Kubernetes Privilege Escalation: Excessive Permissions in Popular Platforms

Table of Contents

Foreword

Kubernetes adoption has skyrocketed in recent years, with more users deploying, testing, and contributing to the project. Weak defaults are a typical growing pain for emerging and complex platforms, and Kubernetes has been no exception. Today though, most Kubernetes® platforms have rooted out insecure defaults, and previously widespread misconfigurations like Kubelets that allow unauthorized access are becoming less and less common. Threat actors who were used to compromising clusters through blatantly simple attacks are probably not very pleased with new improvements, but it seems like the pragmatic ones are starting to evolve and target subtler issues.

Unit 42 recently witnessed that trend in the wild as they caught a sample of Siloscape—one of the most sophisticated Kubernetes malware samples to date. Siloscape chained together multiple exploits to compromise pods, escape and take over nodes, and ultimately gain control over entire clusters. Siloscape demonstrated an approach that wasn't previously seen in the wild: after compromising a node, it checked whether it had excessive permissions and didn't bother continuing the attack if it didn't.

As simpler Kubernetes attacks lose relevance, adversaries have begun targeting excessive permissions and Role-Based Access Control (RBAC) misconfigurations.

Kubernetes RBAC holds the potential to enforce least-privileged access and demoralize attackers, but misconfigurations are easy to miss. Seemingly restricted permissions are often surprisingly powerful, making basic questions like "Which pods can escalate privileges?" difficult to answer. In this report, we aim to address that problem. We introduce a framework that classifies powerful permissions by the attacks they enable; map dozens of the most powerful Kubernetes permissions to it; and release rbac-police, an open source tool that can identify powerful permissions and privilege escalation paths in Kubernetes clusters.

Kubernetes Role-Based Access Control (RBAC) is the main authorization scheme in Kubernetes, and governs the permissions of users, groups, pods, and nodes over Kubernetes resources.

To understand the prevalence and impact of powerful permissions, we've analyzed popular Kubernetes platforms—managed services, distributions, and container network interfaces (CNIs)—and looked for infrastructure components running with excessive permissions. **In 62.5% of the Kubernetes platforms reviewed, powerful DaemonSets distributed powerful credentials across every node in the cluster. As a result, in 50% of platforms, a single container escape was enough to compromise the entire cluster**.

DaemonSets are commonly used to deploy infrastructure pods onto all worker nodes.

We partnered with affected platforms to address these findings and strip excessive permissions. From the original 62.5% that ran powerful DaemonSets, only 25% remain. Likewise, the percentage of platforms where container escape was guaranteed to result in cluster takeover dropped from 50% to just 25%, with more soon to follow. While this moves the needle in the right direction, RBAC misconfigurations and excessive permissions are likely to remain a significant Kubernetes security risk for the near future.

Read on to gain a better understanding of RBAC risks and how you can address them through open source tools and best practice configurations. Learn to transform RBAC from a blind spot into an additional layer of defense.

Executive Summary

Kubernetes platforms have made significant strides in security in recent years, rooting out critical misconfiguration and establishing secure baselines. With fewer clusters vulnerable to straightforward attacks, threat actors are starting to adapt and look for techniques abusing subtler issues. Recent malware samples indicate Kubernetes threat actors are beginning to target excessive permissions.

Kubernetes Role-Based Access Control (RBAC) is an authorization scheme that governs the permissions of users, groups, service accounts, and pods over Kubernetes resources. When used correctly, RBAC can enforce least-privileged access and demoralize attackers. When misconfigured, excessive permissions expose the cluster to privilege escalation attacks and increase the blast radius of compromised credentials and container escape.

RBAC Misconfigurations are Easy to Miss

Seemingly restricted permissions can be surprisingly powerful and, in some cases, on par with cluster admin. As a result, open source add-ons and infrastructure components inadvertently ask for powerful permissions, and users grant them without realizing the full impact on their cluster's security.

Prisma® Cloud researchers identified dozens of powerful Kubernetes permissions, known and novel, and classified them based on the attacks they enable into five major Kubernetes attack types.

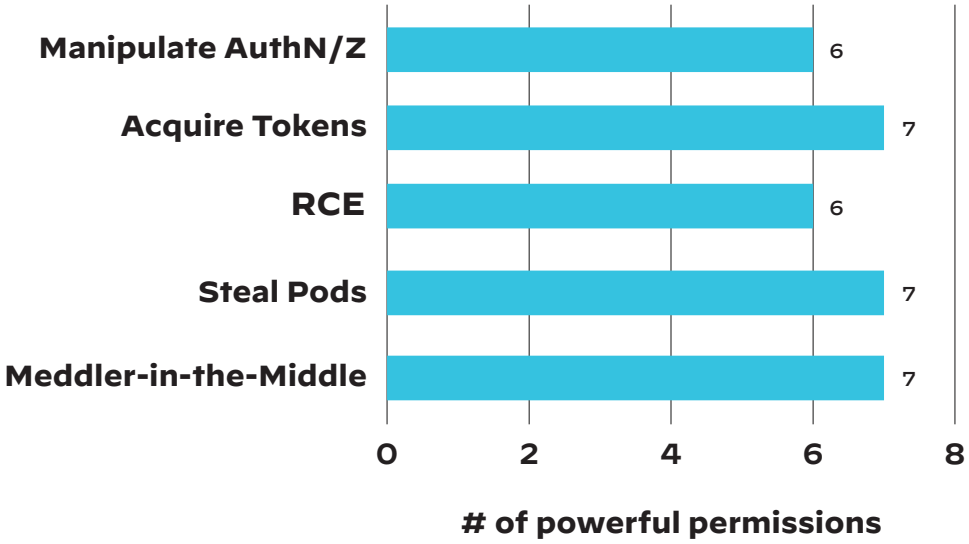

Figure 1: Powerful Kubernetes permissions by attack class

Powerful Permissions are Widespread

To understand the prevalence of powerful permissions, Prisma Cloud researchers analyzed popular Kubernetes platforms—managed services, distributions, and container network interfaces (CNIs)—to identify powerful DaemonSets that distribute powerful credentials across every node in the cluster.

Out of the Kubernetes distributions and managed services examined, 75% ran powerful DaemonSets by default. The remaining 25% did so as well given a recommended feature was enabled. **Examining mainstream Container Network Interfaces (CNIs), 50% installed powerful DaemonSets by default**.

Excessive Permissions Lead to Impactful Attacks

When powerful permissions are loosely granted, they're more likely to fall into the wrong hands. In Kubernetes, that could occur in a number of ways, but it's most easily visible with powerful DaemonSets and container escapes.

The blast radius of container escape drastically increases when powerful tokens are distributed across every node by powerful DaemonSets. **Based on the identified DaemonSets, in 50% of the Kubernetes platforms reviewed, a single container escape was enough to compromise the entire cluster**.

In 12.5% of platforms, a single container escape was likely enough to take over some clusters. For another 12.5%, container escape was enough to compromise the entire cluster given a recommended feature was enabled.

