Welcome to the Access Control and Security in Kubernetes and Google Kubernetes Engine module. In previous modules you've learned how to run workloads in your cluster. Now you will learn how to make sure that appropriate access policies are in place, to make your cluster and applications more secure.

Learn how to ...

Understand Kubernetes authentication and authorization

Define Kubernetes RBAC roles and role bindings for accessing resources in namespaces

Define Kubernetes RBAC cluster roles and cluster role bindings for accessing cluster-scoped resources

Define Kubernetes pod security policies to only allow pods with specific security-related attributes to run

Understand the structure of Cloud IAM

Define IAM roles and policies for Google Kubernetes Engine cluster administration

In this module, you'll learn how to:
Understand Kubernetes authentication and authorization, in order to control who is able to do which things inside your GKE clusters. You'll learn about role-based access control, which we like to call RBAC . Kubernetes RBAC lets you define roles and role bindings, so you can manage access to resources in Kubernetes namespaces.
You'll also learn to define Kubernetes RBAC cluster roles and cluster role bindings, to manage resources that are within the scope of a cluster. That includes sharing roles across namespaces and granting permissions across all namespaces.
You'll learn about defining Kubernetes pod security policies to only allow pods with specific security-related attributes to run.
You'll also learn more about the structure of Google Cloud Identity and Access Management, which we call Cloud IAM, and define IAM roles and policies for Google Kubernetes Engine cluster administration.

Agenda

Authentication and Authorization

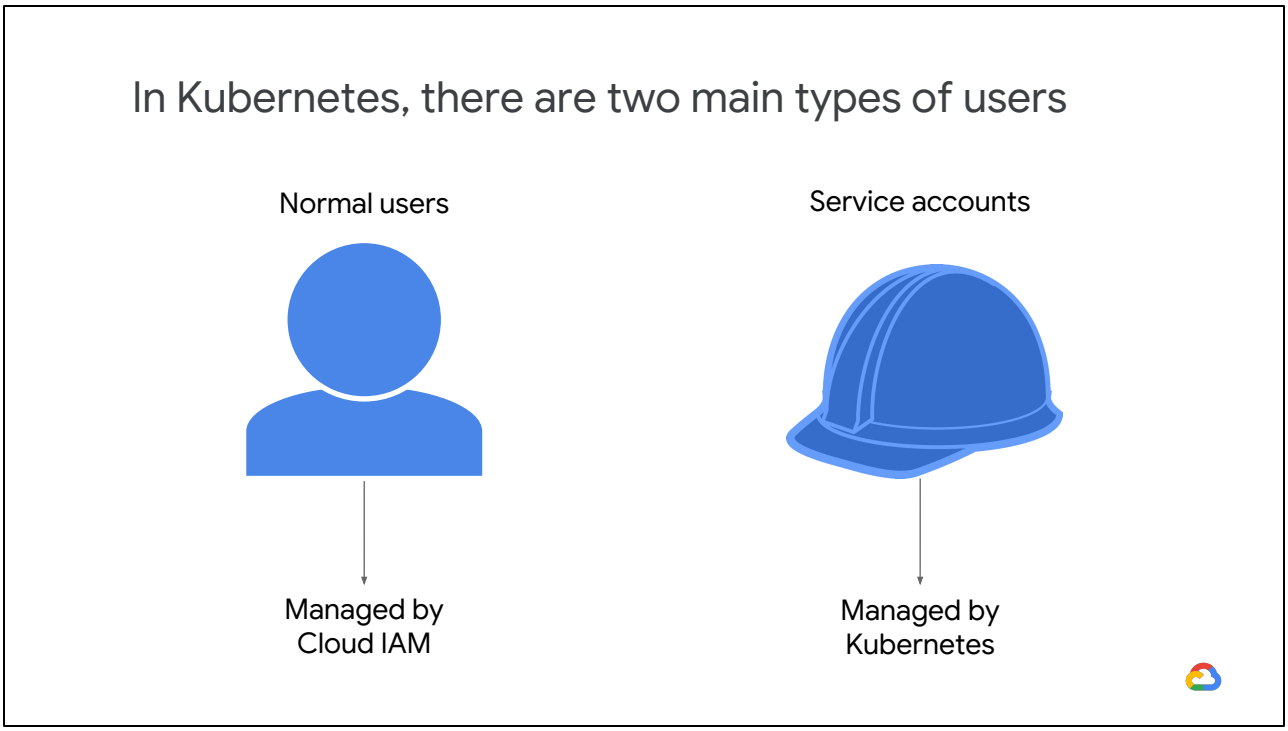
Cloud IAM

Kubernetes RBAC

Kubernetes Control Plane Security

Pod Security

Let’s start with authentication and authorization. In this first lesson, you will learn about how user and service accounts work in Kubernetes. You’ll then learn about how to create and manage those accounts in GKE.



In Kubernetes, there are two main types of users, Normal users and Kubernetes service accounts. Both Google Cloud Platform and Kubernetes have an entity called a “service account,” but they’re not the same. Keep in mind that your day-to-day focus will be on normal users. Normal users are managed outside of Kubernetes. That’s true everywhere you can run Kubernetes.

