ready addresses and set session affinity to `ClientIP`, potentially enabling attackers to access services even when they are not ready or to manipulate session affinity to their advantage.

```
{
  "spec": {
    "publishNotReadyAddresses": true,
    "sessionAffinity": "ClientIP"
  }
}
```

By enabling the publishing of not ready addresses, attackers may exploit services that are still in the process of initialization or recovery, potentially bypassing intended restrictions on access. Setting session affinity to `ClientIP` may allow attackers to manipulate session handling to maintain persistence or evade detection.

Use Pod Security Context

Attackers may attempt to bypass Pod Security Context controls in Kubernetes to escalate privileges and execute malicious actions within the cluster. By leveraging the `kubectl create` command, attackers can create a new service account and bind it to a role associated with Pod Security Policies (PSP), potentially granting themselves elevated privileges beyond the intended restrictions.

```
kubectl create sa pod-sa
kubectl create rolebinding pod-sa --role=psp:vmxnet3 --
serviceaccount=default:pod-sa
```

In executing these commands, attackers create a new service account named `pod-sa` and bind it to a role named `psp:vmxnet3` using a role binding named `pod-sa`. This

effectively associates the new service account with permissions defined by the specified PSP role, potentially granting it escalated privileges related to the use of VMXNET3 network interfaces.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-sa
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: psp:vmxnet3
subjects:
- kind: ServiceAccount
  name: pod-sa
  namespace: default
```

In the resulting RBAC configuration, the role binding `pod-sa` associates the service account `pod-sa` in the default namespace with the PSP role `psp:vmxnet3`, potentially granting the associated permissions to the service account.

Use Kubernetes Secrets

Attackers may attempt to exploit Kubernetes Secrets to gain unauthorized access to sensitive information stored within the cluster. By leveraging the `kubectl create secret` command, attackers can create a new secret and potentially bypass security controls to exfiltrate confidential data.

```
kubectl create secret generic <name> --from-file=<path-to-file>
```

In executing this command, attackers create a new secret named `<name>` and populate it with the contents of a file located at `<path-to-file>`. This file may contain

sensitive information such as passwords, API keys, or certificates.

By gaining access to create secrets within the cluster, attackers can potentially extract sensitive data and use it to escalate privileges, access restricted resources, or carry out further attacks. Additionally, if secrets are not adequately protected or encrypted, attackers may be able to intercept or manipulate them, compromising the security of the entire system.

Enable Container Runtime Protection

Attackers may attempt to bypass container runtime protection in Kubernetes to evade detection and execute malicious activities within containers. By leveraging the `kubectl apply` command, attackers can apply a maliciously crafted configuration file, such as `falco.yaml`, to potentially circumvent existing container runtime security measures.

```
kubectl apply -f falco.yaml
```

In executing this command, the attacker applies the configuration defined in the YAML file `falco.yaml` to the Kubernetes cluster. This configuration may include rules and policies for container runtime protection, such as monitoring for abnormal behaviors or unauthorized access attempts.

The content of the `falco.yaml` file may define rules for detecting suspicious activities within containers:

```
apiVersion: falco.org/v1
kind: FalcoRule
metadata:
  name: suspicious-executable
spec:
  description: Detects execution of suspicious executables
  condition: >
```

```
    (proc.name = bash or proc.name = sh) and
    (fd.name contains /tmp/evil)
  output: "Suspicious executable detected: %{proc.name} executing %
{fd.name}"
  priority: WARNING
  enabled: true
```

In the above example, a Falco rule named `suspicious-executable` is defined to detect the execution of suspicious executables within containers. When triggered, this rule outputs a warning message indicating the detected activity.

Attackers may attempt to bypass container runtime protection by evading detection mechanisms defined in the Falco configuration or by exploiting vulnerabilities in the monitoring system itself. They may modify the configuration to exclude detection of specific activities or to generate false positives, thereby obscuring their actions and evading detection.

Enable Admission Controllers

Attackers may attempt to bypass admission controllers in Kubernetes to execute unauthorized actions or introduce malicious resources into the cluster. By leveraging the `kubectl edit` command, attackers can potentially modify the configuration of the kube-apiserver to disable or manipulate admission controllers, thereby bypassing intended security controls.

```
kubectl edit cm/kube-apiserver -n kube-system
```

In executing this command, the attacker edits the ConfigMap `kube-apiserver` in the `kube-system` namespace, where the configuration settings for the kube-apiserver component are stored. The attacker may modify the `--enable-admission-plugins` parameter to manipulate the admission controllers.

