

# Mitigating RBAC-Based Privilege Escalation in Popular Kubernetes Platforms

## Executive Summary

Prisma Cloud and Unit 42 recently released a [report examining the use of powerful credentials](#) in popular Kubernetes platforms, which found most platforms install privileged infrastructure components that could be abused for privilege escalation. We're happy to share that, as of today, all platforms mentioned in our report have addressed built-in node-to-admin privilege escalation. However, it's possible third party add-ons might reintroduce the issue.

In the research we presented at [KubeCon EU and BlackHat USA](#), we found that in half the platforms, any container escape had previously allowed for a full cluster compromise because all nodes hosted admin-equivalent credentials. Most of the platforms mentioned in our report made their infrastructure unprivileged by default, while one did so through an optional add-on.

Stripping permissions is often complex, and we recognize fixing this is no small matter. We'd like to thank Azure Kubernetes Service (AKS), AWS Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), RedHat OpenShift Container Platform, Antrea and Calico for working to harden their access control.

We provide a short recap of our research and look into the different mitigations the platforms implemented to address privilege escalation and powerful permissions in Kubernetes. If you're interested in evaluating your own cluster's Role Based Access Control (RBAC) posture, try [rbac-police](#), our open-source RBAC analyzer for Kubernetes.

[Prisma Cloud](#) users can catch Kubernetes misconfigurations like excessive RBAC permissions before they're deployed to the cluster via the [Cloud Code Security](#) (CCS) module. In the runtime phase, users can rely on the built-in [admission controller for Kubernetes](#) to enforce policies that alert on suspicious activity in their clusters, including Kubernetes privilege escalation.

|    |                             |
|----|-----------------------------|
| R  | <a href="#">Privileged</a>  |
| el | <a href="#">Escalation,</a> |
| a  | <a href="#">Cloud</a>       |
| t  | <a href="#">Security,</a>   |
| e  | <a href="#">Kubernetes</a>  |
| d  | <a href="#">Networks,</a>   |
| U  | <a href="#">Container</a>   |
| n  | <a href="#">Security,</a>   |
| it | <a href="#">Container</a>   |
| 4  | <a href="#">Escalation</a>  |
| 2  | <a href="#">Escalation</a>  |
| T  | <a href="#">Container</a>   |
| o  | <a href="#">Escalation</a>  |
| p  | <a href="#">Container</a>   |
| ic | <a href="#">Escalation</a>  |
| s  | <a href="#">Escalation</a>  |

## Recap: Powerful Permissions Everywhere

Kubernetes managed services, distributions and add-ons install a set of system pods into our cluster to manage its infrastructure and enable core functions such as networking, DNS and logging. Commonly, these pods are deployed via [DaemonSets](#) that distribute them onto every node in the cluster.

If those DaemonSets' permissions are loosely granted, they could inadvertently spread powerful credentials throughout the cluster. This could be abused for privilege escalation, as shown in Figure 1.