When you use Kubernetes clusters in GKE, you’ll define most user accounts through Google’s identity service. Informally, that encompasses the consumer Google accounts you’re familiar with, and accounts in G Suite domains. If you don’t have a G Suite domain, you should get a Google Cloud Identity domain for your business, which is at no additional charge in most circumstances. A Cloud Identity domain gives you much more organizational control than using consumer Google accounts for all your employees would. All these kinds of accounts together make up Google Cloud

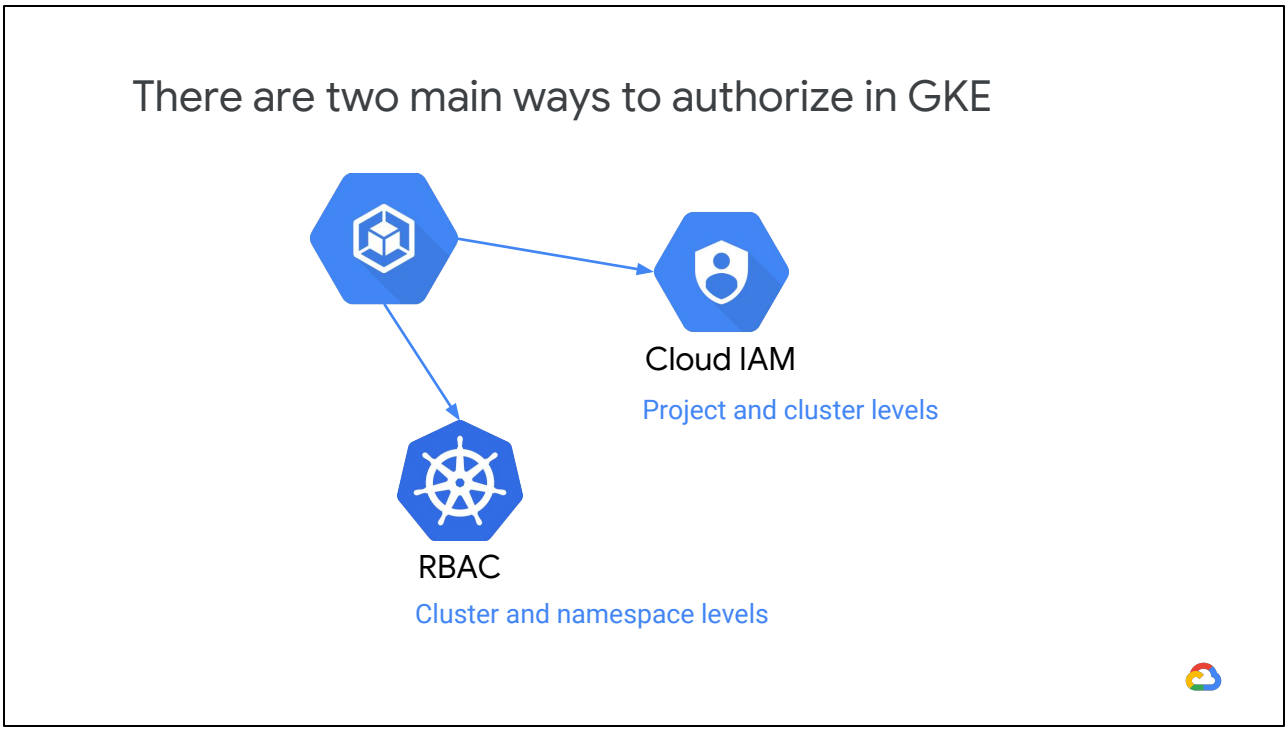
Identity. We use the word “member” to refer to each identity defined by the service. It’s a flexible service: it lets you define user identities directly, or mirror them from your existing directory service such as Active Directory. It also lets you group users, which turns out to be the key part of an important best practice discussed later in the module.

GCP’s Cloud Identity and Access Management service, “Cloud IAM,” is where you bind permissions in your GCP project to members. And then you need to tell Kubernetes about the members you want to give permissions inside your clusters to. Any Google account, Google group, G Suite domain account, or Cloud Identity domain account can be configured as a normal user in GKE Kubernetes clusters.

In contrast to normal users, Kubernetes service accounts are accounts managed by Kubernetes. Kubernetes service accounts provide an identity for processes in a Pod and are used to interact with the Kubernetes cluster. These are different from GCP Service Accounts. Those are managed by Google Cloud Identity, not Kubernetes, and you use them when you want GCP resources to have an identity that’s tied to an application or a virtual machine, not a human being. You can even configure GCP service accounts as a normal Kubernetes user, if that proves necessary. In Kubernetes, every namespace has a default Kubernetes service account. When processes in containers within a Pod communicate with the kube-APIserver, they identify themselves as a Kubernetes service account.

If you are security-sensitive, you might be worried about these Kubernetes service accounts. What if an attacker compromised a

container or Pod? Could they grab service account credentials and then use them in further attacks? Yes, they could. In Kubernetes 1.8 and higher, service accounts have no default permissions or access tokens. And GKE always lets you choose what version of Kubernetes runs in your cluster. So choose version 1.8 or higher, and you don’t have to worry about this potential problem.