```
apiVersion: v1
data:
  enable-admission-plugins:
"ValidatingAdmissionWebhook,AlwaysPullImages"
```

In the edited configuration, the attacker may disable certain admission controllers or add new ones to allow the admission of resources that would otherwise be rejected by default controls. For example, disabling the `PodSecurityPolicy` admission controller can allow the admission of pods that violate security policies.

By manipulating admission controller settings, attackers can potentially bypass security controls and introduce malicious resources or execute unauthorized actions within the Kubernetes cluster. This can lead to a variety of security risks, including privilege escalation, data exfiltration, or the execution of malicious code.

Hardcoded Credential

Attackers may exploit hardcoded credentials in Kubernetes configurations to gain unauthorized access to sensitive resources, such as databases, and execute malicious activities. By embedding credentials directly into configuration files, attackers can easily extract and abuse these credentials if they gain access to the configuration files.

COPY

```
apiVersion: v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: my-app
    spec:
```

```
      containers:
      - name: my-app-container
        image: my-app:v1
        ports:
        - containerPort: 8080
        env:
        - name: DATABASE_URL
          value: "mysql://root:password@localhost:3306/my_database"
```

In this noncompliant code, the Kubernetes Deployment configuration contains a hardcoded database connection string in the env section. The database URL, including the username (`root`), password (`password`), and other sensitive details, is directly embedded in the configuration file, posing a significant security risk.

To address this vulnerability, Kubernetes provides a solution through Secrets, which allows for the secure storage and management of sensitive information.

COPY

```
apiVersion: v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app-container
        image: my-app:v1
        ports:
        - containerPort: 8080
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
```

```
              name: my-app-secrets
              key: database-url
```

In the compliant code, the hardcoded database connection string is replaced with a reference to a Kubernetes Secret named `my-app-secrets`. The Secret contains the sensitive information such as the database URL, username, and password. The `valueFrom` field in the env section instructs Kubernetes to retrieve the value of the `database-url` key from the specified Secret.

By leveraging Secrets, organizations can centralize and securely manage sensitive information in Kubernetes, preventing hardcoded vulnerabilities. Secrets can be encrypted, access-controlled, and rotated more easily compared to hardcoded values.

Container Escape Attack

Attackers may exploit container escape vulnerabilities in Kubernetes to break out of containerized environments and gain unauthorized access to the underlying host system. By running containers with extended privileges, attackers can potentially bypass security controls and execute malicious actions on the host system.

COPY

```
apiVersion: v1
kind: Pod
metadata:
  name: privileged-pod
spec:
  containers:
  - name: privileged-container
    image: my-image
    securityContext:
      privileged: true
```

In this noncompliant code, the Pod definition sets the privileged flag to true, allowing the container to run with extended privileges. This configuration increases the attack

surface and makes it easier for an attacker to escape the container and compromise the underlying host system.

To mitigate container escape attacks, Kubernetes provides security features such as the securityContext field, which allows administrators to control the security settings of containers.

```
apiVersion: v1
kind: Pod
metadata:
  name: restricted-pod
spec:
  containers:
  - name: restricted-container
    image: my-image
    securityContext:
      privileged: false
```

In the compliant code, the privileged flag is set to false, restricting the container from running with extended privileges. By enforcing this security measure, organizations can reduce the risk of container escape attacks and enhance the security of their Kubernetes deployments.

Kubernetes API Server Attack

Attackers may exploit vulnerabilities in Kubernetes API server configurations to gain unauthorized access to sensitive resources and execute malicious actions within the cluster. One common attack vector involves the creation of privileged service accounts without proper Role-Based Access Control (RBAC) restrictions, allowing attackers to abuse their wide-ranging access to the Kubernetes API server.

```
apiVersion: v1
kind: ServiceAccount
metadata:
```

```
  name: privileged-service-account
  namespace: default
```

In this noncompliant code, a privileged service account named `privileged-service-account` is created without specifying any RBAC restrictions. This service account inherits wide-ranging permissions by default, potentially granting attackers unrestricted access to the Kubernetes API server.

To mitigate the risk of Kubernetes API server attacks, it's crucial to implement RBAC controls to restrict the permissions of service accounts.

COPY

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: restricted-service-account
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: restricted-role
  namespace: default
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: restricted-role-binding
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: restricted-role
subjects:
```

```
- kind: ServiceAccount
  name: restricted-service-account
  namespace: default
```

In the compliant code, a restricted service account named `restricted-service-account` is created, and RBAC rules are applied to limit its access. Specifically, a Role named `restricted-role` is defined, granting permissions to get, list, and watch pods. A RoleBinding named `restricted-role-binding` associates the Role with the restricted service account.