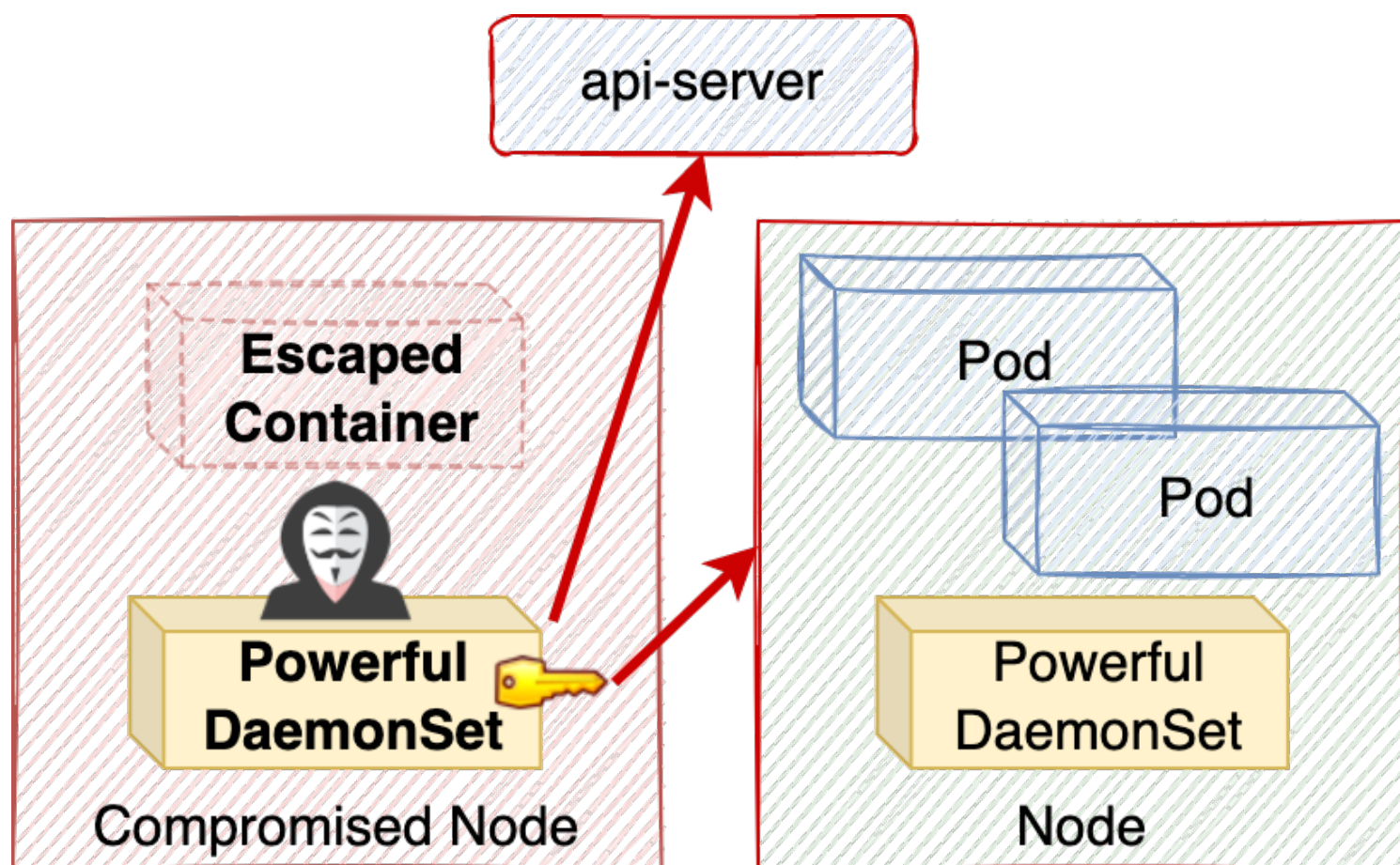


Figure 1. An attacker who escaped a container exploits the credentials of a powerful DaemonSet to spread in the cluster.

To understand the prevalence of powerful DaemonSets, we analyzed popular Kubernetes platforms—managed services, distributions and container network interfaces (CNIs)—to identify privileged infrastructure components. We found that **most platforms ran powerful DaemonSets**, installing privileged credentials onto every node in the cluster.

As shown in Figure 2, in half the platforms, those credentials were admin-equivalent, allowing a single container escape to compromise the entire cluster.

