Container Escape to Shadow Admin: GKE Autopilot Vulnerabilities

63,364 people reacted

61 14 min. read

SHARE

Recommended For You ^

By Yuval Avrahami
March 8, 2022 at 6:00 AM
Category: Cloud
Tags: containers, vulnerabilities

This post is also available in: 日本語 (Japanese)

Executive Summary

In February 2021, Google announced Autopilot, a new mode of operation in Google Kubernetes Engine (GKE). With Autopilot, Google provides a "hands-off" Kubernetes experience, managing cluster infrastructure for the customer. The platform automatically provisions and removes nodes based on resource consumption and enforces secure Kubernetes best practices out of the box.

In June 2021, Unit 42 researchers disclosed several vulnerabilities and attack techniques in GKE Autopilot to Google. Users able to create a pod could have abused these to (1) escape their pod and compromise the underlying node, (2) escalate privileges and become full cluster administrators, and (3) covertly persist administrative access through backdoors that are completely invisible to cluster operators.

Recommended For You ^

An attacker who obtained an initial foothold on an Autopilot cluster, for example, through a compromised developer's account, could have exploited these issues to escalate privileges and become a "shadow administrator," with the ability to covertly exfiltrate secrets, deploy malware or cryptominers and disrupt workloads.

Following our disclosure, Google fixed the reported issues, deploying patches universally across GKE. All Autopilot clusters are now protected.

This blog provides a technical analysis of the issues, as well as mitigations for preventing similar attacks against Kubernetes and GKE environments. For a high-level overview of the issues, please refer to our blog on the Palo Alto Networks site, Unit 42 Discloses Newly Discovered Vulnerabilities in GKE Autopilot.

Palo Alto Networks customers receive protections from the issues discussed below through the Kubernetes admission control and auditing features of Prisma Cloud.

Affected Product	Google Kubernetes Engine (GKE) Autopilot
Related Unit 42 Topics	Container Escape, Cloud

Table of Contents

Background on GKE Autopilot

Autopilot is a new mode of operation in GKE, providing what Google describes as a "hands-off" Kubernetes experience. In GKE Standard, customers manage their cluster infrastructure and pay p node. With GKE Autopilot, Google takes care of cluster infrastructure, and customers only pay fo their running pods. This allows customers to focus on their applications, cutting operational costs.

Recommended For You ^

In a nutshell, managed cluster infrastructure means Google automatically:

1. Provisions and adjusts the number of nodes according to your pods' resource consumption.
2. Enforces a built-in policy to ensure the cluster adheres to secure best practices and can be safe managed by Google.

Below is a simplified diagram of Autopilot's architecture. Components unique to Autopilot are colored in green and shown with a number corresponding to their role from the list above. Unlike GKE Standard, where nodes are visible as Compute Engine VMs, Autopilot nodes are completely managed by Google, thus colored in green.

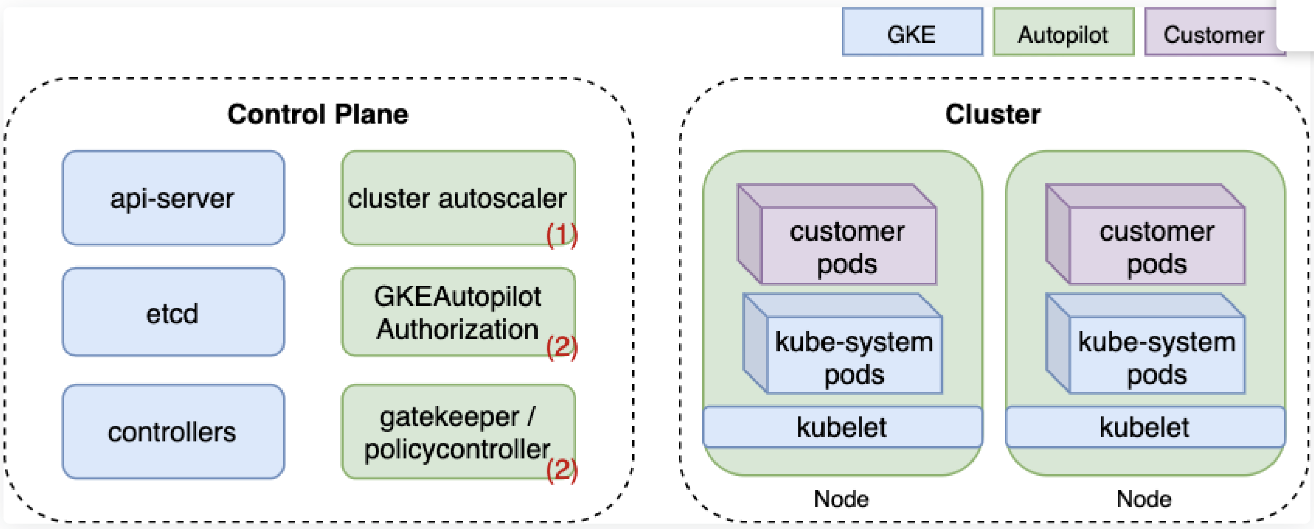
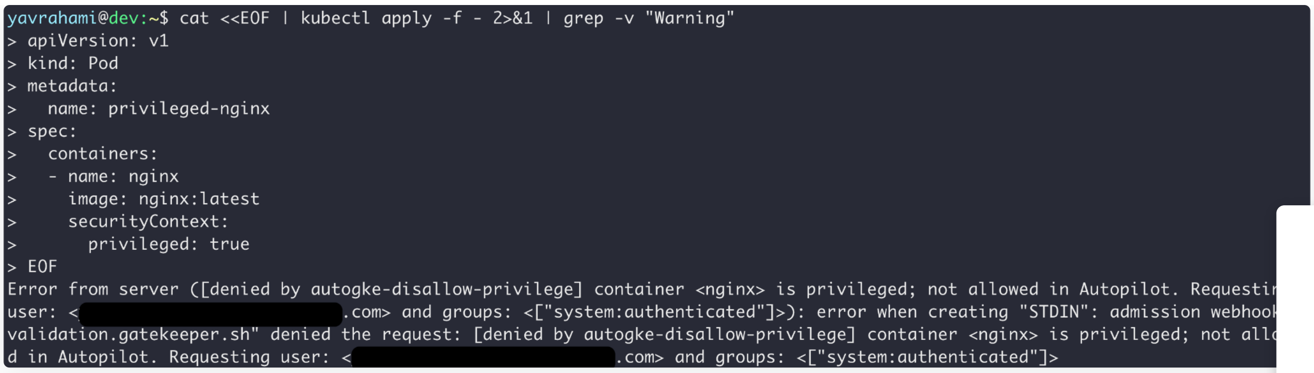


Figure 1. GKE Autopilot architecture.

As seen in Figure 1, two components enforce Autopilot's policy. First is an OPA Gatekeeper validating admission webhook, an open-source project widely used for policy enforcement in Kubernetes. The second is a proprietary Kubernetes authorization mode named GKEAutopilot, which Google implemented by modifying the Kubernetes source code.