Pod-to-Pod Network Attack

Attackers may exploit vulnerabilities in Kubernetes pod networking configurations to carry out pod-to-pod network attacks, potentially compromising the security and integrity of the cluster. One common attack scenario involves deploying multiple containers within the same pod without implementing any network policies or restrictions, allowing unrestricted communication between the containers.

COPY

```
apiVersion: v1
kind: Pod
metadata:
  name: unsecured-pod
spec:
  containers:
  - name: container-a
    image: image-a
  - name: container-b
    image: image-b
```

In this noncompliant code, two containers (`container-a` and `container-b`) are deployed within the same pod (`unsecured-pod`) without any network policies or restrictions. This setup allows unrestricted communication between the containers, potentially exposing sensitive data or services to unauthorized access or interception.

To mitigate the risk of pod-to-pod network attacks, it's crucial to implement network policies to restrict communication between containers within the pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: secured-pod
spec:
  containers:
  - name: container-a
    image: image-a
  - name: container-b
    image: image-b
  networkPolicy:
    podSelector:
      matchLabels:
        app: secured-pod
    ingress:
    - from:
        podSelector:
          matchLabels:
            app: secured-pod
```

In the compliant code, network policies are introduced to restrict communication between containers within the pod. Both `container-a` and `container-b` are part of the `secured-pod`, and the network policy ensures that only pods labeled as `secured-pod` can initiate ingress traffic to this pod. This setup effectively limits the attack surface and prevents unauthorized access or interception of network traffic from other pods.

Privilege Escalation Attack

Attackers may exploit privilege escalation vulnerabilities in Kubernetes to elevate their privileges within containers and gain unauthorized access to sensitive resources or compromise the security of the cluster. One common attack vector involves running

containers with the root user, providing extensive privileges and increasing the risk of privilege escalation attacks.

```
apiVersion: v1
kind: Pod
metadata:
  name: privileged-pod
spec:
  containers:
  - name: privileged-container
    image: my-image
    securityContext:
      runAsUser: 0
```

In this noncompliant code, the `runAsUser` field is set to 0, which runs the container as the root user. Running containers as root grants extensive privileges within the container, making it easier for attackers to exploit privilege escalation vulnerabilities and gain unauthorized access to sensitive resources or the underlying host system.

To mitigate the risk of privilege escalation attacks, Kubernetes provides security features such as the `securityContext` field, which allows administrators to specify security settings for containers.

```
apiVersion: v1
kind: Pod
metadata:
  name: restricted-pod
spec:
  containers:
  - name: restricted-container
    image: my-image
    securityContext:
      runAsUser: 1000
```

In the compliant code, the `runAsUser` field is set to a non-root user (e.g., UID 1000), reducing the container's privileges and mitigating the risk of privilege escalation attacks. By running containers with non-root users, organizations can enforce the principle of least privilege, limiting the impact of potential security breaches and enhancing the overall security posture of their Kubernetes deployments.

Denial-of-Service (DoS) Attack

Attackers may exploit vulnerabilities in Kubernetes resource configurations to carry out Denial-of-Service (DoS) attacks, causing resource exhaustion and disrupting the availability of services within the cluster. One common attack scenario involves specifying resource requests that are significantly higher than necessary, leading to inefficient resource utilization and potential DoS vulnerabilities.

COPY

```
apiVersion: v1
kind: Deployment
metadata:
  name: resource-hungry-app
spec:
  replicas: 5
  template:
    spec:
      containers:
      - name: resource-hungry-container
        image: my-image
        resources:
          requests:
            cpu: "1000m"
            memory: "2Gi"
```

In this noncompliant code, the resource requests for the container are set significantly higher than necessary, with requests of 1000 milliCPU and 2 gigabytes of memory. Such excessive resource requests can lead to inefficient resource utilization and potential DoS vulnerabilities, as the containers may consume more resources than required, leading to resource exhaustion and service disruption.

To mitigate the risk of DoS attacks, it's crucial to set resource requests to more appropriate values that accurately reflect the actual resource requirements of the containers.