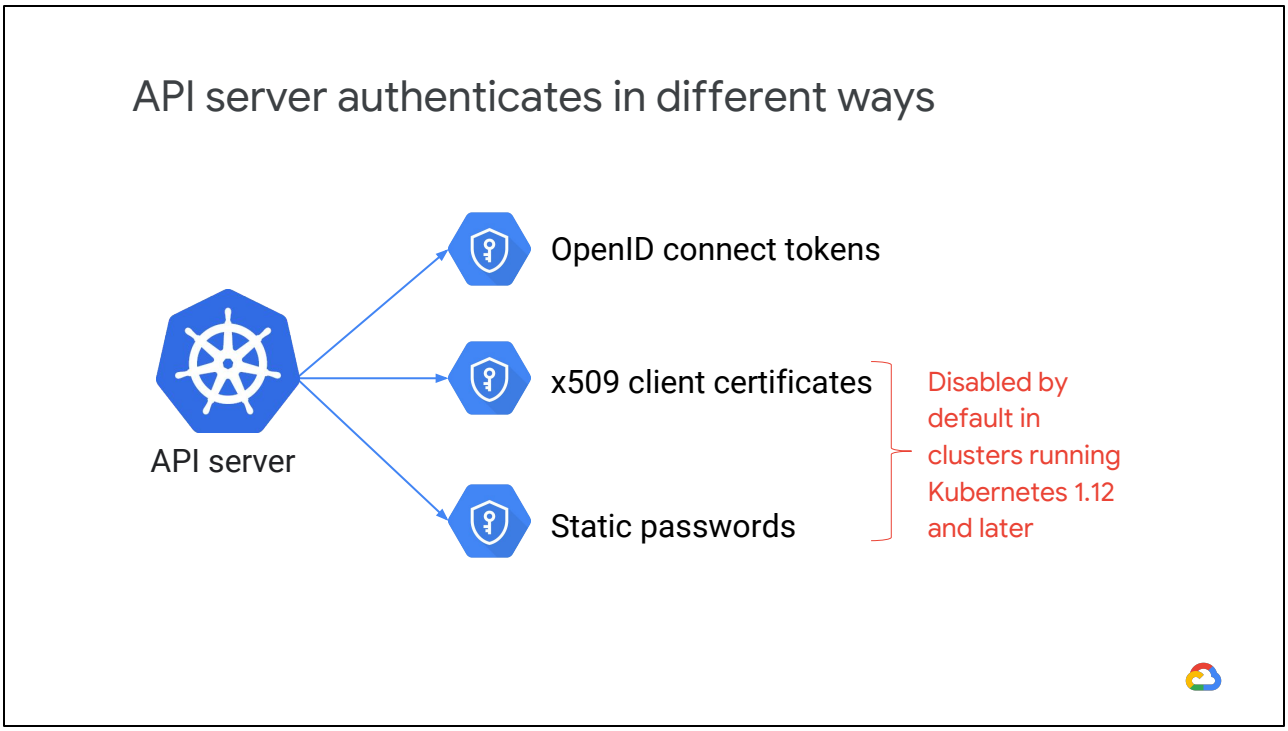
After an account is authenticated, for example a user authenticating via Cloud Identity, there are two main ways to authorize what that account can do. Cloud Identity & Access Management and Kubernetes role-based access control.

You need both levels. Cloud IAM is the access control system for managing GCP resources. It lets you grant permissions to users to perform operations at the project and cluster levels-- in other words, outside your Kubernetes clusters.

On the other hand, Kubernetes RBAC gives you access control inside your Kubernetes clusters, at the cluster level and the namespace level. It lets you create finely tuned roles with granular access to resources within the cluster.

So you'll use Cloud IAM to define who can view or change the

configurations of your GKE clusters. And you'll use Kubernetes RBAC to define who can view or change Kubernetes objects inside those clusters. With both these methods, it's a best practice to exercise the principle of least privilege. Each person who has access to your cluster should have the minimal set of permissions to do their job.



In Kubernetes, the API server listens for remote requests using HTTPS on port 443. Each request must be authenticated before it's acted upon. The API server can perform this authentication using various methods. GKE supports, as authentication methods: OpenID connect tokens, x509 client certificates and basic authentication using static passwords.

OpenID Connect is a simple identity layer on top of the OAuth 2.0 protocol. It lets clients verify the identity of an end-user by querying an authorization server. The clients can also get basic profile information about the end-user in an interoperable way.

x509 client certificates and static passwords present a wider surface of attack than OpenID. In Kubernetes versions earlier than 1.12, both of these access credentials are generated for you when you create a new cluster. Unless your application is using these

methods of authentication, you should disable them and use the OpenID authentication method instead.

In GKE, both X509 and static password authentication are disabled by default in clusters running Kubernetes 1.12 and later. That improves the default security configuration.