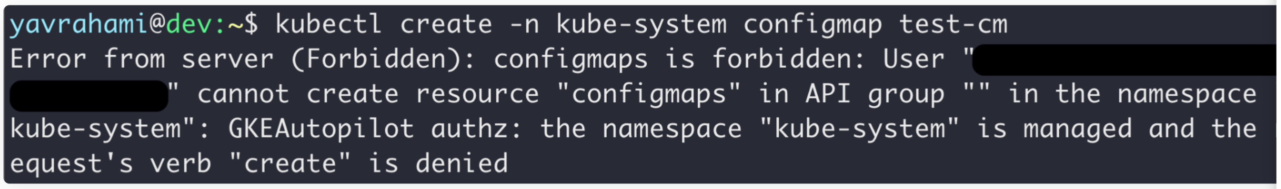
The built-in policy serves two purposes: (a) prevent users from accessing cluster components managed by Google, like nodes; and (b) uphold secure Kubernetes best practices. For example, Autopilot forbids running privileged containers, fulfilling both (a) and (b).



```
yavrahami@dev:~$ cat <<EOF | kubectl apply -f - 2>&1 | grep -v "Warning"
> apiVersion: v1
> kind: Pod
> metadata:
>   name: privileged-nginx
> spec:
>   containers:
>   - name: nginx
>     image: nginx:latest
>     securityContext:
>       privileged: true
> EOF
Error from server ([denied by autogke-disallow-privilege] container <nginx> is privileged; not allowed in Autopilot. Requestin
user: <[redacted].com> and groups: <["system:authenticated"]>): error when creating "STDIN": admission webhook
validation.gatekeeper.sh" denied the request: [denied by autogke-disallow-privilege] container <nginx> is privileged; not allo
d in Autopilot. Requesting user: <[redacted].com> and groups: <["system:authenticated"]>
```

Figure 2. Autopilot's built-in policy prevents privileged containers (Gatekeeper).

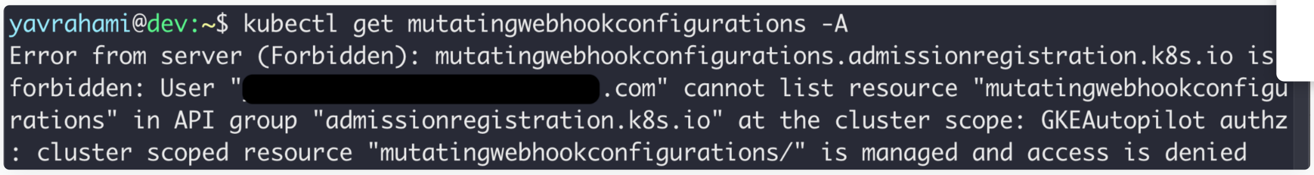
Autopilot's policy goes beyond preventing container escapes. Figures 3, 4, and 5 highlight a few interesting examples. GKE's documentation lists every limit enforced by the policy.



```
yavrahami@dev:~$ kubectl create -n kube-system configmap test-cm
Error from server (Forbidden): configmaps is forbidden: User "[redacted]
[redacted]" cannot create resource "configmaps" in API group "" in the namespace
kube-system": GKEAutopilot authz: the namespace "kube-system" is managed and the
equest's verb "create" is denied
```

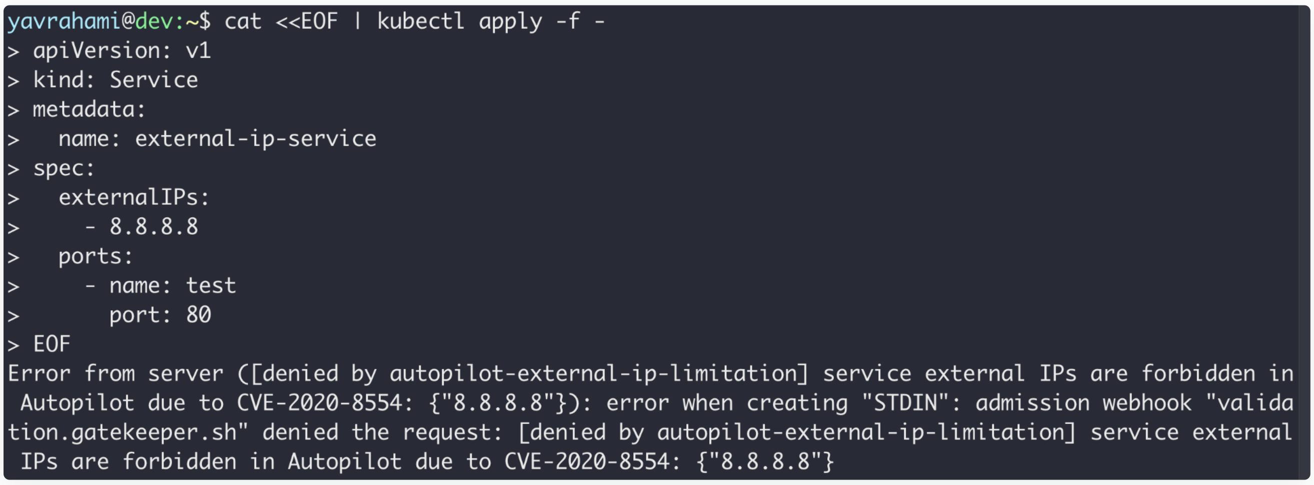
Figure 3. The kube-system namespace is managed, customers are limited to read-only access.

Recommended For You



```
yavrahami@dev:~$ kubectl get mutatingwebhookconfigurations -A
Error from server (Forbidden): mutatingwebhookconfigurations.admissionregistration.k8s.io is
forbidden: User "[redacted].com" cannot list resource "mutatingwebhookconfigu
rations" in API group "admissionregistration.k8s.io" at the cluster scope: GKEAutopilot authz
: cluster scoped resource "mutatingwebhookconfigurations/" is managed and access is denied
```

Figure 4. Users cannot list or create mutating admission webhooks.



```
yavrahami@dev:~$ cat <<EOF | kubectl apply -f -
> apiVersion: v1
> kind: Service
> metadata:
>   name: external-ip-service
> spec:
>   externalIPs:
>     - 8.8.8.8
>   ports:
>     - name: test
>       port: 80
> EOF
Error from server ([denied by autopilot-external-ip-limitation] service external IPs are forbidden in
 Autopilot due to CVE-2020-8554: {"8.8.8.8"}): error when creating "STDIN": admission webhook "valida
tion.gatekeeper.sh" denied the request: [denied by autopilot-external-ip-limitation] service external
 IPs are forbidden in Autopilot due to CVE-2020-8554: {"8.8.8.8"}
```

Figure 5. External IP services are denied to protect against CVE-2020-8554.

Reading the error messages in the Figures above, you can see that Gatekeeper prevented the operations in Figures 2 and 5, while the GKEAutopilot authorization mode prevented the operations in Figures 3 and 4.

Attack Surfaces Unique to GKE Autopilot

Autopilot's built-in policy blocks several exploitation paths out of the box, providing better security posture compared to standard Kubernetes or GKE Standard. That being said, it also creates attack surfaces unique to Autopilot:

1. Administrators may rely on Autopilot's policy to prevent risky configurations. If attackers can somehow circumvent that policy, they may escalate privileges via methods customers expect to b blocked, like deploying a privileged container.

2. Autopilot administrators aren't fully privileged, restricted by the built-in policy from accessing nodes and certain privileged Kubernetes APIs. If attackers can bypass Autopilot's policy, they may gain higher privileges than administrators, opening the door for invisible backdoors.

The following sections present vulnerabilities, privilege escalation techniques and persistence methods we identified that fall under these attack surfaces. Chained together, they allow a restricted user who can create a pod to (1) compromise nodes, (2) escalate privileges to an unrestricted cluster administrator and (3) install invisible and persistent backdoors to the cluster.