```
apiVersion: v1
kind: Deployment
metadata:
  name: optimized-app
spec:
  replicas: 5
  template:
    spec:
      containers:
      - name: optimized-container
        image: my-image
        resources:
          requests:
            cpu: "100m"
            memory: "256Mi"
```

In the compliant code, resource requests are set to more appropriate values, with requests of 100 milliCPU and 256 megabytes of memory. By accurately specifying resource requests based on the actual resource requirements of the containers, organizations can mitigate the risk of DoS attacks and ensure efficient resource utilization within the Kubernetes cluster.

Kubernetes Threat Matrix

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8S secrets	Access the K8S API server	Access cloud resources	Data Destruction
Compromised images in registry	bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8S events	Mount service principal	Access Kubelet API	Container service account	Resource Hijacking
Kubeconfig file	New container	Kubernetes CronJob	hostPath mount	Pod / container name similarity	Access container service account	Network mapping	Cluster internal networking	Denial of service
Application vulnerability	Application exploit (RCE)		Access cloud resources	Connect from Proxy server	Applications credentials in configuration files	Access Kubernetes dashboard	Applications credentials in configuration files	
Exposed Dashboard	SSH server running inside container					Instance Metadata API	Writable volume mounts on the host	
							Access Kubernetes dashboard	
							Access tiller endpoint	

The Threat Matrix highlights various attack techniques, including both known and hypothetical scenarios, that could be exploited by adversaries targeting Kubernetes environments. It categorizes these techniques into different stages of the attack lifecycle, such as initial access, privilege escalation, lateral movement, persistence, and exfiltration.

Initial access

As organizations embrace containerized environments like Kubernetes, it becomes essential to understand the potential vulnerabilities and attack vectors that adversaries may exploit. The initial access tactic poses a significant threat, serving as the entry point for unauthorized actors into Kubernetes clusters. In this article, we will explore some common techniques used to gain initial access and discuss proactive measures to secure your Kubernetes environment.

USING CLOUD CREDENTIALS

In cloud-based Kubernetes deployments, compromised cloud credentials can spell disaster. Attackers who gain access to cloud account credentials can infiltrate the cluster's management layer, potentially leading to complete cluster takeover. It is crucial to implement robust cloud security practices, such as strong access controls

and multi-factor authentication, to safeguard against unauthorized access to cloud credentials.

COMPROMISED IMAGES IN REGISTRY

Running compromised container images within a cluster can introduce significant risks. Attackers with access to a private registry can inject their own compromised images, which can then be inadvertently pulled by users. Additionally, using untrusted images from public registries without proper validation can expose the cluster to malicious content. Employing image scanning and verifying the trustworthiness of container images can help mitigate this risk.

KUBECONFIG FILE

The kubeconfig file, which contains cluster details and credentials, is used by Kubernetes clients like kubectl. If an attacker gains access to this file, they can exploit it to gain unauthorized access to the Kubernetes clusters. Securing the kubeconfig file through secure distribution channels, enforcing access controls, and employing secure client environments are essential steps to mitigate this risk.

VULNERABLE APPLICATION

Running a vulnerable application within a cluster can open the door to initial access. Exploiting remote code execution vulnerabilities in containers can allow attackers to execute arbitrary code. If a service account is mounted to the compromised container, the attacker can use its credentials to send requests to the Kubernetes API server. Regularly patching and updating container images, along with implementing strong network segmentation, are crucial to mitigating this risk.

EXPOSED DASHBOARD

The Kubernetes dashboard, when exposed externally without proper authentication and access controls, becomes a potential entry point for unauthorized access. Attackers can exploit an exposed dashboard to gain remote management capabilities over the cluster. It is essential to restrict access to the dashboard, enable authentication, and ensure it is accessible only through secure connections.

Execution

Once attackers gain initial access to a Kubernetes cluster, the execution tactic becomes their next focus. By leveraging various techniques, attackers attempt to run their malicious code within the cluster, potentially causing widespread damage. In this article, we will explore common execution techniques in Kubernetes and discuss key strategies to mitigate the associated risks.

EXEC INTO CONTAINER:

Attackers with sufficient permissions can exploit the “exec” command (“kubectl exec”) to run malicious commands inside containers within the cluster. By using legitimate images, such as popular OS images, as a backdoor container, attackers can remotely execute their malicious code through “kubectl exec.” Limiting permissions and enforcing strict access controls will help prevent unauthorized execution within containers.