## Container Escape == Cluster Admin?

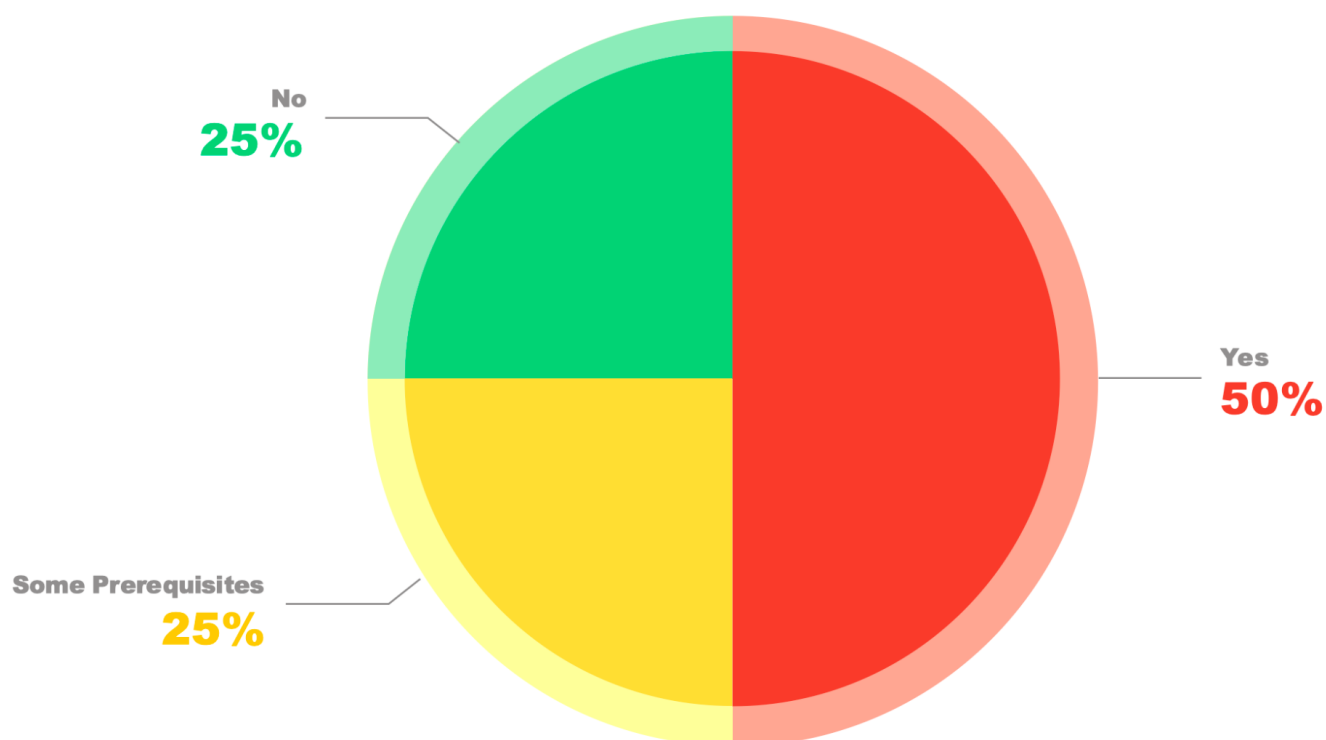


Figure 2. Percentage of platforms where a container escape allowed a complete cluster takeover.

We believe powerful DaemonSets became common for three main reasons:

1. **Historically, Kubernetes clusters weren't secured by default.**

In environments where crucial components like Kubelets allowed unauthenticated access, maintaining a least-privileged RBAC posture wasn't a priority. The infrastructure built then set a precedent for powerful DaemonSets.

2. **Some Kubernetes permissions are simply too broad.**

This means they authorize a large set of operations. Often, granting a service account the ability to perform a necessary but sensitive operation implicitly authorizes it to perform other, potentially harmful operations. RBAC is not a great model for many of these use cases, and an attribute-based access control model that matches some attribute of the principal to some attribute of the resource would often make more sense.

3. **Certain permissions appear benign, but are in fact quite powerful.**

If someone believes a permission is harmless, they won't have second thoughts about granting it. For example, the ability to update the status of pods implicitly allows deleting pods that are part of ReplicaSets.

## Mitigation

After identifying a powerful DaemonSet, we reached out to the relevant platform and started a discussion on mitigation. The response was extremely positive – the teams understood the issue and wanted to resolve it. Mitigations were developed, tested and deployed in recent months. And as of today, all of the privilege escalation attacks we identified are resolved.

Thanks to the work done by the different platforms, the Kubernetes landscape is a safer one, where nodes aren't admins by default. In the following sections, we'll highlight the different approaches platforms took to address powerful DaemonSets in their offerings.

## Strip Permissions

The simplest way to address a risky permission is to remove it. A number of platforms identified certain risky permissions that weren't explicitly necessary, and they removed them. Some permissions were made safe by scoping them down to certain [resourceNames](#) or subresources.

For example, Cilium found that the “delete pods” permission wasn't explicitly necessary for their DaemonSet to operate correctly, and they [removed it](#). Openshift was able to [strip the "update nodes" permission](#) from their software-defined network (sdn) DaemonSet, because the function that relied on it could be replaced with an unprivileged implementation.