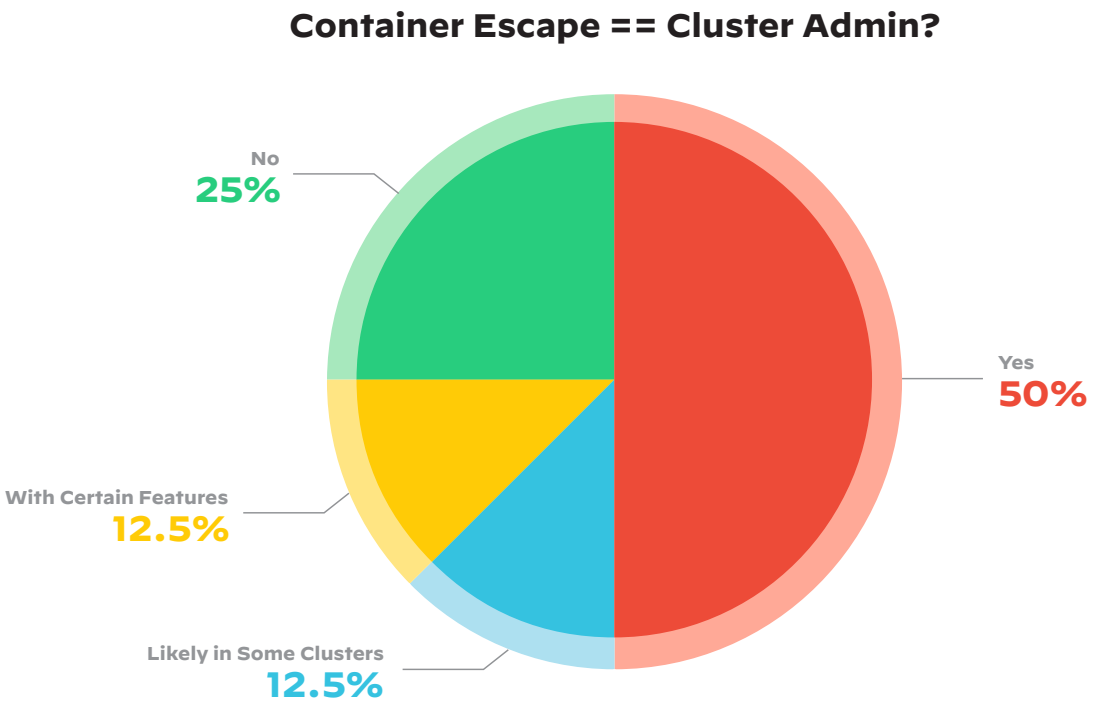

Figure 2: Impact of container escape in the analyzed Kubernetes platforms

RBAC Misconfigurations are Solvable

Prisma Cloud researchers worked with vendors and open source projects to strip excessive permissions and reduce the distribution of powerful credentials. From the original 62.5% running powerful DaemonSets, only 25% remain. Likewise, the number of platforms where container escape is guaranteed to result in cluster takeover dropped from 50% to just 25%. This demonstrates that RBAC misconfigurations are solvable and that powerful permissions can often be removed. It also highlights the commitment of the reviewed vendors and open source projects to the security of their platforms.

To help Kubernetes users evaluate and improve the RBAC posture of their clusters, this report is released alongside rbac-police, **a new open source tool that can identify powerful permissions and privilege escalation paths in Kubernetes clusters**. New RBAC checks were also contributed to Checkov, a leading open source infrastructure as code (IaC) scanner.

Finally, the Recommendations section explores a number of best practices that decrease the distribution of powerful credentials and limit the blast radius of compromised ones, along with admission policies that can detect and prevent privilege escalation attacks in real time.

Role-Based Access Control 101

Kubernetes RBAC is an authorization scheme that governs access to Kubernetes resources. Permissions are grouped into Roles or ClusterRoles, and can be granted via RoleBindings or ClusterRoleBindings to users, groups, and service accounts. Permissions granted via RoleBindings are scoped to a namespace, while ones granted via ClusterRoleBindings are in effect cluster-wide.

The ClusterRoleBinding that follows, for example, grants the 'pod-reader' ClusterRole to the 'reader-sa' service account.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
 name: grant-pod-reader-to-reader-sa
roleRef:
 apiGroup: rbac.authorization.k8s.io
 kind: Clusterole
 name: pod-reader
subjects:
- kind: ServiceAccount
  name: reader-sa
  namespace: default
```

The 'reader-sa' service account is now authorized to perform the operations listed in the 'pod-reader' ClusterRole.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Clusterole
metadata:
 name: pod-reader
rules:
- apiGroups:
 - ""
 resources:
 - pods
 verbs:
 - list
 - get
```

As seen above, Kubernetes permissions are expressed by rules. Each rule permits one or more verbs over one or more resources in one or more API groups. The rule above permits listing and getting pods in the core API group. Common verbs include:

- **get**: retrieve a resource by name
- **list**: retrieve all resources
- **create**: create a resource
- **update**: replace an existing resource
- **patch**: modify an existing resource
- **delete**: delete a resource

Roles and ClusterRoles (i.e., permissions) can be granted to a pod by binding them to its service account, as illustrated in figure 3. A pod assigned the 'reader-sa' service account, for example, will be able to retrieve pods cluster-wide.

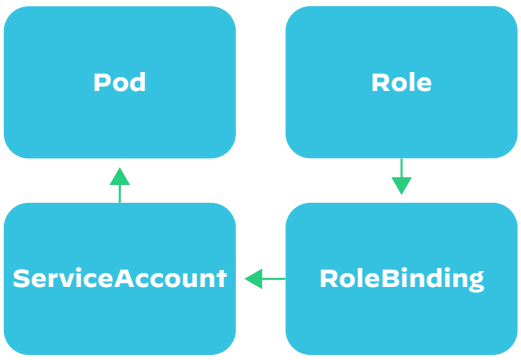

Figure 3: A Role granted to a pod

Classifying Powerful Kubernetes Permissions

Attackers may abuse certain Kubernetes permissions to escalate privileges, move laterally or obtain broader control over a cluster. From here on, those will be referred to as 'powerful permissions.'

Some powerful permissions are near-equivalent to cluster admin, while others can only be abused in specific scenarios for limited attacks. To establish a common framework when discussing powerful permissions, we classified them based on the attacks they enable into five attack types.

Table 1: Powerful Kubernetes Permissions by Attack Class

Manipulate AuthN/Z	Acquire Tokens	RCE	Steal Pods	MitM
impersonate	list secrets	create pods/exec	modify nodes	control endpointslices
escalate	create secrets	update pods/ephemeral-containers	modify nodes/status	modify endpoints
bind	create serviceaccounts/token	create nodes/proxy	create pods/eviction	modify services/status
approve signers	create pods	control pods	delete pods	modify services/status
update certificatesigning-requests/approval	control pod controllers	control pod controllers	delete nodes	modify pods
control mutating webhooks	control validating webhooks	control mutating web-hooks	modify pods/status	create services
—	control mutating webhooks	—	modify pods	control mutating webhooks

Scope is key when it comes to powerful permissions. A permission can be admin-equivalent when granted over the entire cluster, but harmless when scoped to a namespace or to specific resource names. In order to include all possible powerful permissions, the table above assumes permissions are granted cluster-wide.