NEW CONTAINER:

Attackers with permissions to deploy pods or controllers, like DaemonSets, ReplicaSets, or Deployments, may attempt to create new resources within the cluster for running their code. It is crucial to regularly audit and review access controls, ensuring that only authorized entities can create and deploy containers. Monitoring the creation of new resources and implementing least privilege principles will limit unauthorized code execution.

APPLICATION EXPLOIT:

Exploiting vulnerabilities in applications deployed within the cluster presents an opportunity for attackers to execute their code. Vulnerabilities that allow remote code execution or enable unauthorized access to resources can be leveraged. Mounting service accounts to containers, which is the default behavior in Kubernetes, may grant attackers the ability to send requests to the API server using compromised service account credentials. Regular patching and vulnerability management are crucial to mitigating this risk.

SSH SERVER RUNNING INSIDE CONTAINER:

In some cases, attackers may discover containers running SSH servers. If attackers acquire valid credentials, either through brute-force attempts or phishing, they can

exploit these SSH servers to gain remote access to the container. To mitigate this risk, it is essential to employ strong authentication mechanisms, enforce secure credential management practices, and regularly audit containers for unauthorized SSH servers.

Persistence

In the context of Kubernetes security, persistence refers to the techniques employed by attackers to maintain access to a cluster even after their initial entry point has been compromised. By understanding and addressing the persistence tactics used by adversaries, organizations can strengthen their security posture and protect their Kubernetes environments. In this article, we will explore common persistence techniques in Kubernetes and discuss strategies to mitigate these risks.

BACKDOOR CONTAINER:

One method attackers employ to establish persistence is by running malicious code within a container in the cluster. By leveraging Kubernetes controllers like DaemonSets or Deployments, attackers can ensure that a specific number of containers constantly run on one or more nodes in the cluster. To counter this, regular monitoring of controller configurations and thorough auditing of container images can help detect and remove unauthorized backdoor containers.

WRITABLE HOSTPATH MOUNT:

The hostPath volume allows mounting a directory or file from the host to a container. Attackers with permissions to create containers within the cluster can exploit this feature by creating a container with a writable hostPath volume. This provides them with persistence on the underlying host and potential avenues for unauthorized access. Implementing strict access controls and regular auditing of container configurations can help identify and mitigate this risk.

KUBERNETES CRONJOB:

Kubernetes CronJob is a scheduling mechanism used to run Jobs at specified intervals. Attackers may leverage Kubernetes CronJob functionality to schedule the execution of malicious code as a container within the cluster. This allows them to

maintain persistence by regularly running their code. Monitoring and reviewing CronJob configurations, as well as conducting periodic vulnerability scans, are crucial in identifying and addressing any unauthorized or suspicious CronJobs.

Privilege escalation

Privilege escalation is a critical tactic employed by attackers to gain higher privileges within a Kubernetes environment. By obtaining elevated access, attackers can potentially compromise the entire cluster, breach cloud resources, and disrupt critical operations. Understanding common privilege escalation techniques is crucial for implementing effective security measures. In this article, we will explore common privilege escalation techniques in Kubernetes and discuss strategies to mitigate these risks.

PRIVILEGED CONTAINER

A privileged container possesses all the capabilities of the host machine, allowing unrestricted actions within the cluster. Attackers who gain access to a privileged container, or have permissions to create one, can exploit the host's resources. It is essential to enforce strict container security policies, limit the creation of privileged containers, and regularly monitor for unauthorized access or configuration changes.

CLUSTER-ADMIN BINDING

Role-based access control (RBAC) is a fundamental security feature in Kubernetes, controlling the actions of different identities within the cluster. Cluster-admin is a built-in high-privileged role in Kubernetes. Attackers with permissions to create bindings and cluster-bindings can create a binding to the cluster-admin ClusterRole or other high-privileged roles. Implementing least privilege principles, regularly reviewing RBAC configurations, and conducting frequent audits are vital for preventing unauthorized privilege escalation.

HOSTPATH MOUNT

Attackers can leverage the hostPath volume mount to gain access to the underlying host, breaking out of the container's isolated environment. This allows them to escalate privileges from the container to the host. Implementing strict access