Agenda

Authentication and Authorization

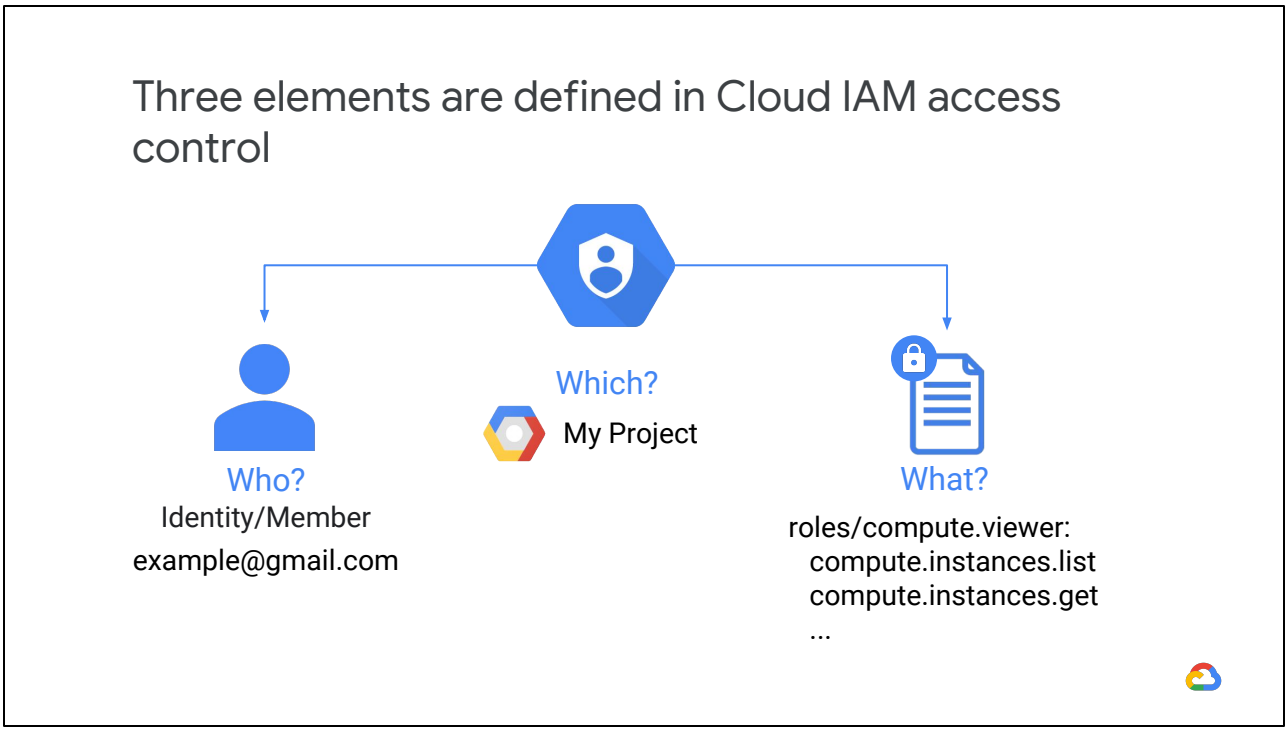
Cloud IAM

Kubernetes RBAC

Kubernetes Control Plane Security

Pod Security

Now let's take a look at Google Cloud IAM access control. In this lesson, you will learn about how Cloud IAM access control is configured. You'll also learn how you use Cloud IAM policies and roles to control who can interact with and manage your GKE clusters.



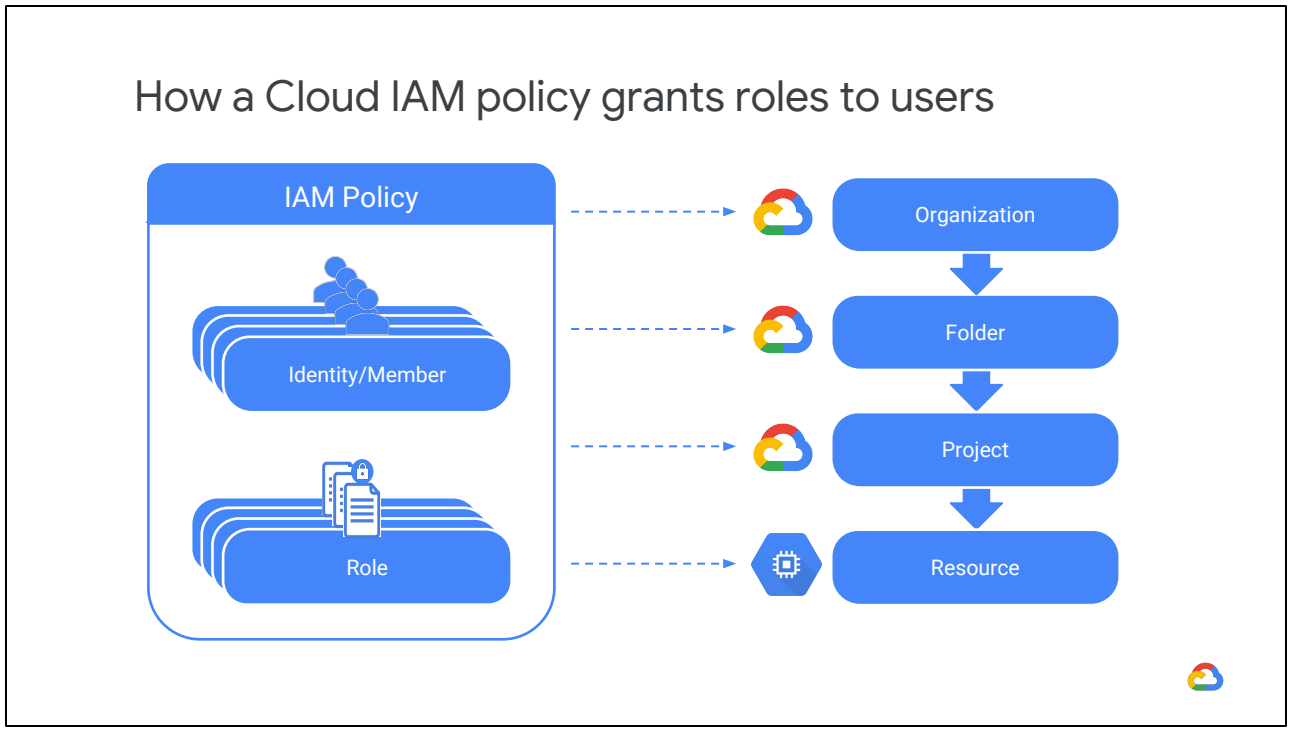
“Who” refers to the identity of the person making the request. “What” refers to the set of permissions that are granted. And “which” refers to which resources this policy applies to.

In Cloud IAM, you grant access to members. Cloud IAM will authenticate users against Cloud Identity. Don’t forget: Cloud Identity is the service that governs Google accounts, G Suite domain accounts and GCP Service Accounts. And don’t forget that you can gather these identities into groups, and groups themselves can have permissions. What if your company uses Microsoft® Active Directory®, or has an LDAP server? You can configure Google Cloud Directory Sync (GCDS) to synchronize data from kind of directory service into your Google domain. It’s a one-way sync operation. That means that your Google users, groups, and shared contacts are synchronized to match the information in your directory service. The data in your directory service isn’t modified.

You'll use Cloud IAM to grant permissions to members to let them perform certain operations on certain resources. GCP permissions are grouped into roles that make sense. Permissions can't be individually assigned to members. Instead, members are assigned roles. Every operation on a GCP resource is carried out through calls to Google Cloud APIs. Each API call is controlled using a permission. If the member trying to do the operation doesn't have the permission needed by the API call, it's denied. Here's an example. There's a role called "compute viewer," which lets someone who has it fetch various kinds of information about a Compute Engine resources. This role consists of many permissions, which have names like `compute.instances.get`. These names consist of an abbreviated name for a GCP service, a kind of GCP resource, and a verb. The `compute.instances.get` permission, which we will see again later in this module, controls whether someone can view the metadata of a Compute engine virtual machine.