Recommended For You ^

Masquerading as Allowlisted Workloads to Compromise Nodes

Our research began at the following paragraph in Autopilot's documentation:

"Our intent is to prevent unintended access to the node virtual machine. We accept submissions to that effect through the Google Vulnerability Reward Program (VRP)..."

This seemed like an interesting challenge, and so we created an Autopilot cluster and started looking around. Autopilot installed OPA Gatekeeper onto the cluster along with several policies (called "constraints" in Gatekeeper terminology) in charge of preventing risky configurations like privileged containers. The cluster also had a Custom Resource Definition (CRD) that seemed interesting, named `allowlistedworkloads`.

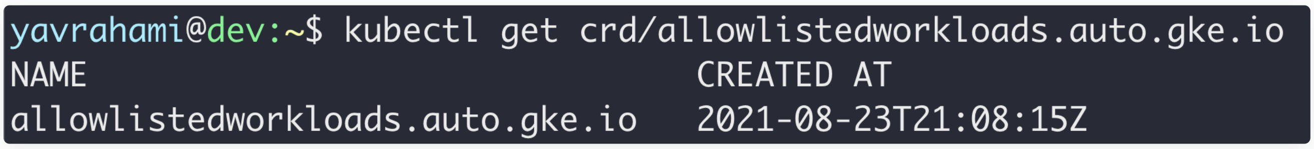
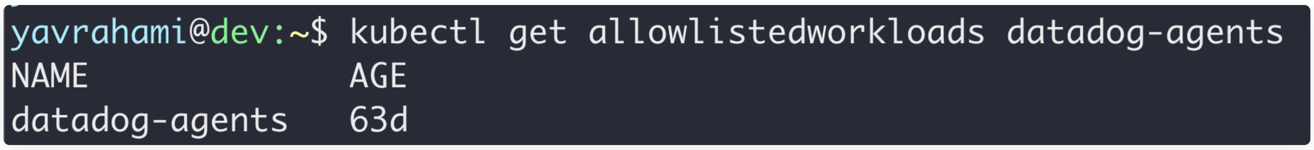

```
yavrahami@dev:~$ kubectl get crd/allowlistedworkloads.auto.gke.io
NAME                                CREATED AT
allowlistedworkloads.auto.gke.io    2021-08-23T21:08:15Z
```

Figure 6. Autopilot installs a Custom Resource Definition (CRD) named allowlistedworkloads

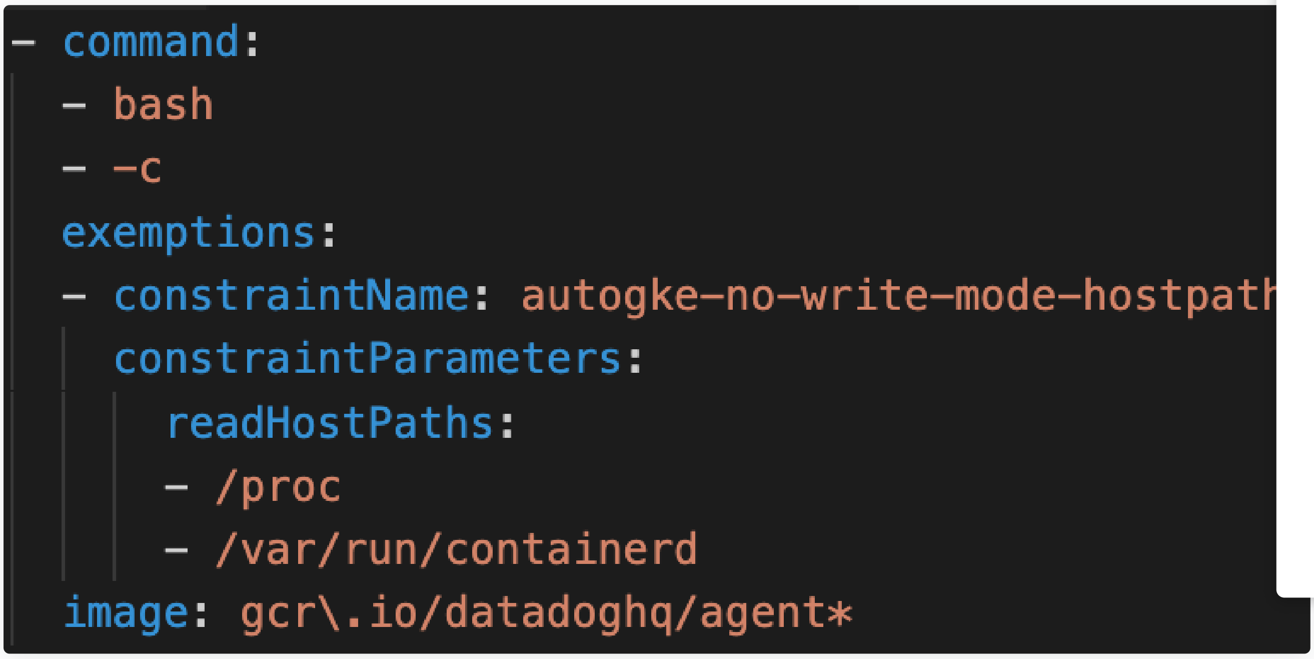
As shown earlier in Figure 2, Autopilot forbids pod configurations that could allow container escapes. To support add-ons that require some level of node access, Autopilot created a notion of allow-listed workloads. If a container matches an allow-listed workload, it's permitted to use the privileged features specified in the `allowlistedworkload` configuration. In June, the only allow-listed workloads were Datadog agents.



```
yavrahami@dev:~$ kubectl get allowlistedworkloads datadog-agents
NAME              AGE
datadog-agents    63d
```