Certain powerful permissions enable a number of attacks and are thus mapped to multiple attack classes. On the other hand, some of the more complicated attacks require a combination of their listed permissions to carry out. Permissions that aren't powerful enough to carry the attack on their own are marked in yellow.

To avoid disproportionate inflation, Table 1 aggregates similar verbs and resources. The update and patch verbs were aggregated to a virtual "modify" verb, while modify and create were combined to "control". DaemonSets, Deployments, CronJobs and other pod controllers were counted as "pod controllers". Therefore, write privileges over pod controllers are represented as one virtual "control pod controllers" permission rather than the actual 21 related permissions (e.g., create Deployments, update Deployments, patch Deployments, create CronJobs, etc.).

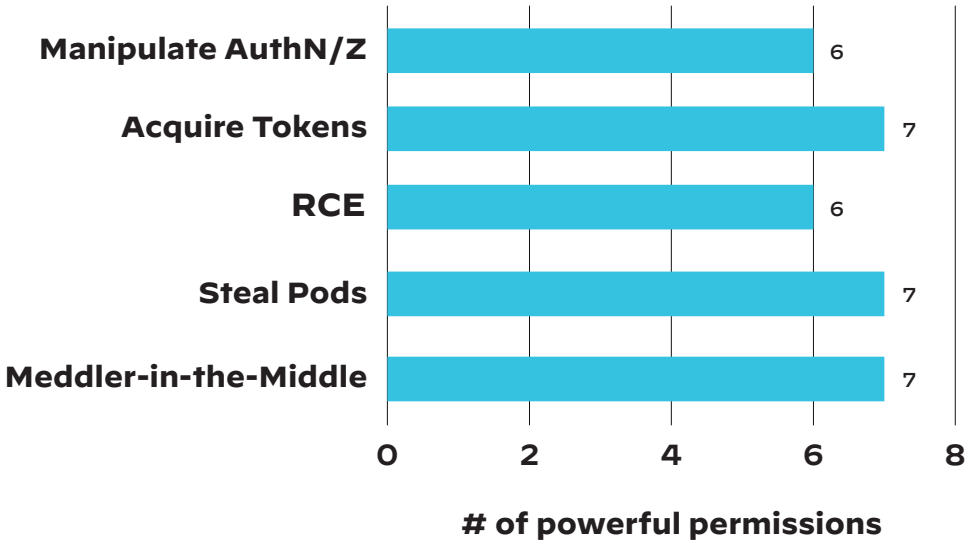

Figure 4: Powerful RBAC permissions by attack class

It's unlikely that Table 1 contains every powerful permission in Kubernetes, but it's the most complete list we're aware of. It's also worth noting that there are other "weaker" attack classes that we haven't looked into, such as Denial-of-Service (DoS).

Below is a breakdown of each attack class.

Acquire Tokens

This group contains permissions that allow, either directly or indirectly, to retrieve or issue service account tokens. The main factor that dictates the impact of these permissions is their scope—whether or not they're granted over a privileged namespace that hosts powerful service accounts. The only namespace that's privileged by default is kube-system, but some platforms may install additional privileged namespaces.

Permissions include: create pods, create secrets, list secrets, update Deployments, create serviceaccounts/token

Attack Example

An attacker armed with the create serviceaccounts/token permission in the kube-system namespace can issue new tokens for pre-installed powerful service accounts through TokenRequests.

Remote Code Execution

Permissions in this group allow executing code on pods, and possibly on nodes. Attackers won't necessarily escalate privileges by abusing these permissions—it depends on the permissions of the attacked pod or node. Still, these permissions increase the compute resources and possibly the business logic that is under the attacker's control.

Permissions include: create pods/exec, create nodes/proxy, patch DaemonSets, create pods

Attack Example

An attacker armed with the create pods/exec permission can execute code on other pods, for example via the interface provided by kubectl exec.

Manipulate Authentication/Authorization (AuthN/AuthZ)

Permissions in this group permit manipulation of authentication and authorization. They often enable privilege escalation by design for use cases like granting permissions or impersonating other identities. They're extremely powerful, and users should be extra careful when granting them.

Permissions include: bind clusterrolebidings, impersonate serviceaccounts, escalate roles

Attack Example

An attacker that can bind clusterrolebindings can grant the pre-installed cluster-admin clusterrole to his compromised identity.

Steal Pods

Certain permissions or permission combinations may allow attackers to steal pods from one node to another. For this attack to be impactful, the attacker must first compromise a node where he intends to place the stolen pod. Stealing a pod consists of two steps: evicting a pod, and then ensuring it lands on your node. To maximize impact, attackers would target pods with powerful service account tokens.

A similar attack—affecting the scheduling of future pods—isn't covered as part of this report.

Permissions include: update nodes, create pods/eviction, delete pods, update nodes/status

Attack Example

An attacker that compromised a node and has the update nodes permission can steal pods from other nodes onto his compromised node. By adding a taint with the NoExecute effect to the target node, the attacker can force Kubernetes to evict and reschedule the target node's pods. By adding a taint with the NoSchedule effect to all other nodes, the attacker can ensure the evicted pods are rescheduled onto his compromised node.

It's worth noting that pods that tolerate NoExecute taints cannot be stolen through this technique. These pods aren't very common, but one popular example would be the admin-equivalent "tigera-operator" pod installed by Calico.

To the best of our knowledge, stealing pods with NoExecute taints is a novel attack technique.

Meddler-in-the-Middle

Permissions in this group may allow attackers to launch meddler (man)-in-the-middle attacks against pods, nodes, or services in the cluster. Exploiting permissions in this group often requires a number of prerequisites for relatively weak impact. Additionally, securing communication with TLS can nullify most MitM attacks.

Permissions include: update services/status, control endpointslices, patch pods/status

Attack Example

An attacker armed with the update services/status permission can exploit CVE-2020-8554 via Load Balancer IPs to redirect traffic sent by pods and nodes from its intended target to an existing endpoint. The attacker must control an existing endpoint for this to be a meaningful attack.

Container Escapes and Powerful DaemonSets: A Toxic Combination

When powerful permissions are loosely granted, they're more likely to fall into the wrong hands. In Kubernetes, that could occur in a number of ways, but it's most easily visible with powerful DaemonSets and container escapes.

The blast radius of container escapes drastically increases when powerful DaemonSets distribute powerful tokens across every node in the cluster. **With powerful DaemonSets installed, attackers that managed to escape a container are guaranteed to hit the jackpot—powerful credentials on their compromised node**.