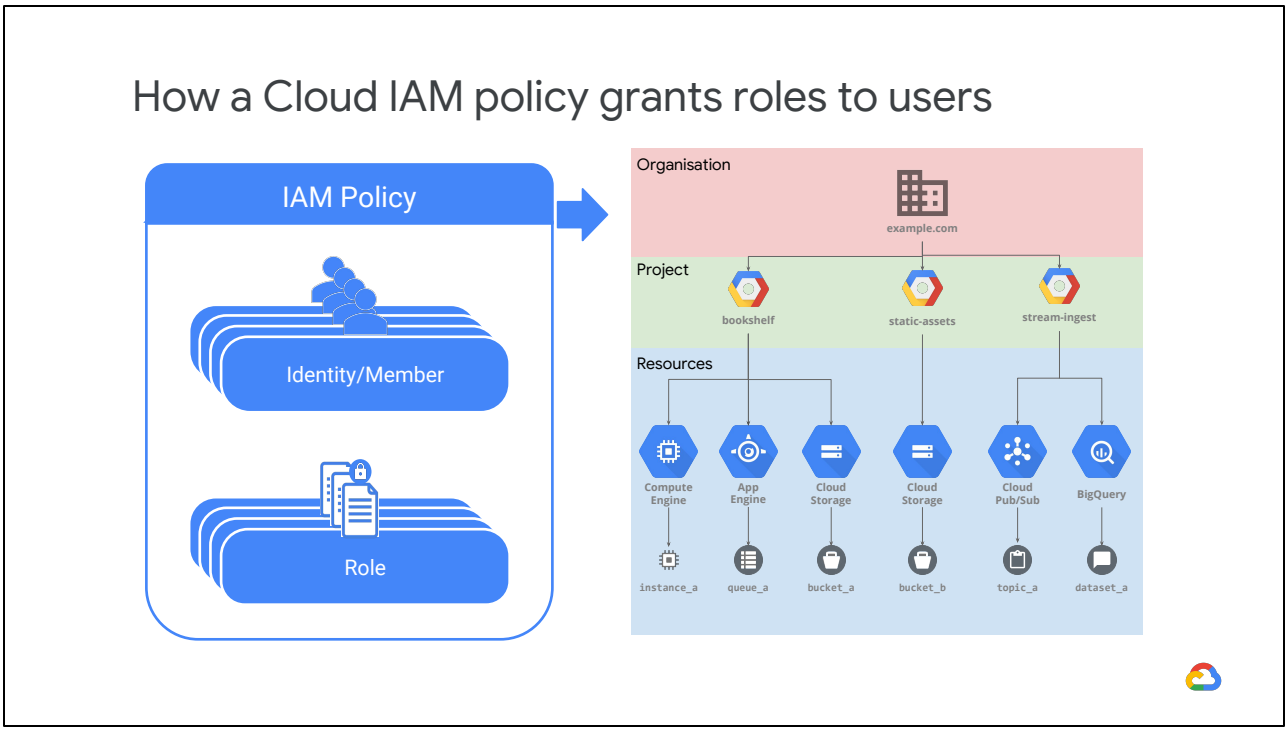
A GCP administrator could give this role to the member called example@gmail.com, at the level of a GCP project, which would let that member examine any Compute Engine resources in the project.

So, let's put the whole Cloud IAM story together. Whenever you make a request of GCP, Cloud IAM authenticates you, validates that you have the needed permissions for what you are trying to do and on the resource you're trying to do it to.