Figure 7. Datadog agent allowlistedworkload

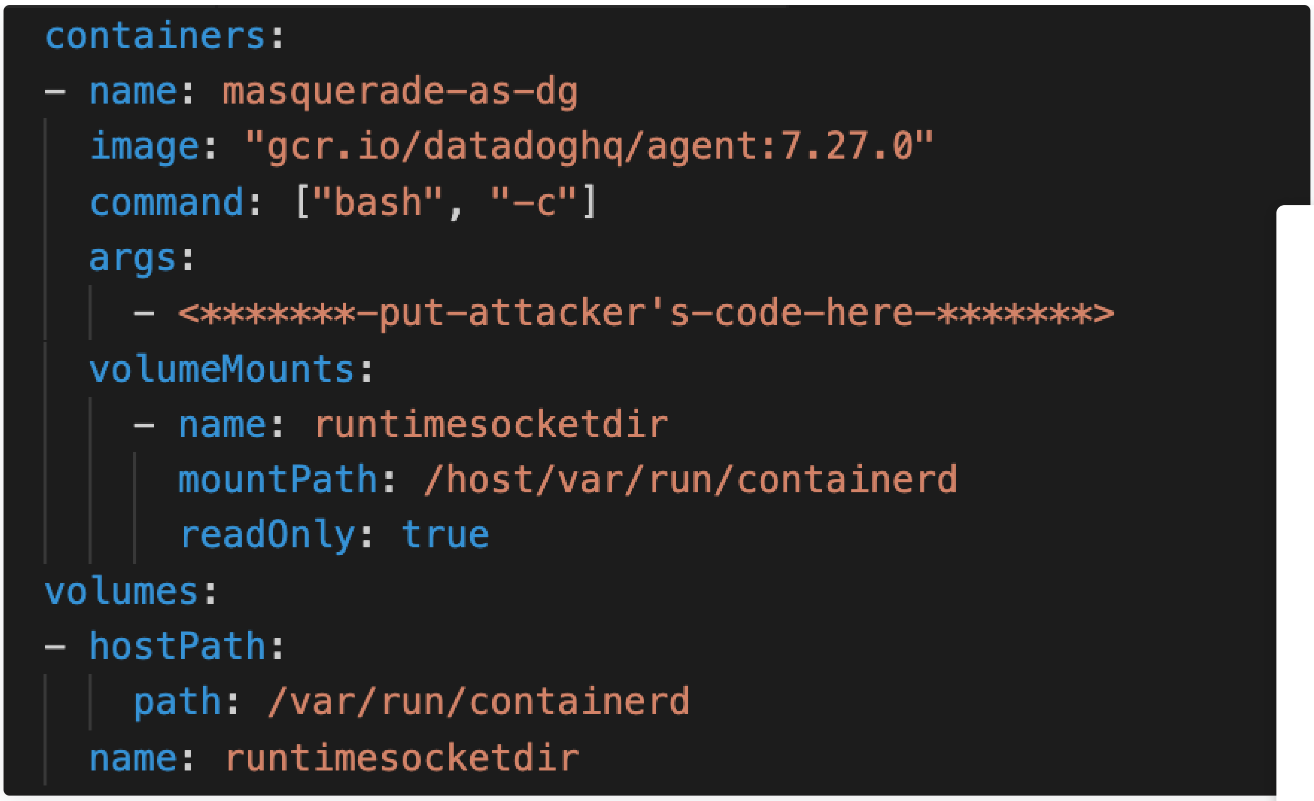
Below is the allow-listed workload configuration for one of the Datadog agents that caught our attention. If a container specifies the listed command and image, it's allowed to mount the listed host paths in read-only volumes.



```
- command:
  - bash
  - -c
  exemptions:
  - constraintName: autogke-no-write-mode-hostpath
    constraintParameters:
      readHostPaths:
      - /proc
      - /var/run/containerd
  image: gcr\.io/datadoghq/agent*
```

Figure 8. One of the allowlistworkloadconfiguration examples for Datadog agents.

The issue here is insufficient verification. Only checking the command and image isn't enough to ensure the container runs Datadog code. Using the following `PodSpec`, a container can masquerade as the Datadog agent while running attacker-controlled code, and abuse the exposed host volumes to break out.



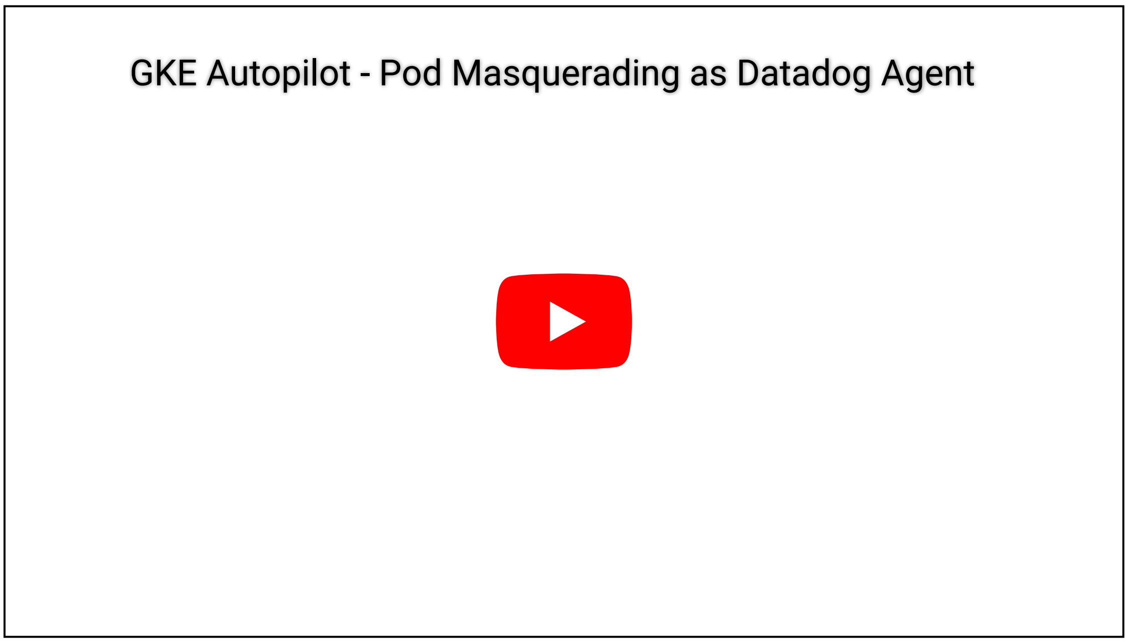
```
containers:
- name: masquerade-as-dg
  image: "gcr.io/datadoghq/agent:7.27.0"
  command: ["bash", "-c"]
  args:
    - <*******-put-attacker's-code-here-*******>
  volumeMounts:
    - name: runtimesocketdir
      mountPath: /host/var/run/containerd
      readOnly: true
volumes:
- hostPath:
    path: /var/run/containerd
  name: runtimesocketdir
```

Figure 9. Masquerading as the Datadog agent.

In the video below, a malicious user deploys a pod masquerading as the Datadog agent. The pod takes over its underlying node through the following steps:

1. Abuse the mounted containerd socket to create a privileged container that mounts the host filesystem.

2. Have that privileged container install a systemd service that spawns a reverse shell from the node to an attacker-controlled machine.



Video 1. Masquerading as an allowlistedworkload to compromise the underlying node.

Impact of Node Compromise

An attacker who can create a pod may exploit this issue to create malicious containers that escape and take over their underlying nodes. Autopilot users expect the platform to prevent this kind of attack, and will be caught off-guard.

Node compromise opens up the following attack vectors:

1. The attacker immediately gains control over neighboring pods and their service account tokens potentially escalating privileges and spreading to other namespaces.
2. The attacker can query the node's instance metadata endpoint for an access token. By default, this token provides read access to cloud storage in the customer's project.
3. Since Autopilot administrators cannot access nodes, attackers may abuse this issue to install covert malware or cryptominers on them. However, Autopilot automatically scales nodes, so ensuring malware persists isn't straightforward.
4. The attacker gains access to the underlying Kubelet credentials, allowing visibility to nearly all cluster objects.

Finally, because Autopilot only bills per running pods, crafty users could have abused this issue to cut some costs, running some workloads directly on nodes. We advise reducing bills in more legitimate ways.