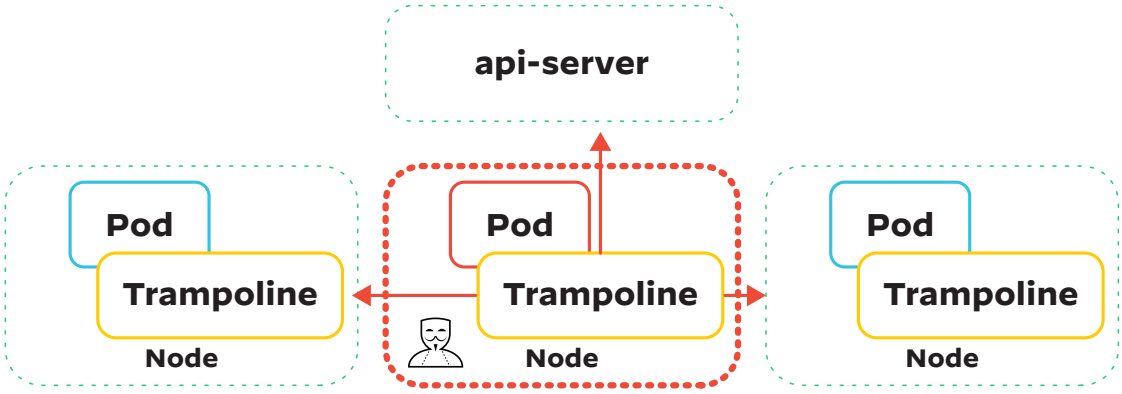

Figure 5: Powerful DaemonSets drastically increase the impact of container escape

We use "Trampoline pods" as a synonym for powerful pods. The name denotes their impact: attackers that manage to compromise a Trampoline pod or its node can abuse its token to jump around the cluster, compromise other nodes and gain higher privileges. Not all Trampolines offer the same bounce. Depending on their permissions, some may allow an attacker to compromise the entire cluster, while others may only be abused in certain scenarios.

It's reasonable to run some powerful pods. Powerful permissions exist for a reason: they're sometimes needed. Powerful pods that don't run as parts of DaemonSets can be isolated from untrusted and publicly exposed ones through several methods (described in "Recommendations"). Even without actively taking measures to isolate them, non-DaemonSet Trampolines are simply less likely to be present on a particular compromised node.

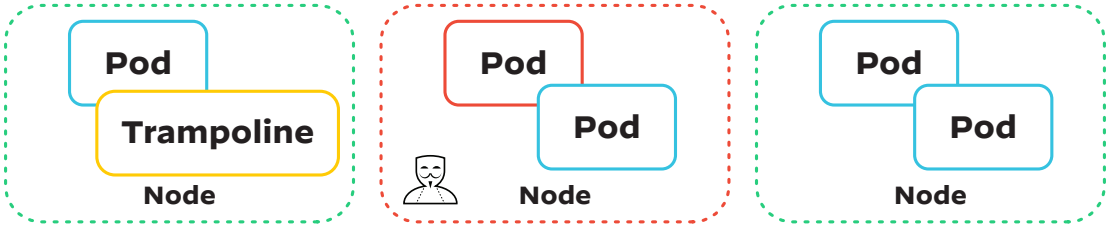

Figure 6: Non-DaemonSet trampolines can be isolated from untrusted pods, either actively or by chance

What primarily makes Trampoline DaemonSets a security concern is **the distribution of powerful credentials**. With powerful DaemonSets, every node in the cluster hosts powerful credentials, meaning attackers that managed to escape a container are guaranteed to find a powerful token on the compromised node.

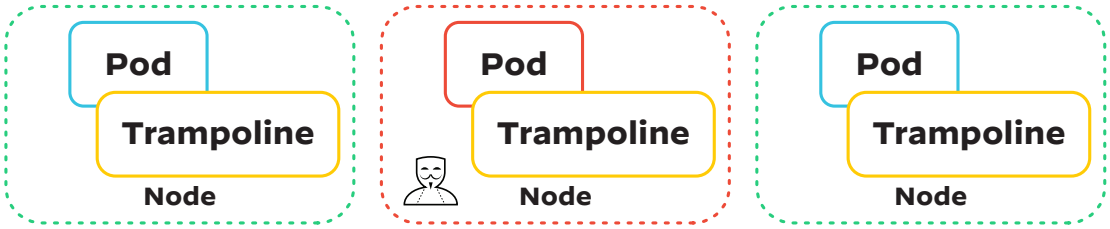

Figure 7: With Powerful DaemonSets, attackers are guaranteed to find powerful credentials on a compromised node

Aren't Nodes Powerful by Default?

Without powerful DaemonSets, the only cluster credentials available on a node belong to the node agent—the Kubelet. In 2017, Kubernetes addressed privilege escalation attacks rooted in the Kubelet permissions by releasing the NodeRestriction admission controller. NodeRestriction limits the permissions of the Kubelet to resources that are already bound to its node, like the pods running on top of it. As a result, nodes cannot escalate privileges or become cluster admins, and thus without Trampoline Pods, a container escape isn't enough to take over the entire cluster.

It's worth noting that NodeRestiction isn't perfect - Kubelets can still read most cluster objects, bypass egress network policies, initiate certain Denial-of-Service (DoS) attacks, and even launch Meddler-in-the-Middle attacks against pod-backed services. While these are all possible, it's important to differentiate from permissions that enable low severity attacks against certain configurations, from ones that can be reliably abused to escalate privileges and compromise clusters.

The next section goes over Trampoline DaemonSets in popular Kubernetes platforms. We didn't consider DaemonSets to be powerful if they only enabled low severity or unreliable attacks, including those that Kubelets can carry out independently. Daemonsets were only considered powerful if their permissions could realistically lead to a full cluster compromise.

Powerful DaemonSets in Popular Kubernetes Platforms

To understand the prevalence and real-world impact of powerful permissions, Prisma Cloud researchers analyzed eight popular Kubernetes platforms and looked for DaemonSets running with powerful permissions.

Table 2: Analyzed Kubernetes Platforms		
Platform	**Type**	**Vendor**
Analyzed Kubernetes Platforms	Managed Service	Microsoft Azure
Elastic Kubernetes Service (EKS)	Managed Service	Amazon Web Services
Google Kubernetes Engine (GKE)	Managed Service	Google Cloud Platform
Openshift Container Platform (OCP)	Distribution	Red Hat
Antrea	CNI	Open Source
Calico	CNI	Open Source
Cilium	CNI	Open Source
Weave Net	CNI	Open Source

Out of the Kubernetes platforms examined, 62.5% installed powerful DaemonSets by default, while another 12.5% did so as well with a recommended feature enabled.