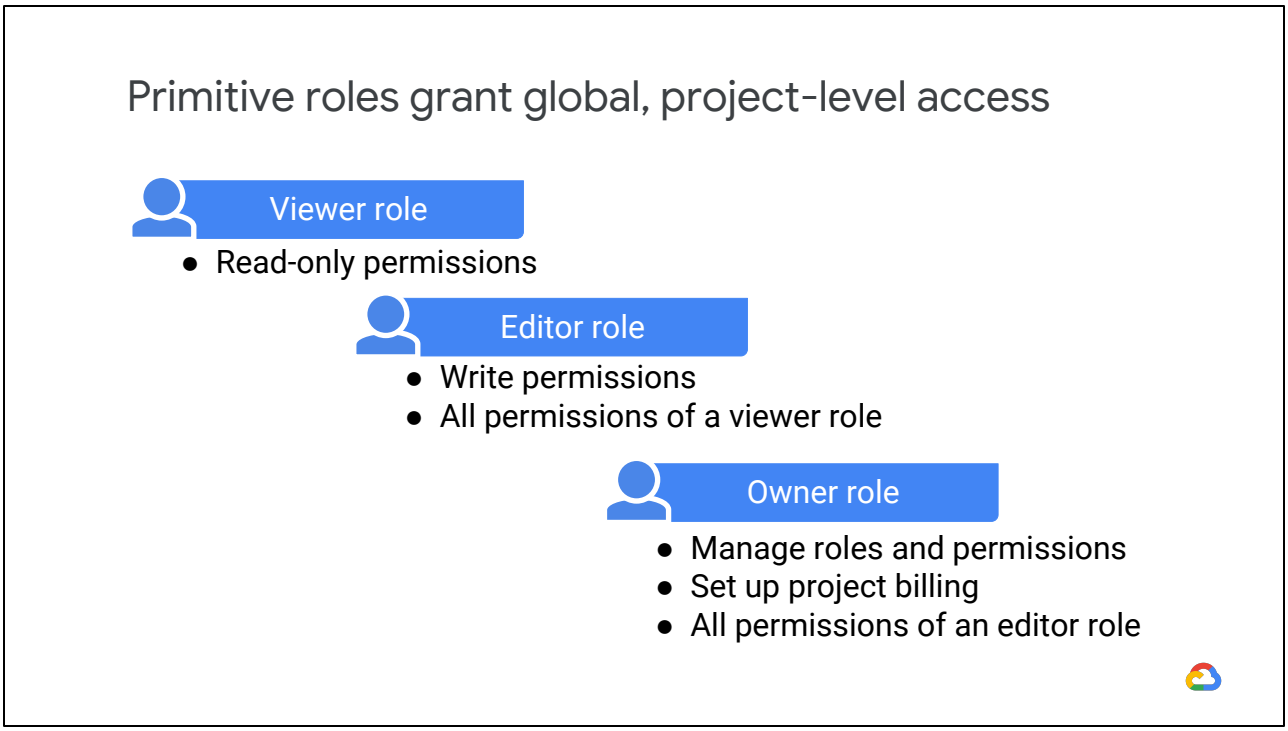


Let’s look at how a Cloud IAM policy grants roles to users.
A Cloud IAM policy is a list of bindings, and in each binding a set of members is bound to one or more roles.
The IAM policy, in turn, can be attached to a specific resource, a project, a project folder, or a whole organization.

GCP resources are organized hierarchically, starting with the organization. Underneath the organization, you can have multiple folders containing multiple projects, and so on. An IAM policy attached at the organization level will automatically have access to all folders, all projects, and ultimately all relevant resources. In this way, you can set up access control at any level within the organizational hierarchy and choose the most appropriate level for each IAM policy.



Cloud IAM policies applied at higher levels of a GCP organizational hierarchy are inherited by resources lower down that hierarchy. Policies applied at the organizational level apply to every project, and to all resources in those projects. Policies applied at the project folder level apply to all projects in that folder. Policies applies at the project level only apply to the resources that belong to that project and policies that are applied to resources within a project only apply to those specific resources.



There are three kinds of roles in Cloud IAM: Primitive, Predefined, and Custom.

Primitive roles are the first type of role. Primitive roles existed before the introduction of Cloud IAM but can still be used with Cloud IAM. These roles can be used to grant users global, project-level access to all GCP resources within a Project. There are three primitive roles.

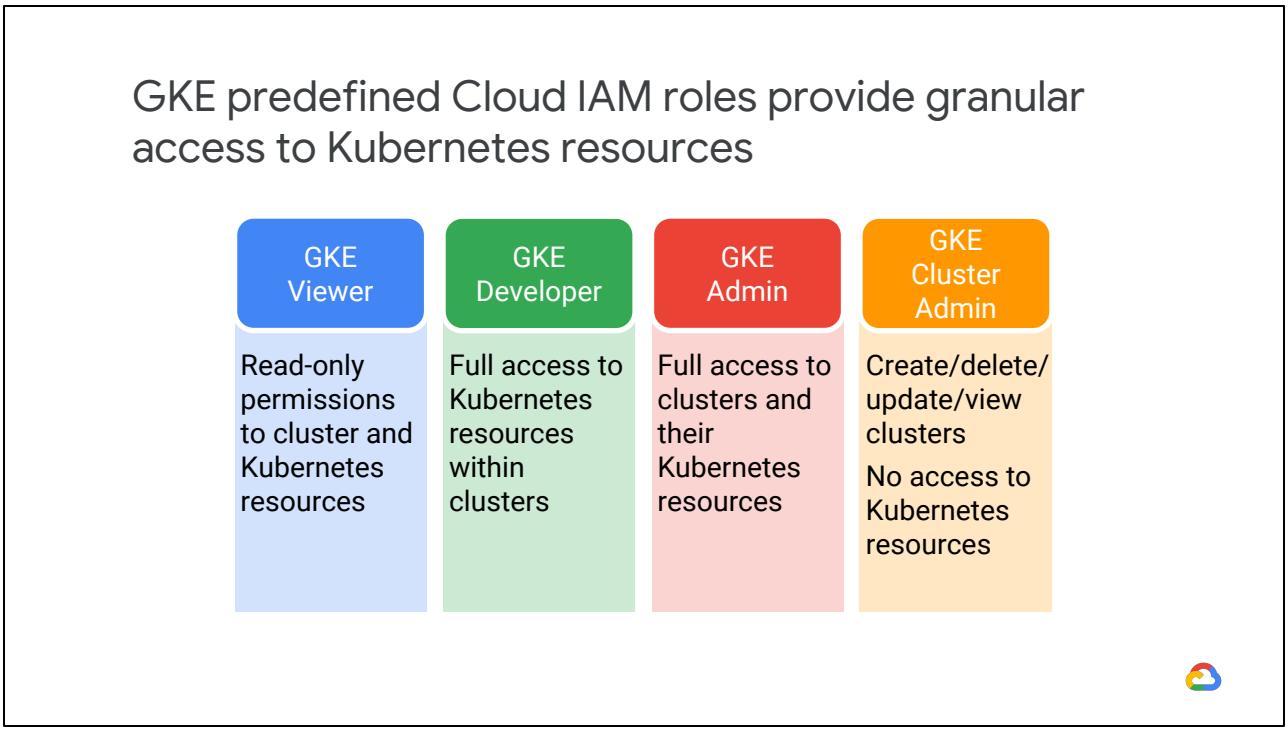
The first primitive role, the viewer role, permits read-only actions, such as viewing existing resources or data across a whole project.

In addition to all viewer permissions, the editor role is allowed to modify existing resources.

The final primitive role, the owner role, has all of the editor

permissions and also has the right to manage roles and permissions and set up billing for a project.
Any of these roles can access all resources such as App Engine, Compute Engine, and Cloud Storage.

Primitive roles are very broad. I've mentioned the principle of least privilege: the idea that users should have the minimum set of permissions needed to do their jobs. If you are using primitive roles, you probably aren't adhering to this principle, and you are exposing yourself to risks of operator error, and maybe worse.