Recommended For You ^

Escalating to Unrestricted Administrators

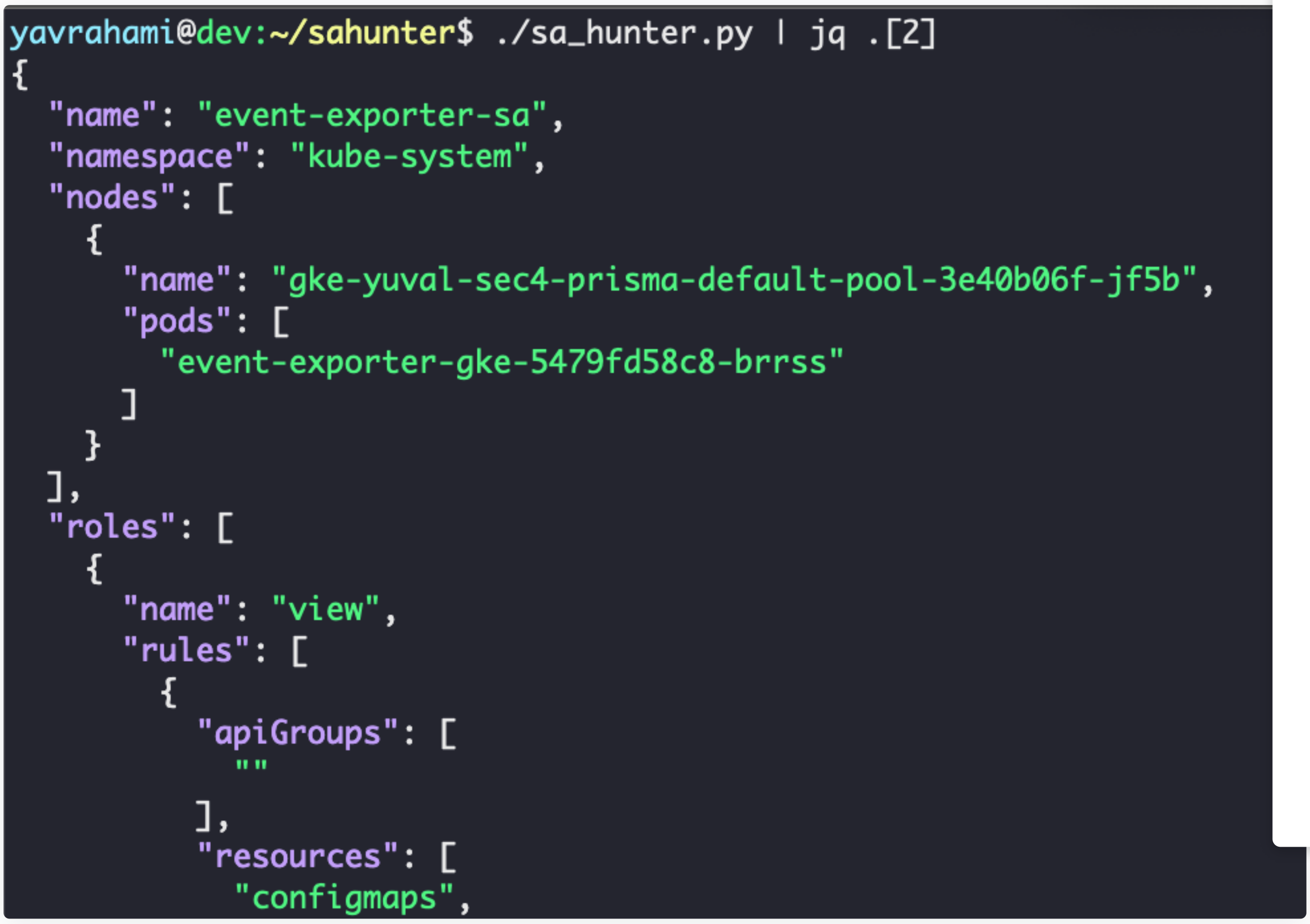
Following the trajectory of a motivated intruder, we looked for reliable methods of escalating this container escape into a full cluster takeover. By compromising a node, attackers can steal the service account tokens of neighboring pods. Naturally, it would make sense to target nodes hosting pods with powerful service accounts. These may be pods deployed by the user, or more interestingly, system pods deployed natively in all Autopilot clusters.

After examining the built-in policy, we discovered that **Autopilot completely exempts kube-system service accounts**. This made `kube-system` pods the most interesting targets, as stolen tokens could be used freely without worrying about the policy.

```
non_privileged_actor(privilegedUsers, privilegedGroups, userInfo) = true {
  not contains_element(privilegedUsers, userInfo.username)
  not startswith(userInfo.username, "system:serviceaccount:kube-system:")
  groupNames := {grp | grp := userInfo.groups[_]}
  allowedGroupNames := {grp | grp := privilegedGroups[_]}
  sub := groupNames & allowedGroupNames
  count(sub) == 0
}
```

Figure 10. Autopilot's policy exempts kube-system service accounts in line 3.

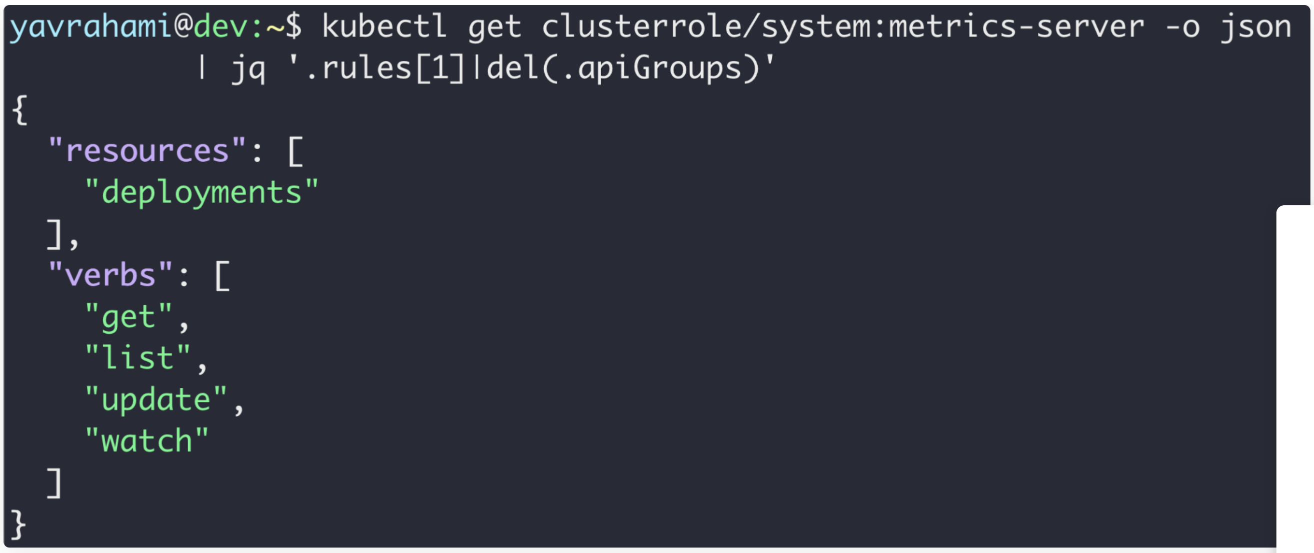
To search for powerful pods in Autopilot, we created `sa-hunter`, a Python tool that maps pods' service accounts to their Kubernetes permissions (i.e. roles and clusterroles). Existing tools link service accounts to their permissions, but don't show whether any pods actually use a given service account. Figure 11 shows an example output of `sa-hunter`:



```
yavrahami@dev:~/sahunter$ ./sa_hunter.py | jq .[2]
{
  "name": "event-exporter-sa",
  "namespace": "kube-system",
  "nodes": [
    {
      "name": "gke-yuval-sec4-prisma-default-pool-3e40b06f-jf5b",
      "pods": [
        "event-exporter-gke-5479fd58c8-brrss"
      ]
    }
  ],
  "roles": [
    {
      "name": "view",
      "rules": [
        {
          "apiGroups": [
            ""
          ],
          "resources": [
            "configmaps",
```

Figure 11. sa-hunter output, linking running pods to their permissions.