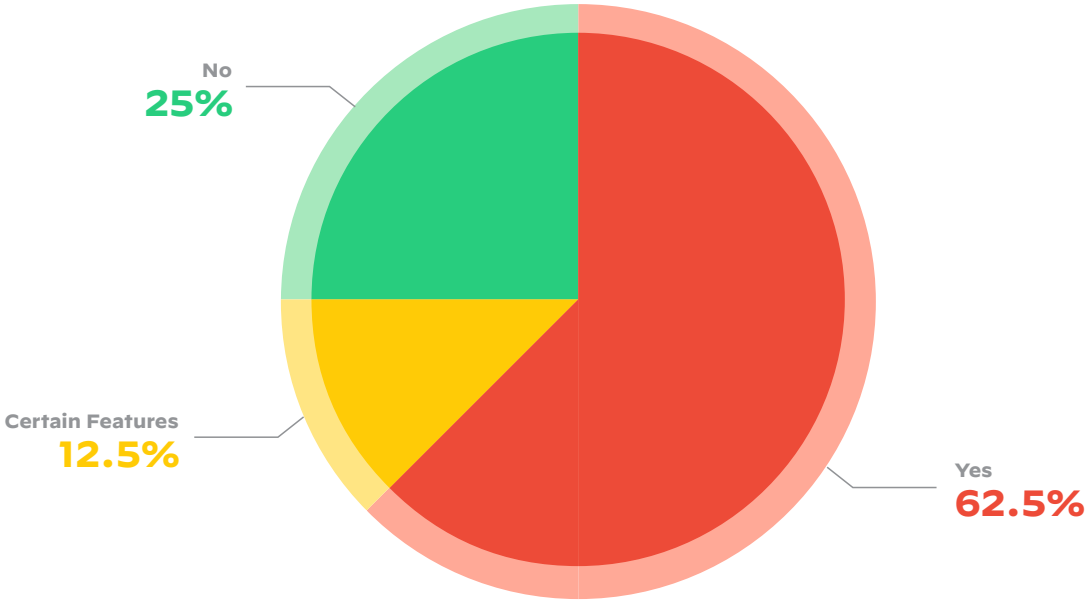

Figure 8: Popular DaemonSets in the analyzed Kubernetes platforms

Table 3: Powerful DaemonSet in the Analyzed Kubernetes Platforms			
Platform	**Powerful DaemonSets**	**DaemonSet**	**Most Powerful Permissions**
AKS	Yes	cloud-node-manager, csi-azurefile-*	list secrets, update nodes
EKS	Yes	aws-node	update nodes
GKE	Only with Dataplane v2	anetd	update nodes, update pods
OCP	Yes	machine-config, sdn, multus-*	create pods, update validatingwebhook-configurations
Antrea	Yes	antrea-agent	patch nodes, patch pods, update services, update services/status
Calico	No	—	—
Cilium	Yes	cilium	update nodes, update pods, delete pods
Weave Net	No	—	—

Container Escape Blast Radius

Based on the identified powerful DaemonSets, in 50% of the Kubernetes platforms reviewed a single container escape was enough to compromise the entire cluster. In another 12.5%, a container escape was likely enough to take over some clusters. For 12.5% of platforms, a container escape was enough to compromise the entire cluster given a recommended feature was enabled.

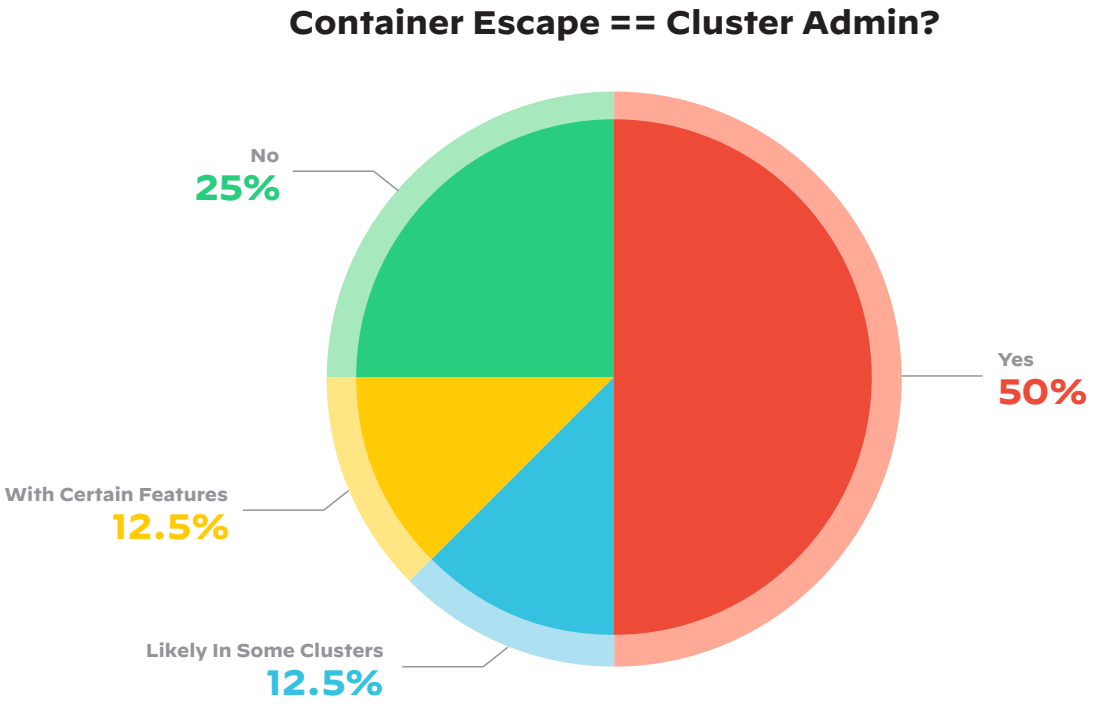

Figure 9: Impact of container escape in the analyzed Kubernetes platforms

In some platforms, DaemonSets possessed admin-equivalent permissions, meaning that abusing them to acquire admin privileges was straightforward. In other platforms, DaemonSets weren't powerful enough to become full admins by themselves, but they did possess permissions that allowed them to take over other pods. In most of these platforms, because admin-equivalent pods were installed by default, attackers could still abuse the platform's DaemonSets to acquire admin privileges.

In Antrea, for example, the antrea-agent DaemonSet wasn't powerful enough to acquire admin privileges by itself, but it did possess powerful permissions allowing it to take over other pods. Because Antrea installs an admin-equivalent pod by default, the antrea-controller, antrea-agent's permissions could still have been exploited to acquire admin privileges by abusing them to compromise the antrea-controller pod.

Table 4: Impact of Container Escape Across Analyzed Platforms

Platform	Escape == Admin	Attack	Prerequisite
AKS	Yes	Acquire Token → Manipulate AuthN/Z	—
EKS	Likely in some clusters	Steal Pods	A stealable powerful pod
GKE	With Dataplane v2	Steal Pod / RCE → Acquire token → Manipulate AuthN/Z	Dataplane v2 enabled
OCP	Yes	Acquire Token	—
Antrea	Yes	Steal Pods / RCE → Acquire token → Manipulate AuthN/Z	—
Calico	No	—	—
Cilium	Yes	Steal Pod / RCE –→ Acquire token → Manipulate AuthN/Z	—
Weave Net	No	—	—

If your clusters rely on one of the impacted platforms, please don't panic. Here's why:

1. To abuse powerful DaemonSets, attackers first need to compromise and then escape a container. Best practices and active defenses can prevent that.
2. Several platforms have already released versions that de-privilege powerful DaemonSets.
3. Best practice hardenings can prevent certain attacks. For example, an allow-list policy for container images can hinder lateral movement attacks that abuse the 'patch pods' permission to replace the image of an existing pod with an attacker-controlled one.
4. That being said, if you run multitenant clusters, you're at greater risk.