controls, conducting regular vulnerability scans, and monitoring for suspicious hostPath mount configurations are essential for mitigating this risk.

ACCESSING CLOUD RESOURCES:

In cloud-based Kubernetes deployments, attackers may leverage their access to a single container to gain unauthorized access to other cloud resources outside the cluster. For instance, in Azure Kubernetes Service (AKS), each node contains a service principal credential used for managing Azure resources. Attackers who gain access to this credential file can exploit it to access or modify cloud resources. Strictly managing access to service principal credentials, encrypting sensitive files, and regularly rotating credentials are critical mitigation steps.

Defense evasion

Defense evasion techniques are employed by attackers to evade detection and conceal their activities within Kubernetes environments. By actively evading security measures, attackers can prolong their presence, increase the likelihood of successful attacks, and bypass traditional security controls. Understanding common defense evasion techniques is crucial for organizations to enhance threat detection capabilities and bolster overall Kubernetes security. In this article, we will explore common defense evasion tactics and discuss strategies to mitigate these risks effectively.

CLEAR CONTAINER LOGS:

Attackers may attempt to delete application or operating system logs on compromised containers to conceal their malicious activities. Organizations should implement robust log management practices, including centralizing logs and establishing secure backup mechanisms. Regularly monitoring log files for suspicious activities and implementing access controls to prevent unauthorized log modifications are vital to maintain visibility into container activities.

DELETE KUBERNETES EVENTS:

Kubernetes events play a critical role in logging state changes and failures within the cluster. Attackers may seek to delete Kubernetes events to avoid detection of their

activities. Organizations should ensure proper event logging and implement log integrity checks to detect any tampering or deletion of events. Retaining logs in a secure and immutable manner can aid in the identification of anomalous behavior.

POD/CONTAINER NAME SIMILARITY:

Attackers may attempt to hide their malicious activities by naming their backdoor pods in a way that resembles legitimate pods created by controllers like Deployments or DaemonSets. By blending in with existing pod naming conventions, attackers aim to avoid suspicion. Organizations should implement strict naming conventions and conduct regular audits to identify any discrepancies or suspicious pod/container names.

CONNECT FROM PROXY SERVER

To obfuscate their origin IP addresses, attackers may employ proxy servers, including anonymous networks like TOR, to communicate with applications or the Kubernetes API server. Organizations should consider implementing network security measures to monitor and restrict access from suspicious IP ranges or anonymous networks. Implementing intrusion detection and prevention systems (IDPS) and conducting regular threat intelligence analysis can aid in identifying proxy server usage by attackers.

Credential access

The security of credentials is of paramount importance in Kubernetes environments. Attackers employ various techniques to steal credentials, including application credentials, service accounts, secrets, and cloud credentials. Safeguarding credential access is crucial to prevent unauthorized access, data breaches, and potential compromise of sensitive information. In this article, we will explore common credential access tactics and discuss strategies to enhance identity protection and mitigate the risks associated with credential theft in Kubernetes.

LIST KUBERNETES SECRETS:

Kubernetes secrets are used to store sensitive information, such as passwords and connection strings, within the cluster. Attackers with appropriate permissions can

retrieve these secrets from the API server, potentially gaining access to critical credentials. Organizations should adopt a defense-in-depth approach to secure secrets, including strong access controls, encryption, and regular auditing of secret configurations. Implementing fine-grained RBAC policies and limiting access to secrets based on the principle of least privilege can help mitigate the risk of unauthorized access.

MOUNT SERVICE PRINCIPAL:

In cloud deployments, attackers may exploit their access to a container in the cluster to gain unauthorized access to cloud credentials. For example, in Azure Kubernetes Service (AKS), each node contains a service principal credential. Organizations should implement robust security measures, such as secure cluster configurations, strict access controls, and regular rotation of service principal credentials, to prevent unauthorized access to cloud resources.

ACCESS CONTAINER SERVICE ACCOUNT:

Service accounts (SAs) are used to represent application identities within Kubernetes. By default, SAs are mounted to every pod in the cluster, allowing containers to interact with the Kubernetes API server. Attackers who gain access to a pod can extract the SA token and potentially perform actions within the cluster based on the SA's permissions. It is crucial to implement RBAC and enforce strong authentication mechanisms to mitigate the risk of unauthorized SA access. Regular audits and monitoring of SA permissions can help identify and remediate any potential security gaps.