`sa-hunter` found two powerful `kube-system` pods installed by default: `stackdriver-metadata-agent-cluster-level` and `metrics-server`. Both pods can update existing deployments, as shown in Figure 12. This privilege may appear innocent at first glance, but it is enough to escalate to full cluster admin. Interestingly, these pods are also deployed by default in GKE Standard, **making the following privilege escalation technique relevant to all GKE clusters, Standard and Autopilot**.



```
yavrahami@dev:~$ kubectl get clusterrole/system:metrics-server -o json
          | jq '.rules[1]|del(.apiGroups)'
{
  "resources": [
    "deployments"
  ],
  "verbs": [
    "get",
    "list",
    "update",
    "watch"
  ]
}
```

Figure 12. Privileged role assigned to the metrics-server pod can update deployments.

Recommended For You ^

After taking over a node hosting either the `stackdriver-metadata-agent-cluster-level` or `metrics-server` pod, an attacker can harvest their service account token from the node filesystem. Armed with that token, the attacker can attain the privileges of any service account in the cluster with three simple steps:

1. Update an existing deployment's service account to the target service account. There are a number of preinstalled deployments, any one of which can be used for this step.

2. Add a malicious container to that deployment.

3. Have that malicious container retrieve the target service account token mounted in the container at `/run/secrets/kubernetes.io/serviceaccount/token`.

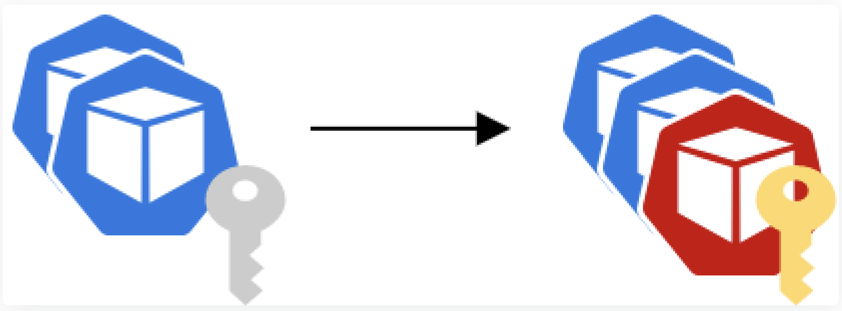
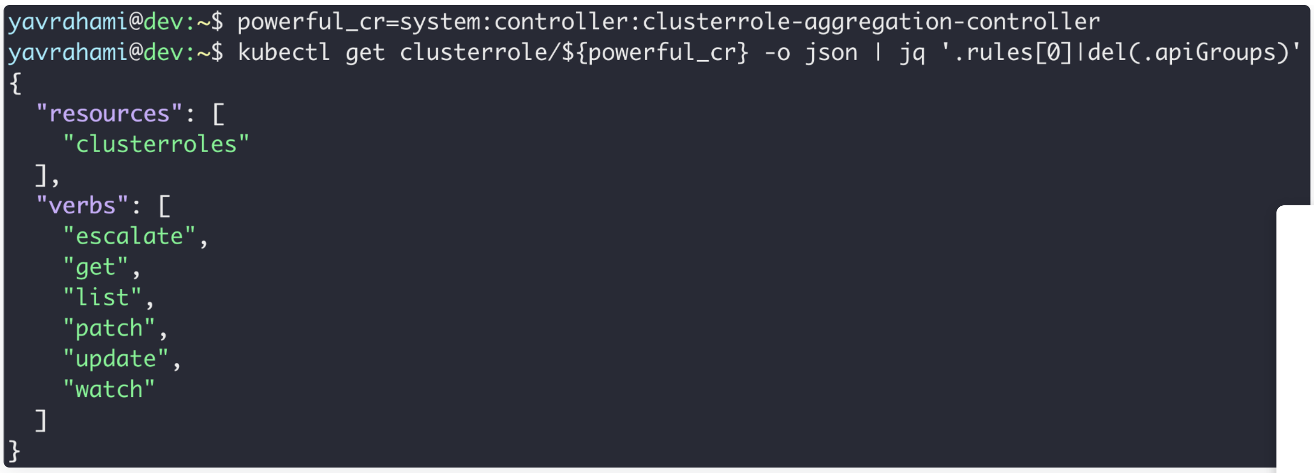


Figure 13. Abusing deployment update privileges to obtain any service account's token.

For this to be a meaningful privilege escalation, the attacker would need to target a powerful service account. The `kube-system` namespace offers a number of preinstalled, extremely powerful service accounts to choose from. The `clusterrole-aggregation-controller` (`CRAC`) service account is probably the leading candidate, as it can add arbitrary permissions to existing cluster roles.



```
yavrahami@dev:~$ powerful_cr=system:controller:clusterrole-aggregation-controller
yavrahami@dev:~$ kubectl get clusterrole/${powerful_cr} -o json | jq '.rules[0]|del(.apiGroups)'
{
  "resources": [
    "clusterroles"
  ],
  "verbs": [
    "escalate",
    "get",
    "list",
    "patch",
    "update",
    "watch"
  ]
}
```

Figure 14. The clusterrole-aggregation-controller service account can escalate cluster roles.

After using the technique illustrated in Figure 13 to obtain `CRAC`'s token, the attacker can update the cluster role binded to `CRAC` to possess all privileges. At this point, the attacker is effectively cluster admin, and is also exempt from Autopilot's policy (as seen in Figure 10).

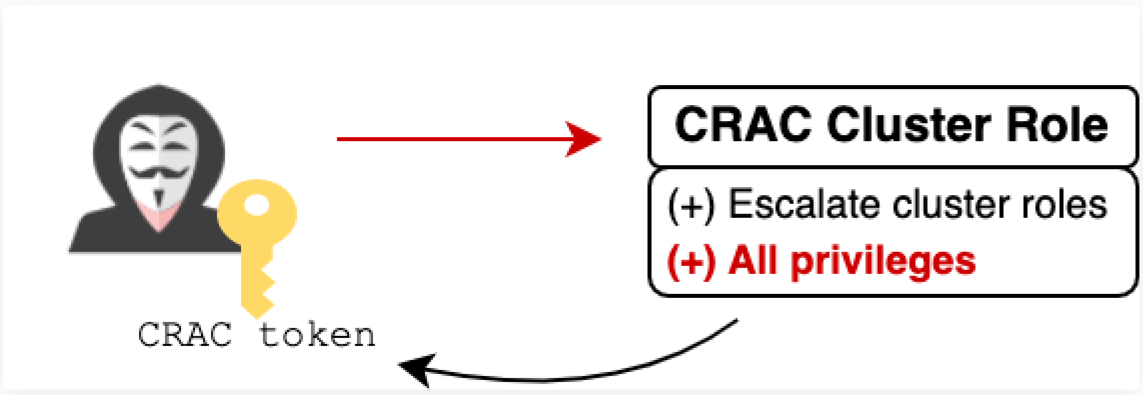


Figure 15. The clusterrole-aggregation-controller's token can add admin privileges to itself.

Rewinding, if the attacker wants to chain this privilege escalation technique with the container escape discussed earlier, he needs to somehow schedule his breakout pod on a node hosting either the `stackdriver-metadata-agent-cluster-level` or `metrics-server` pod. Although Autopilot rejects pods that have `nodeSelectors`, it does permit the simplest form of node assignment — the `nodeName` field.

The `nodeName` field assures the breakout pod will land on the target node as long as it has adequate resources for another pod. Even if there's no room on the target node, the attacker still has a couple of options. He can either (1) watch the target node and wait for a pod to be deleted; or (2) create pods to trigger a node scale up, tricking Autopilot's autoscaler into redistributing workloads so that a powerful pod ends up on an emptier node.