A "Likely in Some Clusters" in the "Escape == Admin" column signifies that there's a prerequisite for a container escape to be enough to compromise the entire cluster, but that it's likely to be met in some clusters. For example, an attacker abusing a powerful DaemonSet that can Steal Pods can only acquire cluster admin privileges if there's an admin-equivalent pod to steal in the cluster.

In EKS for example, there isn't such a pod by default. Still, based on the sheer number of popular Kubernetes add-ons that install admin-equivalent pods, it's likely that this prerequisite is met in many clusters in-the-wild. Some popular projects that install admin-equivalent pods by default include ingress-nginx, cert-manager, Kynvero, traefik, and aws-load-balancer.

It's worth noting that with Cilium, there were two popular installation methods. The table above pertains to the one documented as the default—the cilium-cli. While the default Helm installation also deployed the same powerful DaemonSet that can take over other pods, it didn't deploy an admin-equivalent pod that can be targeted by it. Accordingly, when Cilium was installed via Helm, a container escape was only enough to compromise the entire cluster given the user installed an admin-equivalent pod (or, in other words, "Likely in Some Clusters").

Powerful Kubelets in Popular Platforms

While the majority of Kubernetes distributions and managed services have adopted the NodeRestriction admission controller, some still run powerful Kubelets. Powerful Kubelets introduce the same security risks as powerful DaemonSets—compromised nodes can escalate privileges and take over the rest of the cluster. Below is a breakdown of powerful Kubelets across the analyzed managed services and distributions.

Table 5: Powerful Kubelets Across Analyzed Managed Services and Distributions

Platform	Type	Powerful Kubelets
AKS	Managed Service	Yes
EKS	Managed Service	No
GKE	Managed Service	No
OCP	Distribution	No

Fixes and Mitigations by Affected Platforms

We reported the identified powerful DaemonSets and Kubelets to affected vendors and open source projects between December 2021 and February 2022. The vast majority of platforms pledged to strip powerful permissions from their Daemonsts, and some of them have already done so. From the original 62.5%, only 25% still run powerful DaemonSets.

Table 6: Fixes and Mitigations by Affected Platforms

Platform	Had Powerful DaemonSets	Fixed	Had Powerful Kubelets	Fixed
AKS	Yes	No	Yes	WIP
EKS	Yes	Yes, from Kubernetes v1.18	No	—
GKE	With Dataplane v2	Yes, from v1.23.4-gke.900, 13022$ Bounty	No	—
OCP	Yes	WIP set for v4.11, possible backports	No	—
Antrea	Yes	Yes, v1.6.1 alongside an admission policy	No	—
Calico	No	—	No	—
Cilium	Yes	Yes, v1.12.0-rc2, some fixes backported	No	—
Weave Net	No	—	No	—

Platforms addressed powerful DaemonSets through a variety of techniques. Most applied one or more of these three solutions:

1. **Remove**: Certain permissions were deemed unnecessary, or too widely scoped, and were simply removed.
2. **Relocate**: Move the functionality that required the powerful permissions from DaemonSets running on all nodes to deployments that run on few, or to the control plane.
3. **Restrict**: Release admission policies that limit powerful DaemonSets to a number of safe and expected operations.

According to the improvements above, the number of platforms where a single container escape is enough to compromise the entire cluster dropped from 50% to just 25%. Keep in mind that this number pertains to Kubernetes-native attacks and doesn't cover possible platform-specific privilege escalations.

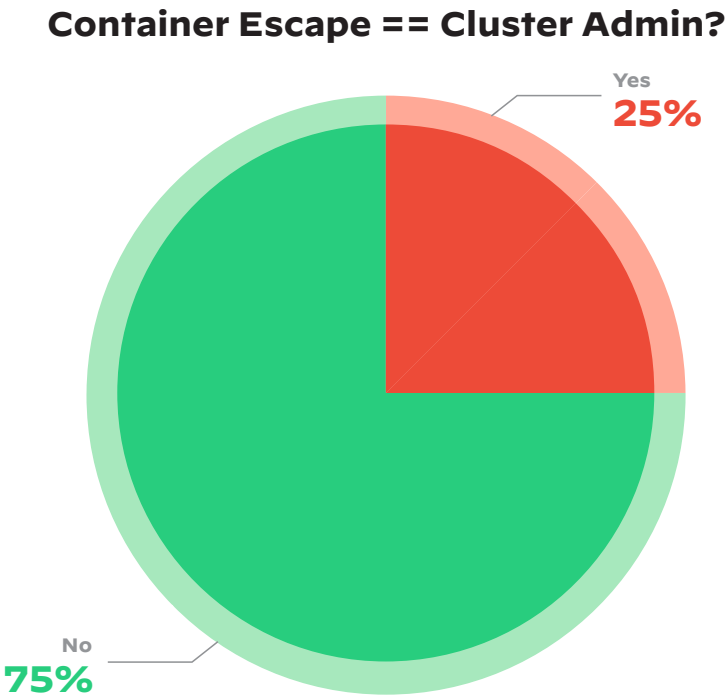

Figure 10: Impact of container escape in the analyzed Kubernetes platforms following fixes

Stripping existing permissions can be challenging. We'd like to thank the vendors and open source projects mentioned in this report for their effort to remove powerful DaemonSets and Kubelets from their platforms.

Toward Better Node Isolation

One step at a time, Kubernetes is moving toward stronger node isolation. This effort started with the NodeRestriction admission controller and is slowly inching forward with every powerful permission removed from a popular DaemonSet. Complete node isolation is unlikely in the near future: some low severity attacks will likely remain, and certain nodes will need to host powerful pods. That being said, better node isolation is certainly possible. At the very least, clusters shouldn't host powerful credentials on every node. Removing Trampoline DaemonSets can ensure the majority of nodes are unprivileged.

Some powerful permissions will be harder to drop, in part due to the lack of fine-grained access control for certain operations. This shouldn't be seen as an "all or nothing" issue though. Even when certain permissions cannot be easily stripped, it's still a welcomed improvement when a DaemonSet that could previously acquire admin tokens is now only able to launch meddler-in-the-middle attacks.

Identifying Powerful Permissions

Whether you use a mentioned platform or not, if you run Kubernetes, your clusters likely host powerful pods. The first step of addressing risky permissions is identifying them. The following tools can be used to identify powerful permissions in running clusters and in Kubernetes manifests.

rbac-police

We're very excited to release rbac-police, a tool we used throughout this research to identify powerful permissions.