Predefined roles are the second type of Cloud IAM role. GKE provides several predefined Cloud IAM roles that provide granular access to Kubernetes Engine resources.

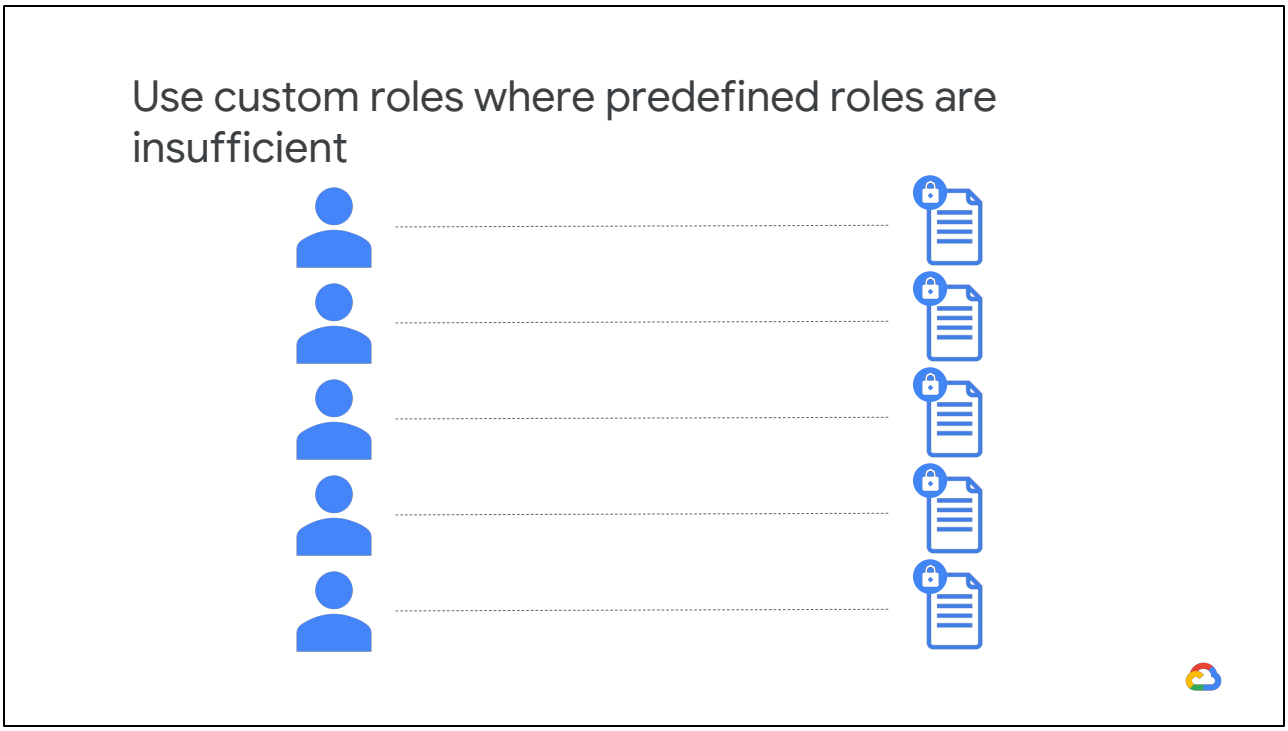
The GKE Viewer role gives read-only access, as might be needed for auditing.

The GKE Developer role is suited to developers and release engineers. It grants full control to all resources within a cluster.

The GKE Cluster Admin role is used to create, delete, update, and view clusters, but provides no access to Kubernetes resources.

The GKE Admin role gives full access to clusters and the Kubernetes Engine resources inside the clusters. Example users are project owners, system administrators, and on-call engineers.

The GKE Host Service Agent User is another role intended for use by service accounts and is used for network management in a shared VPC environment.



Finally, the third type of role. If the primitive roles and the GKE predefined roles are too permissive, or the assumptions don't fit your business case, you can craft custom roles with more granular control. To build a custom role, you specify each individual permission that will make up the role. For example, suppose you needed to create a specific user account that will be used solely to manage the software running inside a certain GKE cluster, but who should not have any access to view GCP resources, not even to view them. You can create a custom IAM role that just has the permissions needed to authenticate to the cluster and nothing else. In the next section, we'll discuss how you can give permissions within Kubernetes to such a user.