## Restrict via Admission Control

A common use case for DaemonSets is local node management, where each DaemonSet pod manages its local node. Unfortunately, Kubernetes doesn't support scoping down a DaemonSet pod's permissions to its local node. Thus every pod in the DaemonSet must be authorized to manage all nodes, not just its local one. This allows attackers who take over a node to abuse the credentials of the local DaemonSet pod to compromise other nodes and spread in the cluster.

Both AKS and EKS ran a DaemonSet that had pods that needed to update their local node. Unfortunately, the only way built in to Kubernetes to grant this permission was to permit every pod in the DaemonSet to update every node in the cluster.

An attacker who compromised a node could abuse the DaemonSet's token to [taint](#) other nodes, allowing it to steal pods. This attack is carried out in three steps, illustrated in Figure 3.

1. Add a NoSchedule taint to every node in the cluster beside the compromised one, making the compromised node the only available node in the cluster.
2. Add a NoExecute taint to the node hosting the target pod to force Kubernetes to evict it and delete its pods.
3. Kubernetes recreates the target pod, and can now only schedule it on the one available node in the cluster, which is the compromised node.

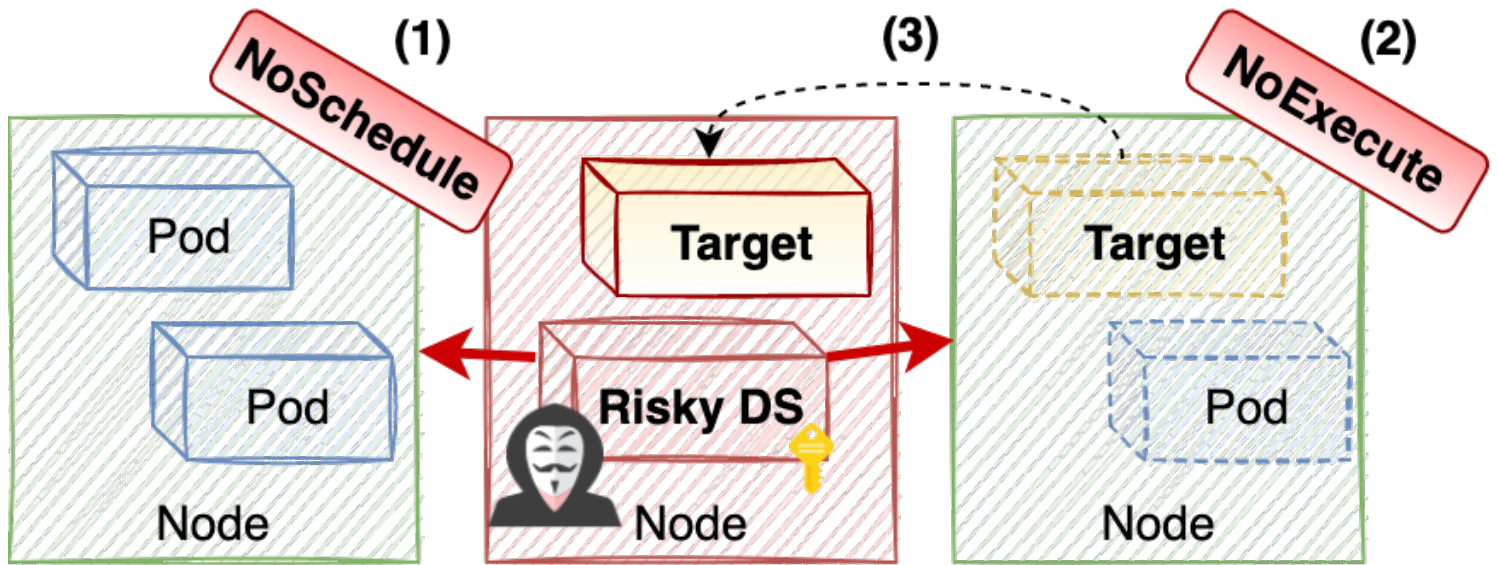


Figure 3. Abusing a DaemonSet's “update nodes” permission to manipulate taints in order to steal pods.