An open source command line interface (CLI) written in Golang, rbac-police retrieves the permissions of pods, nodes, and services accounts in a cluster, and evaluates them through built-in or custom Rego policies. Assessing the RBAC posture of your cluster is as easy as running `'rbac-police eval lib'`. The image below shows a slice of rbac-police's output:

```
yavrahami@M-C02YT7FTLVDQ:~/rbac-police$ ./rbac-police eval lib
{
    "policyResults": [
        {
            "policy": "lib/assign_kubesystem_sa.rego",
            "severity": "Critical",
            "description": "SAs and nodes that can create pods or create, update or patch pod controllers (e.g. Daemonsets, Deployments,
Jobs) in the kube-system namespace, may assign an admin-equivalent SA to a pod in their control",
            "violations": {
                "nodes": [
                    "aks-agentpool-32959266-vmss000000",
                    "aks-agentpool-32959266-vmss000001"
                ]
            }
        },
        {
            "policy": "lib/control_webhooks.rego",
            "severity": "High",
            "description": "SAs and nodes that can create, update or patch ValidatingWebhookConfigurations or MutatingWebhookConfiguratio
ns can read, and in the case of the latter also mutate, any object admitted to the cluster",
            "violations": {
                "serviceAccounts": [
                    {
                        "name": "gatekeeper-admin",
                        "namespace": "gatekeeper-system",
                        "nodes": [
                            {
                                "aks-agentpool-32959266-vmss000000": [
                                    "gatekeeper-controller-manager-9fff5d5-6sk6t"
                                ]
                            },
```

Figure 11: rbac-police alerts on excessive permissions of service accounts, pods, and nodes

Out of the box, rbac-police is equipped with more than 20 policies that each hunt for a different set of powerful permissions. It's also 100% customizable though. You can write your own policies that search for any pattern in Kubernetes RBAC—powerful permissions we've missed, permissions that only affect certain platforms, or ones related to CRDs (Custom Resources Definitions). If you end up writing a policy, please consider contributing it.

Supported commands for rbac-police are as follows:

- `rbac-police eval` evaluates the RBAC permissions of services accounts, pods, and nodes through built-in or custom Rego policies.
- `rbac-police collect` retrieves the RBAC permissions of services accounts, pods, and nodes in a cluster. Useful for saving a RBAC snapshot of the cluster for multiple evaluations with different options.
- `rbac-police expand` presents the RBAC permissions of services accounts, pods, and nodes in a slightly more human friendly format. Useful for manual drilldown.

For fine-tuned evaluation, rbac-police provides a variety of options, including:

- `--only-sa-on-all-nodes` to evaluate only service accounts that exist on all nodes. Useful for identifying powerful DaemonSets.
- `--namespace, --ignored-namespaces` to scope evaluation to a single namespace; ignore certain namespaces.
- `--severity-threshold` to evaluate only policies with severity equal or larger than a threshold.

Additionally, rbac-police also supports policies that evaluate the effective permissions of a node—the union of its Kubelet permissions and pods' permissions. Some of the more complex attacks require a number of permissions to execute. Thus, it is possible that while no single pod has all the permissions necessary to carry out an attack, a combination of pods on a node do.

Check out rbac-police's GitHub page for more information. If you run Kubernetes, consider trying it out. It takes seconds to run and provides a lot of valuable insight into your RBAC posture and possible risks.

Checkov

Checkov is an open source static code analysis tool by Bridgecrew for scanning infrastructure as code (IaC) files for misconfigurations that may lead to security or compliance problems. Checkov shifts security left by alerting on misconfigurations before they're committed to production environments.

We've recently contributed four new RBAC checks that alert on Kubernetes IaC files containing Roles or ClusterRoles that define powerful permissions: CKV_K8S_155, CKV_K8S_156, CKV_K8S_157 and CKV_K8S_158. These focus on highly powerful permissions that can be abused to manipulate authentication and authorization, such as impersonation.