APPLICATION CREDENTIALS IN CONFIGURATION FILES:

Developers often store secrets, such as application credentials, in Kubernetes configuration files, including environment variables in the pod configuration. Attackers may attempt to access these configuration files to steal sensitive information. Organizations should promote secure coding practices, such as externalizing secrets to a secure secret management solution, and avoid storing credentials directly in configuration files. Implementing secure coding guidelines, regular security training for developers, and automated vulnerability scanning can help reduce the risk of unauthorized access to application credentials.

Discovery

Discovery attacks pose a significant threat to the security of Kubernetes environments. Attackers employ various techniques to explore the environment, gain insights into the cluster's resources, and perform lateral movement to access additional targets. Understanding and mitigating these discovery tactics is crucial to bolster the overall security posture of Kubernetes deployments. In this article, we will delve into common discovery techniques and discuss strategies to enhance defense and thwart unauthorized exploration in Kubernetes.

ACCESS THE KUBERNETES API SERVER:

The Kubernetes API server acts as the gateway to the cluster, enabling interactions and resource management. Attackers may attempt to access the API server to gather information about containers, secrets, and other resources. Protecting the API server is paramount, and organizations should implement strong authentication mechanisms, robust access controls, and secure communication channels (TLS) to prevent unauthorized access and unauthorized retrieval of sensitive data.

ACCESS KUBELET API:

Kubelet, running on each node, manages the execution of pods and exposes a read-only API service. Attackers with network access to the host can probe the Kubelet API to gather information about running pods and the node itself. To mitigate this risk, organizations should implement network segmentation and restrict network access to the Kubelet API, employing firewalls or network policies to allow communication only from trusted sources.

NETWORK MAPPING:

Attackers may attempt to map the cluster network to gain insights into running applications and identify potential vulnerabilities. Implementing network segmentation, network policies, and utilizing network security solutions can help limit unauthorized network exploration within the cluster, reducing the attack surface and minimizing the impact of network mapping attempts.

ACCESS KUBERNETES DASHBOARD:

The Kubernetes dashboard provides a web-based interface for managing and monitoring the cluster. Attackers who gain access to a container in the cluster may attempt to exploit the container's network access to access the dashboard pod. Organizations should secure the Kubernetes dashboard by implementing strong authentication, role-based access controls (RBAC), and secure network access policies to prevent unauthorized access and information leakage.

INSTANCE METADATA API:

Cloud providers offer instance metadata services that provide information about virtual machine configurations and network details. Attackers who compromise a container may attempt to query the instance metadata API to gain insights into the underlying node. Protecting the metadata API is crucial, and organizations should implement network-level security controls, such as restricting access to the metadata service from within the VM only, to prevent unauthorized access and limit the exposure of sensitive information.

Lateral movement

Lateral movement attacks pose a significant threat in containerized environments, allowing attackers to traverse through a victim's environment, gain unauthorized access to various resources, and potentially escalate privileges. Understanding and mitigating lateral movement tactics is crucial for bolstering the security of Kubernetes deployments. In this article, we will explore common techniques used by attackers for lateral movement and discuss strategies to enhance defense and minimize the impact of these attacks in Kubernetes.

ACCESS THE KUBERNETES API SERVER:

The Kubernetes API server acts as the gateway to the cluster, enabling interactions and resource management. Attackers may attempt to access the API server to gather information about containers, secrets, and other resources. Protecting the API server is paramount, and organizations should implement strong authentication mechanisms, robust access controls, and secure communication channels (TLS) to prevent unauthorized access and unauthorized retrieval of sensitive data.

ACCESS CLOUD RESOURCES:

Attackers who compromise a container in the cluster may attempt to move laterally into the cloud environment itself. Organizations must implement strong access controls, employ least privilege principles, and regularly monitor cloud resources to detect and prevent unauthorized access attempts.

CONTAINER SERVICE ACCOUNT:

Attackers with access to a compromised container can leverage the mounted service account token to send requests to the Kubernetes API server and gain access to additional resources within the cluster. Securing container service accounts through RBAC and regularly rotating credentials can help mitigate the risk of lateral movement through compromised containers.