The DaemonSets on EKS and AKS needed the “update nodes” permission, so simply removing it wasn't a valid solution. Instead, both platforms were able to mitigate the attack using custom [validating admission webhooks](#).

## EKS - Restrict by Operation

In EKS, the aws-node DaemonSet needed to update certain node attributes, but not taints. So if an aws-node pod unexpectedly attempts to taint a node, it's likely an attack. The EKS team implemented a validating admission webhook that intercepts node update requests, and blocks attempts to manipulate taints that are issued by the aws-node service account. You can see this in action in Figure 4, below.



```
[root@ip-172-31-14-106 ~]$ kubectl \
> --insecure-skip-tls-verify --server ${APISERVER} \
> --token ${LOCAL_AWS_NODE_TOKEN} \
> taint node ip-172-31-33-254.ec2.internal EvilTaint=value:NoExecute
Error from server (aws-node can only update limited fields on the Node Object): admission
webhook "vnode.vpc.k8s.aws" denied the request: aws-node can only update limited fields on
the Node Object
```

Figure 4. EKS prevents the aws-node pod from tainting nodes.

## AKS - Restrict by Target

AKS had a slightly more difficult situation to contend with. Their cloud-node-manager DaemonSet does taint nodes, so blocking taints altogether wasn't a valid option.

To be exact, each cloud-node-manager only needed to taint its hosting node. AKS used that to mitigate the attack. They wrote a validating admission webhook that intercepts node update requests, and identifies which cloud-node-manager pod issued the request.

If a cloud-node-manager pod attempts to update a node that isn't its hosting one, the request is denied. As shown in Figure 5, if an attacker attempts to abuse a cloud-node-manager pod's credentials, they are denied.

```
root@aks-agentpool-30252850-vmss000000:~# kubectl \
> --insecure-skip-tls-verify --server ${APISERVER} \
> --token ${LOCAL_CLOUD_NODE_MANAGER_TOKEN} \
> taint node aks-agentpool-30252850-vmss000001 EvilTaint=value:NoExecute
Error from server: admission webhook "aks-node-validating-webhook.azmk8s.io" denied the request: (UID: 99193586-cfef-411a-88e0-dfc6c5ecf995) pod cloud-node-manager-h8thb is not allowed to update node "aks-agentpool-30252850-vmss000001": Error "Pod cloud-node-manager-h8thb using service account system:serviceaccount:kube-system:cloud-node-manager running on node aks-agentpool-30252850-vmss000000 is not allowed to taint node aks-agentpool-30252850-vmss000001"
```

Figure 5. AKS prevents the cloud-node-manager pod from tainting other nodes.

To function correctly, this admission webhook must know which pod issued the request, not only which service account. This is possible thanks to a recent Kubernetes enhancement, [bound service account tokens](#).

In the past, a token only referenced the service account it represents. Now, a pod's service account token also includes a claim that specifies the pod the token was issued for. Admission controllers can use that claim to identify which pod a request originated from.

This webhook nicely demonstrates how bound tokens enable node-local authorization for DaemonSet pods. It's possible that in the future, Kubernetes would have built-in support for that functionality, without the need for a custom admission webhook.

## Move Privileged Functionality Elsewhere

To carry out privileged tasks, someone in the cluster needs to possess powerful credentials. Because DaemonSets' credentials are widely distributed, they're not the best fit for carrying out privileged tasks. Several platforms addressed powerful DaemonSets by having control plane controllers or non-DaemonSet pods carry out privileged tasks instead. DaemonSets' powerful permission could then be removed.

For example, in GKE clusters running Dataplane v2 the anted DaemonSet was authorized to update pods and nodes for certain tasks. These permissions are risky, and could also be abused for a number of privilege escalation attacks. By having the control plane take care of pod and node updates, GKE was able to remove these powerful permissions from the anted DaemonSet.

## Recommendations

Even when the underlying Kubernetes infrastructure is configured to maintain appropriate privilege boundaries, add-ons and applications that are misconfigured with excessive permissions can still re-introduce the same attack paths to a cluster. Below are best practices for hardening your cluster's RBAC posture and preventing intracuster privilege escalation.