Full Chain: Invisible Backdoors via Mutating Admission Webhooks

Video 2 shows the full attack chain, combining the container escape with the privilege escalation route built into GKE. Following exploitation, the attacker has higher privileges than Autopilot administrators, as he's exempt from the built-in policy (as seen in Figure 10). **This level of access can be abused to install invisible and persistent backdoors** in the form of mutating admission webhooks.

Mutating admission webhooks receive any objects created or updated in the cluster, including po and secrets. If that's not scary enough, these webhooks can also arbitrarily mutate any received object, making them a ridiculously powerful backdoor. As shown earlier in Figure 4, Autopilot administrators cannot list mutating admission webhooks, and thus will never see this backdoor.

Recommended For You ^

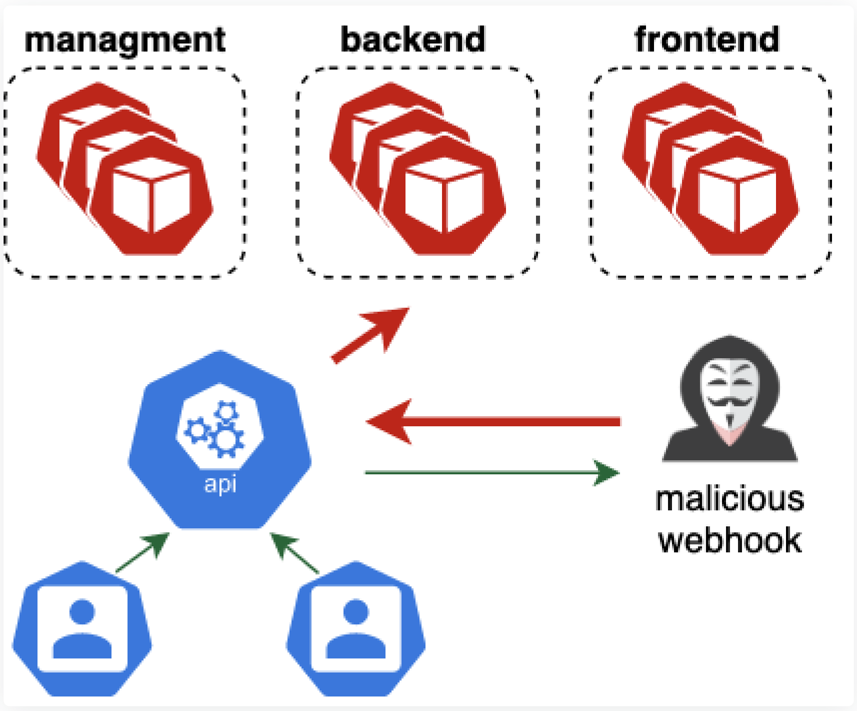
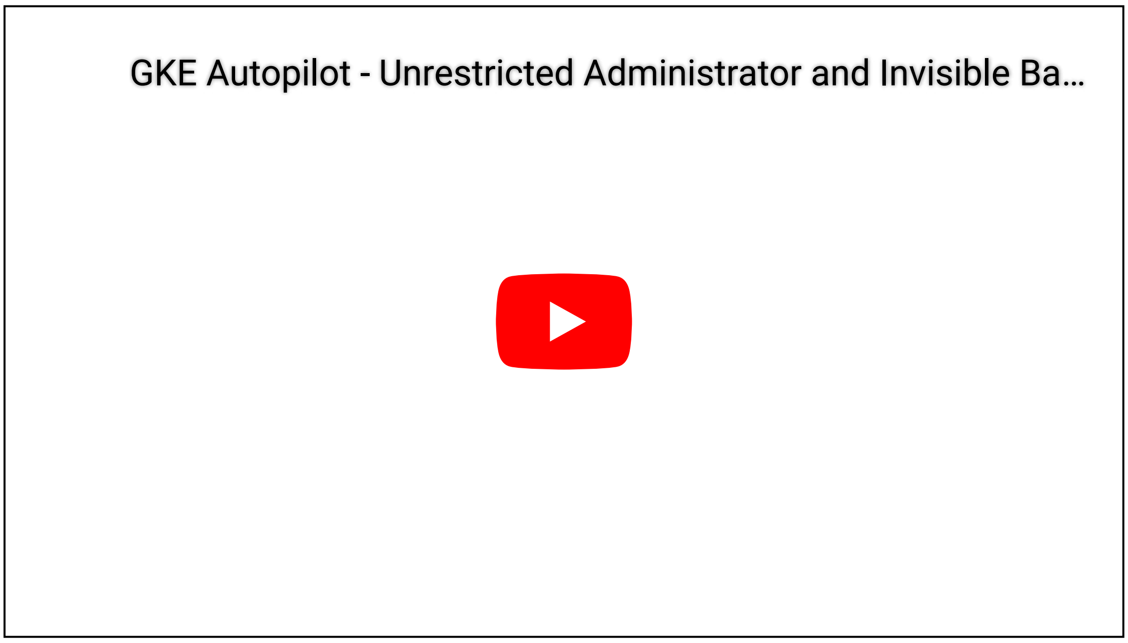
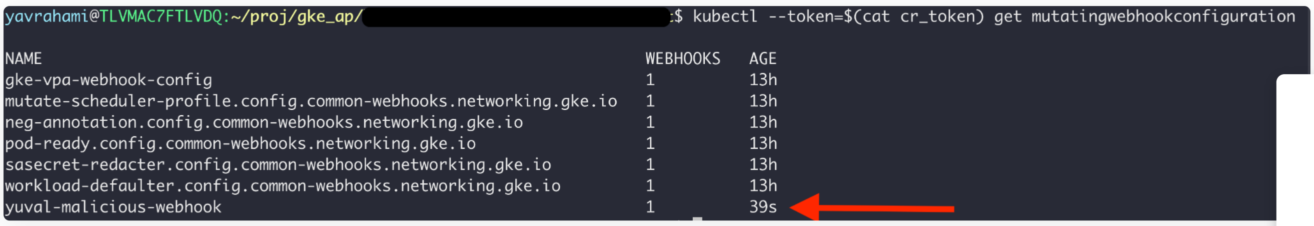


Figure 16. Becoming a shadow administrator through an invisible mutating admission webhook.


GKE Autopilot - Unrestricted Administrator and Invisible Ba...

Video 2. Escalating from pod creation to an unrestricted administrator and an invisible backdoor.



```
yavrahami@TLVMAC7FTLVDQ:~/proj/gke_ap/[redacted]:$ kubectl --token=$(cat cr_token) get mutatingwebhookconfiguration
NAME                                                     WEBHOOKS   AGE
gke-vpa-webhook-config                                   1          13h
mutate-scheduler-profile.config.common-webhooks.networking.gke.io   1          13h
neg-annotation.config.common-webhooks.networking.gke.io  1          13h
pod-ready.config.common-webhooks.networking.gke.io       1          13h
sasecret-redacter.config.common-webhooks.networking.gke.io   1          13h
workload-defaulter.config.common-webhooks.networking.gke.io  1          13h
yuval-malicious-webhook                                  1          39s
```

Figure 17. A malicious mutating admission webhook installed by abusing the reported issues. Note that seeing the backdoor requi the unrestricted admin token acquired in the attack.