CLUSTER INTERNAL NETWORKING:

By default, Kubernetes allows communication between pods within the cluster. Attackers who gain access to a single container can leverage this networking behavior to traverse the cluster and target additional resources. Implementing network segmentation, network policies, and regular network monitoring can restrict unauthorized lateral movement within the cluster.

APPLICATION CREDENTIALS IN CONFIGURATION FILES:

Developers often store sensitive credentials in Kubernetes configuration files, such as environment variables in pod configurations. Attackers who gain access to these credentials can use them to move laterally and access additional resources both inside and outside the cluster. Employing secure secrets management practices, such as encrypting configuration files and limiting access to sensitive information, can mitigate the risk of credential-based lateral movement.

WRITABLE VOLUME MOUNTS ON THE HOST:

Attackers may attempt to exploit writable volume mounts within a compromised container to gain access to the underlying host. Securing host-level access controls, implementing strong container isolation, and regularly patching and hardening the underlying host can help mitigate the risk of lateral movement from containers to the host.

ACCESS KUBERNETES DASHBOARD:

Attackers with access to the Kubernetes dashboard can manipulate cluster resources and execute code within containers using the built-in “exec” capability. Securing the Kubernetes dashboard through strong authentication, access controls, and monitoring for suspicious activities can minimize the risk of unauthorized lateral movement through the dashboard.

ACCESS TILLER ENDPOINT:

Tiller, the server-side component of Helm, may expose internal gRPC endpoints that do not require authentication. Attackers who can access a container connected to the Tiller service may exploit this vulnerability to perform unauthorized actions within the cluster. Organizations should consider migrating to Helm version 3, which removes the Tiller component and eliminates this specific risk.

Impact

The Impact tactic in Kubernetes refers to techniques employed by attackers to disrupt, abuse, or destroy the normal behavior of the environment. These attacks can lead to data loss, resource abuse, and denial of service, resulting in severe consequences for organizations. Protecting Kubernetes deployments from such impact attacks is crucial to ensure the availability, integrity, and confidentiality of resources. In this article, we will explore common impact techniques used by attackers and discuss strategies to mitigate their effects in Kubernetes environments.

DATA DESTRUCTION:

Attackers may target Kubernetes deployments to destroy critical data and resources. This can involve deleting deployments, configurations, storage volumes, or compute resources. To mitigate the risk of data destruction, it is essential to implement robust backup and disaster recovery mechanisms. Regularly backing up critical data, verifying backup integrity, and employing proper access controls can help in minimizing the impact of data destruction attacks.

RESOURCE HIJACKING:

Compromised resources within a Kubernetes cluster can be abused by attackers for malicious activities such as digital currency mining. Attackers who gain access to containers or have the permissions to create new containers may exploit these resources for unauthorized tasks. Implementing strict pod security policies, monitoring resource utilization, and regularly auditing containers for unauthorized activities can help detect and prevent resource hijacking attempts.

DENIAL OF SERVICE (DOS):

Attackers may launch DoS attacks to disrupt the availability of Kubernetes services. This can involve targeting containers, nodes, or the API server. To mitigate the impact of DoS attacks, it is crucial to implement network-level security measures such as ingress and egress filtering, rate limiting, and traffic monitoring. Additionally, implementing resource quotas, configuring horizontal pod autoscaling, and monitoring resource utilization can help in maintaining service availability and mitigating the impact of DoS attacks.

References

- https://devsecopsguides.com/docs/attacks/cloud/
- https://github.com/center-for-threat-informed-defense/mappings-explorer/
- https://center-for-threat-informed-defense.github.io/security-stack-mappings/Azure/README.html
- https://medium.com/mitre-engenuity/security-control-mappings-a-starting-point-for-threat-informed-defense-a3aab55b1625
- https://github.com/0xJs/CARTP-cheatsheet/

Read articles from **DevSecOpsGuides** directly inside your inbox. Subscribe to the newsletter, and don't miss out.

Enter your email address SUBSCRIBE

Devops Kubernetes Docker containers DevSecOps

Written by

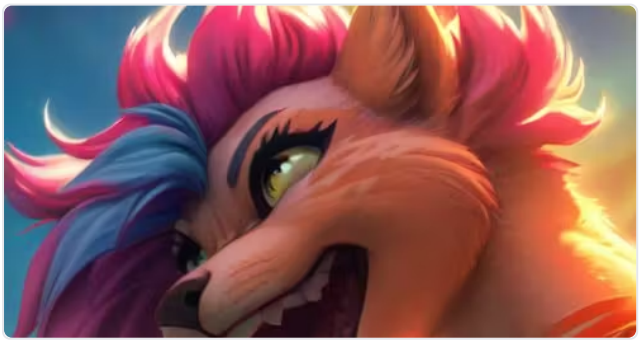
RR **Reza Rashidi** Follow

Published on

DevSecOpsGuides Follow

MORE ARTICLES

RR **Reza Rashidi**



Attacking Azure

Microsoft Azure, a leading cloud computing platform, offers a myriad of services and features to fac...

RR **Reza Rashidi**

Attacking Supply Chain

In today's interconnected and rapidly evolving technological landscape, DevOps practices have revolu...

RR **Reza Rashidi**

Attacking Docker

Docker has revolutionized the way software is developed, deployed, and managed by providing a lightw...