- Follow the principle of least-privilege. Only grant explicitly required permissions. When possible, scope down permissions to specific namespaces, resources names, or subresources.
- Add guardrails to your Kubernetes Infrastructure-as-Code (IaC) continuous integration/continuous delivery (CI/CD) pipelines. This will prevent developers from unintentionally granting powerful permissions to service accounts. [Checkov](#), an open-source IaC scanner, [supports a number of Kubernetes checks](#) that alert



on excessive RBAC permissions, as shown below in Figure 6.

- Routinely review your RBAC posture. This allows you to identify potential threats and overly powerful identities. Ensure powerful permissions aren't granted to less trusted or publicly exposed pods. Consider using automated tools like [rbac-police](#).
- Refrain from assigning tasks requiring privileged credentials to DaemonSets. Control plane controllers or deployments are preferable in this instance.
- Separate resources and workloads requiring different trust levels into different namespaces.
- Use [admission control](#) to implement fine grained authorization for permissions that cannot be expressed in RBAC. When possible, implement a safe operation allowlist to block unexpected and malicious requests from powerful service accounts.
- Isolate powerful pods on separate nodes from untrusted or publicly-exposed ones using scheduling constraints like [Taints and Tolerations](#), [NodeAffinity](#) rules, or [PodAntiAffinity](#) rules.

```
yavrahami@M-C02YT7FTLVDQ:~$ checkov --quiet -d templates/
kubernetes scan results:

Passed checks: 4, Failed checks: 1, Skipped checks: 0

Check: CKV_K8S_157: "Minimize Roles and ClusterRoles that grant permissions to bind RoleBindings or ClusterRoleBindings"
      FAILED for resource: ClusterRole.default.powerful-cluster-role
      File: /templates/clusterrole-failed-1.yaml:1-8

1 | kind: ClusterRole
2 | apiVersion: rbac.authorization.k8s.io/v1
3 | metadata:
4 |   name: powerful-cluster-role
5 | rules:
6 | - apiGroups: ["rbac.authorization.k8s.io", ""]
7 |   resources: ["clusterrolebindings"]
8 |   verbs: ["bind", "create"]
```

Figure 6: Checkov alerts on a ClusterRole configured with powerful permissions.

Your cluster's threat model should be taken into account when implementing industry best practices. Kubernetes is a platform for building platforms, meaning a cluster's threat model varies heavily depending on its architecture. The above guidelines tackle intracluster privilege escalation.

If you're building clusters with different trust levels, intracluster privilege is a major threat. Below are a few examples for clusters hosting different trust levels:

- Multitenant clusters hosting possibly malicious tenants/workloads
- Running different teams on a single cluster
- Deploying multiple applications across a large cluster

On the other hand, if you're running small clusters that are each dedicated to a single application, then it makes more sense to invest in segregating those clusters from one another and from external services. When the cluster doesn't host different trust levels, preventing privilege escalation inside it shouldn't be a top priority. That being said, **detecting** privilege escalation attacks can still help identify breaches.

## Conclusion

Maintaining a secure RBAC posture in Kubernetes is complex, both for platforms and users. When leading platforms enforce hardened defaults, it's easier for Kubernetes users to adopt secure architectures.

We'd like to thank each of the platforms mentioned in this blog for taking the difficult problem of unprivileged infrastructure head on, and coming up with creative implementations to solve it.

We recommend Kubernetes users and platforms automate the detection of RBAC misconfigurations in their CI/CD pipelines through tools like [Checkov](#) and [rbac-police](#). For managed services, even when an infrastructure component requires risky permissions, a validating admission webhook might be able to prevent misuse.

See our report, [Kubernetes Privilege Escalation: Excessive Permissions in Popular Platforms](#), for more information. [Prisma Cloud](#) users are encouraged to read the report's "Prisma Cloud" section to learn how Prisma's [Kubernetes IaC scanning capabilities](#) and the built-in [admission controller for Kubernetes](#) can tackle the challenges of securing Kubernetes identities.

## Additional Resources

- [Kubernetes Privilege Escalation: Excessive Permissions in Popular Platforms](#)
- [Trampoline Pods: Node to Admin PrivEsc Built Into Popular K8s Platforms](#)
- [Privileged pod escalations in Kubernetes and GKE](#)
- [Checkov](#)
- [rbac-police](#)