Checkov is currently adding support for graph checks that can evaluate connections between multiple Kubernetes resources. Once that feature is released, expect to see more RBAC checks added.

```
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 12: Checkov alerts on a ClusterRole with powerful permissions

Check out Checkov's website for more information.

Recommendations

Tackling powerful RBAC permissions can be complex. They're easy to miss and often asked by third-party add-ons or the underlying infrastructure. Even when you manage the powerful component, dropping permissions isn't always straightforward and often involves code changes.

Whether you run Kubernetes clusters or maintain a popular Kubernetes project, below are best practices and hardening measures that can improve your RBAC posture.

1. Follow the principle of least privilege: only assign explicitly required permissions:
   a. When possible, use RoleBindings to grant permissions over a certain namespace rather than cluster-wide.
   b. Use resourceNames to scope down permissions to specific resources.
2. Track powerful permissions and ensure they're not granted to less-trusted or publicly exposed pods. If you maintain a Kubernetes project, document the powerful permissions asked by your platform.
3. Refrain from running powerful DaemonSets:
   a. Move functionalities that require powerful privileges from DaemonSets running on all nodes to deployments running on few or to control plane controllers.
   b. Rely on the Kubelet credentials for operations that only involve objects bound to the local node, such as retrieving secrets of neighboring pods.
   c. Minimize write permissions by storing state in CRDs and ConfigMaps rather than in core objects like pods.
4. Isolate powerful pods from untrusted or publicly exposed ones using scheduling constraints like Taints and Tolerations, NodeAffinity rules, or PodAntiAfinity rules.

5. Configure policy controllers to alert on automated identities such as service accounts and nodes that query for their permissions via the SelfSubjectReviews APIs. These requests may point to compromised credentials.
6. Configure policy controllers to detect or prevent misuse of powerful permissions for nefarious activities. Abuse of powerful permissions often diverges from normal usage. See the examples below for more details.

Detecting Attacks with Admission Control

Quite often, compromised credentials exhibit irregular behaviors, and present an opportunity for defenders to identify breaches. In Kubernetes, admission control can detect and prevent attacks powered by compromised credentials and privileged permissions. Policy controllers like OPA Gatekeeper and Kynvero can enforce policies that prevent or alert on suspicious requests to the Kubernetes API. Below are two examples for this approach using OPA Gatekeeper.

Suspicious SelfSubjectReviews

A common attacker pattern following credential theft is querying the system for their permissions. In Kubernetes, that is done via the SelfSubjectAccessReview or SelfSubjectRulesReview APIs. Non-human identities like serviceAccounts and nodes querying these APIs for their permissions are strong indicators of compromise. A policy that detects these requests offers a great opportunity to catch compromised credentials.

Here's an example of a policy for OPA Gatekeeper that detects such queries.

Suspicious Assignment of Controller Service Accounts

By default, the kube-system namespace hosts several admin-equivalent service accounts that are used by controllers running as part of the api-server. Attackers that can create pods or pod controllers in the kube-system namespace, or modify pod controllers in kube-system namespace, can assign one of these admin-equivalent service accounts to a pod in their control and abuse their powerful token to gain complete control over the cluster.

In the framework introduced in "Classifying Powerful Kubernetes Permissions," this attack is classified under Acquire Tokens.

Controller service accounts aren't normally assigned to running pods. Defenders can capitalize on that to detect this privilege escalation attack with a policy that alerts on requests that attach a controller service account to an existing or new kube-system pod. We wrote an example for OPA Gatekeeper, which is available here.

Conclusion

As outlined in this report, excessive RBAC permissions are common, easily missed, and can result in impactful privilege escalation attacks against Kubernetes clusters. At the same time, hardened RBAC settings can enforce least privilege, block unintended access, and demoralize attackers.

Maintaining a secure RBAC posture can be challenging due to the dynamic nature of Kubernetes and the number of third-party add-ons commonly used to operate modern clusters. Refer to the "Identifying Powerful Permissions" section for tools like rbac-police that can evaluate your RBAC posture, and see the "Recommendations" section for ways you can minimize risk and hold off attacks even when some powerful pods still exist in a cluster.

We'd like to thank the vendors and open source projects mentioned in this report for their collaboration as well as their efforts to minimize the distribution of powerful credentials in their platforms.

About

Prisma Cloud

Prisma® Cloud is a comprehensive cloud native security platform with the industry's broadest security and compliance coverage—for applications, data, and the entire cloud native technology stack—throughout the development lifecycle and across hybrid and multicloud deployments. Prisma Cloud's integrated approach enables security operations and DevOps teams to stay agile, collaborate effectively, and accelerate cloud native application development and deployment securely.

Prisma Cloud's Cloud Workload Protection (CWP) module delivers flexible protection to secure cloud VMs, containers and Kubernetes apps, serverless functions and containerized offerings like Fargate tasks. Using the built-in admission control for Kubernetes, users can enforce policies that alert on suspicious or non-compliant activities in their cluster, including Kubernetes privilege escalation. Refer to CWP's sample repository for admission policies that can detect the attacks outlined in this report

Prisma Cloud's Cloud Code Security (CCS) module delivers automated security for cloud native infrastructure and applications, integrated with developer tools. The module shifts security left by catching misconfigurations in code and IaC before they're pushed to production. Prisma Cloud CSS is powered by Checkov, the popular open source policy-as-code IaC scanner, and will soon leverage the newly contributed Kubernetes RBAC checks to identify and alert on powerful permissions in Kubernetes manifests.

Unit 42

Unit 42 brings together our world-renowned threat researchers with an elite team of security consultants to create an intelligence-driven, response-ready organization. The Unit 42 Threat Intelligence team provides threat research that enables security teams to understand adversary intent and attribution while enhancing protections offered by our products and services to stop advanced attacks. As threats escalate, Unit 42 is available to advise customers on the latest risks, assess their readiness, and help them recover when the worst occurs. The Unit 42 Security Consulting team serves as a trusted partner with state-of-the-art cyber risk expertise and incident response capabilities, helping customers focus on their business before, during, and after a breach.

Authors

Yuval Avrahami, Principal Security Researcher, Palo Alto Networks

Shaul Ben Hai, Staff Security Researcher, Palo Alto Networks

Contributors

This report would not be possible without the tremendous work and efforts taken by the larger Palo Alto Networks team. The following people assisted significantly in its creation.

Reviewing

Ariel Zelivansky

Jay Chen

Nathaniel Quist

Sharon Ben Zeev

Editing

Grace Cheung

Aimee Savran

Appendix A: Powerful Permissions by Attack Class

Manipulate Authentication/Authorization (AuthN/AuthZ)

impersonate users/groups/serviceaccounts

Impersonate other identities, such as users, groups, and service accounts.

escalate roles/clusterroles

Add arbitrary permissions to existing roles or clusterroles.

bind rolebindings/cluster role bindings

Grant existing roles or clusterroles to arbitrary identities.

approve signers & update certificatesigningrequests/approval

Have an existing signer approve a certificatesigningrequest.

control mutating webhooks

Mutate admitted roles and clusterroles.

Acquire Tokens

list secrets

Retrieve service account tokens for existing service accounts in a namespace.

This attack is set to be addressed in the future by Kubernetes Enhancement Proposal (KEP) 2799: Reduction of Secret-based Service Account Tokens.

create secrets

Issue new service account tokens for existing service accounts.

create serviceaccounts/token

Issue temporary service account tokens for existing service accounts via TokenRequests.

create pods

Assign an existing service account to a new pod, allowing the pod to access its token. Alternatively, attach the token secret of an existing service account token to a new pod as an environment variable or volume.

control pod controllers

Assign an existing service account to new or existing pods, allowing them to access its token. Alternatively, attach the token secret for an existing service account token to new or existing pods as an environment variable or volume.

control validating webhooks

Get tokens as they're created, for example when a token secret is created for a new service account.

control mutating webhooks

Get tokens as they're created, for example when a token secret is created for a new service account. Attach service account tokens to new pods.

Remote Code Execution

create pods/exec

Execute commands in an existing pod via the API server.

update pods/ephemeralcontainers

Attach a new container to an existing pod to execute code on it. Attach the container as privileged to execute code on the underlying node.

create nodes/proxy

Execute commands in the existing pod via the Kubelet.

control pods

Replace the image of a container by modifying an existing pod. Create a new privileged pod to execute code on a node.

control pod controllers

Freely create or modify pods via pod controllers like Deployments. Execute code on nodes by setting a container to be privileged.

control mutating webhooks

Mutate admitted pods and execute code by replacing the image, command, arguments, environment variable, or volumes for one of their containers.

Steal Pods

modify nodes

Evict a pod by tainting its node with the NoExecute effect. Ensure its replacement (given the pod is managed by ReplicaSets, for example) lands on a specific node by marking others as unscheduled, for example via a NoSchedule taint.

modify nodes/status

Mark a node as unschedule, for example by setting its pod capacity to 0.

create pods/eviction

Evict a pod, mainly in order to cause controllers like ReplicaSets to respawn it.

delete pods

Delete a pod to cause controllers like ReplicaSets to respawn it.

delete nodes

Delete a node to delete its pods, and cause controllers like ReplicaSets to respawn it.

modify pods/status

Match a pod's labels to the selector of an existing replica controller (e.g. a ReplicaSet) in the same namespace, to trick it into deleting an existing replica. Ensure the fake pod isn't the one being deleted by setting his ready time to be the earliest among replicas.

modify pods

Match a pod's labels to match the selector of a replica controller like a ReplicaSet in the same namespace, to trick it into deleting an existing replica.

Meddler-in-the-Middle

control endpointslices

Modify existing endpointslices for existing services to redirect some of their traffic. Create new endpointslices for existing services to redirect some of their traffic.

modify endpoints

Modify the endpoint of an existing service to redirect the service's traffic elsewhere. This attack is nullified on clusters configured to use endpointslices instead of endpoints.

modify services/status

Attach a Load Balancer IP to exploit CVE-2022-8554 and redirect traffic from pods and nodes from their designated target to existing endpoints.

modify pods/status

Match a pod's labels to the selector of a service in the same namespace to intercept some of its traffic.

modify pods

Match a pod's labels to the selector of a service in the same namespace to intercept some of its traffic.

create services

Create an ExternalIP service to exploit CVE-2022-8554 and redirect traffic from pods and nodes from their designated target to existing endpoints.

control mutating webhooks

Mutate newly admitted services, endpoints and endpointslices to redirect cluster traffic.

paloalto NETWORKS

3000 Tannery Way
Santa Clara, CA 95054

Main: +1.408.753.4000
Sales: +1.866.320.4788
Support: +1.866.898.9087

www.paloaltonetworks.com