Full Chain: Impact

Before the fixes, attackers could have exploited the presented issues to transform a limited breacl into a full cluster takeover on any Autopilot cluster. Using webhooks that are invisible to users, attackers can covertly persist their administrative access, effectively becoming "shadow administrators." At that point, they could have covertly exfiltrated secrets, deployed malware or cryptominers, and disrupted workloads.

Other Issues

During our research we found two additional issues that allow node compromise, but with lower impact. The first involves two service account names in the default namespace that were exempt from Autopilot's policy: `csi-attacher` and `otelsvc`. If an attacker gained control over the default namespace, it would have been possible to create these service accounts to bypass the built-in policy. The attacker could then create privileged pods to compromise nodes, and use the discussed privilege escalation technique to take over the entire cluster.

The second attack exploited CVE-2020-8554 through Load Balancer services to compromise nodes, but required admin privileges to exploit. The attack scenario here is an attacker who has already compromised an Autopilot cluster and is looking to bypass the built-in policy to establish a covert backdoor.

Fixes and Mitigations

Following our advisory and over the course of the last several months, Google deployed numerous fixes and mitigations to GKE Autopilot. These prevent the reported attack and harden the platform against similar exploits.

1. Cluster administrators can now list, view and even create mutating admission webhooks, preventing their abuse as invisible backdoors.

2. Google hardened the verification process of `allowlistedworkloads`.

3. Policy enforcement moved from OPA Gatekeeper to Google's policy controller, allowing customers to deploy their own Gatekeeper instance. Customers can now enforce their own policy

on top of the built-in one. As defense-in-depth and to mitigate possible future issues, we recommend deploying the same policies you would have deployed to GKE standard.

4. The built-in policy is no longer visible.

5. The `csi-attacher` and `otelsvc` service accounts are no longer exempt from Autopilot's policy.

6. Google open-sourced a policy for OPA Gatekeeper that restricts the powerful `kube-system` pods abused in the attack. The policy prevents these pods from assigning a new service account t an existing pod. See GKE's hardening guide for more information.

We highly recommend reading Google's official advisory, which describes the issues from Google' perspective and lists their mitigations.

Preventing Similar Attacks on Kubernetes Environments

The presented attack can be classified as a Kubernetes privilege escalation, where an attacker wit limited access obtains broader permissions over a cluster. This kind of follow-on attacker activity must start from an initial breach: a malicious image in your cluster supply chain, a vulnerable publi exposed service, stolen credentials or an insider threat. Securing your cluster's software supply chain, identities and external perimeter can reduce the chances of such breaches occurring in you clusters.

Recommended For You ^

Sophisticated attackers may still find creative ways to infiltrate clusters. Proactively solving comm misconfigurations, like Kubelets that allow unauthenticated access, can significantly reduce the internal attack surface available to an intruder. Security controls such as `NetworkPolicies` and `PodSecurityStandards` further restrict and demoralize malicious actors.

In the presented attack chain, the attacker only had to compromise one node to take over the entire cluster. The underlying issue wasn't the node's permissions, but those of the powerful pods it hosted, which included the ability to update deployments. This permission and others may appear somehow restricted at first glance, but are equivalent to cluster admin.

Powerful pods are still common in production clusters: natively installed by the underlying Kubernetes platform or introduced through popular open-source add-ons. If a node hosting a powerful pod is compromised, the attacker can easily harvest the pod's powerful service account token to spread in the cluster.

Tackling powerful pods is complex, mostly because their permissions may be rightfully needed. The first step is detection — identifying whether powerful pods exist in your cluster. We hope `sa-hunter` can help with that, and we plan to release additional tools that focus on automating detection of powerful pods.

If you identified powerful pods in your cluster, we recommend taking one of the approaches below:

1. If you manage the powerful pod, consider whether it's possible to strip unnecessary privileges from its service account, or scope them to specific namespaces or resource names.

2. If these pods are part of an external solution, reach out to the relevant cloud provider or project to pursue reducing the pod's privileges. If the pod is deployed by a managed Kubernetes service, it may be possible for them to replace it with a control plane controller.

3. Some Kubernetes permissions are too broad, meaning the pod in question may not require acce to the dangerous operations its permissions expose. In that case, it may be possible to implement policy (e.g. via OPA Gatekeeper) that prevents the pod from performing certain dangerous operations, or even better, restricts the pod to a set of allowed and expected operations.

4. Use Taints, NodeAffinity or PodAntiAffinity rules to isolate powerful pods from untrusted or publicly exposed ones, ensuring they don't run on the same node.

Recommended For You ^

As an example for the third approach, the following Rego policy can stop the presented privilege escalation attack. The attack abused system pods who can update deployments to replace the service account of an existing deployment to a powerful one. Inspecting the source code of these powerful pods revealed they don't need the ability to change the service account of deployments they update. The policy below capitalizes on that and forbids these pods from unexpectedly updating deployments' service accounts. Prisma Cloud users on GKE are encouraged to import th policy as an admission rule set on Alert.

```
match[{"msg": msg}] {
 input.request.object.kind == "Deployment"
 request_by_powerful_dep_update_sa(input.request.userInfo.username)
 old_spec := input.request.oldObject.spec.template.spec
 new_spec := input.request.object.spec.template.spec
 new_service_account := is_updating_the_service_account(old_spec, new_spec)
 msg := sprintf("SA '%v' may be compromised, it unexpectedly tried to replace the ser
}

request_by_powerful_dep_update_sa(username) { # metrics-server pod on GKE
 username == "system:serviceaccount:kube-system:metrics-server"
} { # stackdriver pod on older GKE clusters
 username == "system:serviceaccount:kube-system:metadata-agent"
}

is_updating_the_service_account(oldspec, newspec) = new_service_account {
 oldspec.serviceAccountName != newspec.serviceAccountName
 new_service_account := newspec.serviceAccountName
} {
 not has_key(oldspec, "serviceAccountName")
 new_service_account := newspec.serviceAccountName
}

has_key(obj, k) {
 _ = obj[k]
}
```

Conclusion

As organizations migrate to Kubernetes, attackers follow suit. Recent malware samples like Silocape indicate adversaries are evolving beyond simple techniques into advanced Kubernetes tailored attacks. Against sophisticated attackers, solely securing the cluster's perimeter may not be enough. We encourage defenders to adopt policy and audit engines that enable detection and prevention of follow-on, "stage 2" attacker activities, and we hope this research can highlight how those may look.

Prisma Cloud customers are encouraged to enable our Kubernetes admission control and auditing features aimed at tackling this threat.

We'd like to thank Google for their cooperation in resolving these issues, the bounty reward and their wonderful policy that doubles bounties donated to charity.

Palo Alto Networks has shared these findings, including file samples and indicators of compromise with our fellow Cyber Threat Alliance members. CTA members use this intelligence to rapidly depl protections to their customers and to systematically disrupt malicious cyber actors. Learn more about the Cyber Threat Alliance.

Recommended For You ^

Additional Resources

- GKE Autopilot documentation
- sa-hunter
- Siloscape: First Known Malware Targeting Windows Containers to Compromise Cloud Environments
- Protecting Against an Unfixed Kubernetes Man-in-the-Middle Vulnerability (CVE-2020-8554)

Most Read Articles

From DarkGate to AsyncRAT: Malware Detected and Shared As Unit 42 Timely Threat Intelligence

From DarkGate to AsyncRAT: Malware Detected and Shared As Unit 42 Timely Threat Intelligence

- By Samantha Stallings and Brad Duncan
- December 29, 2023 at 6:00 AM

37 9 min. read

Tackling Anti-Analysis Techniques of GuLoader and RedLine Stealer