Agenda

Authentication and Authorization

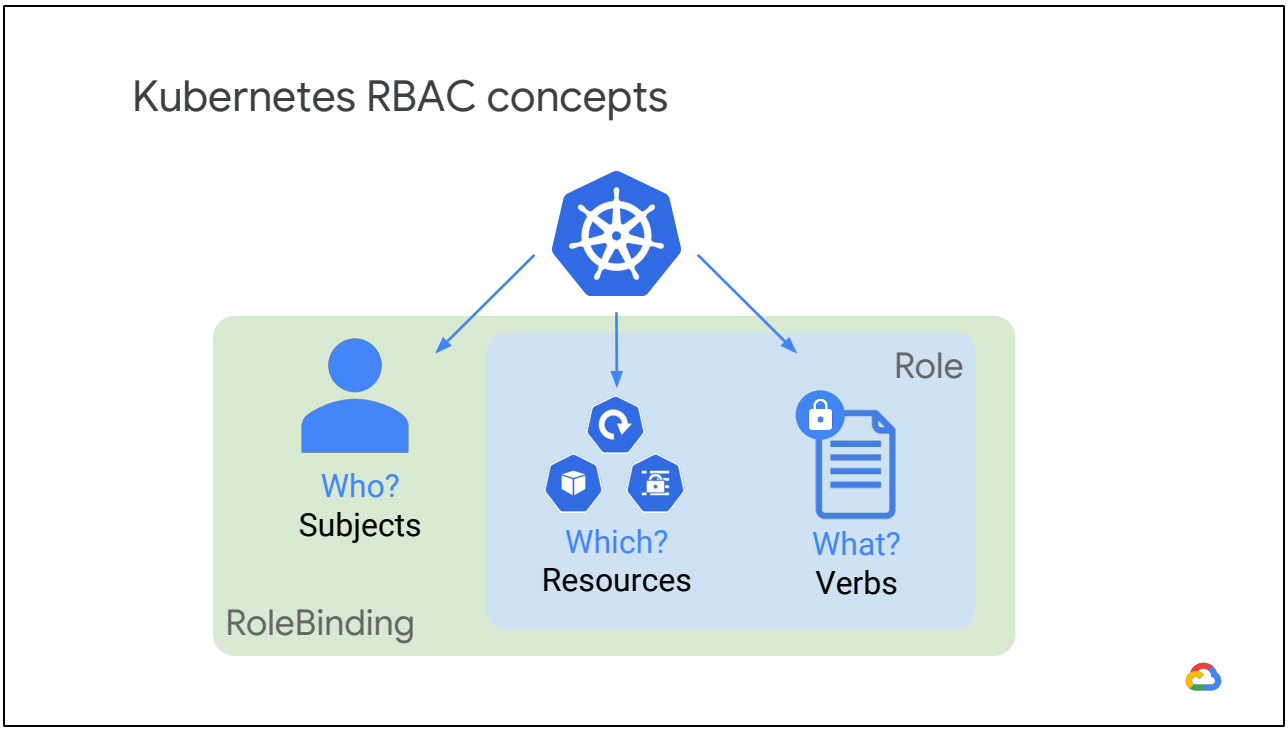
Cloud IAM

Kubernetes RBAC

Kubernetes Control Plane Security

Pod Security

Next, let's look at the next level of access control. In this lesson you will learn about Kubernetes role-based access control (RBAC). RBAC is a native Kubernetes security feature that provides you with fine grained tools to manage user account permissions. In GKE environments, RBAC extends Cloud IAM security by offering control over Kubernetes resources within the cluster supplementing the control provided directly by Cloud IAM which allowed you control access at the GKE and cluster level.

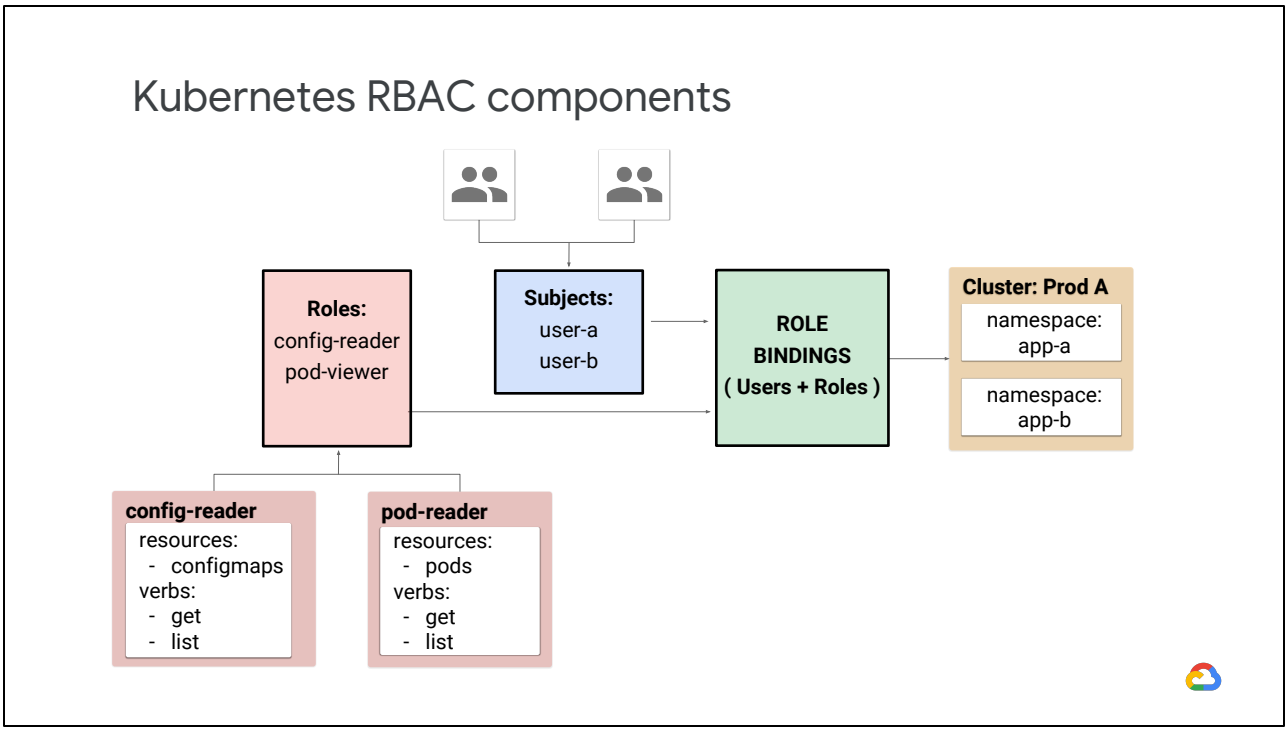


There are three main elements to Kubernetes role-based access control: subjects, resources, and verbs.
With Kubernetes RBAC you define what operations (verbs) can be executed over which objects (resources) by who (subjects).
A subject is a set of users or processes that can make requests to the Kubernetes API.
A resource is a set of Kubernetes API objects, such as Pods, Deployments, Services, or PersistentVolumes.
A verb is a set of operations that can be performed on resources, such as get, watch, create, and describe.

These three elements can be connected by creating two types of RBAC API objects: Roles and RoleBindings. Roles connect API resources and verbs.

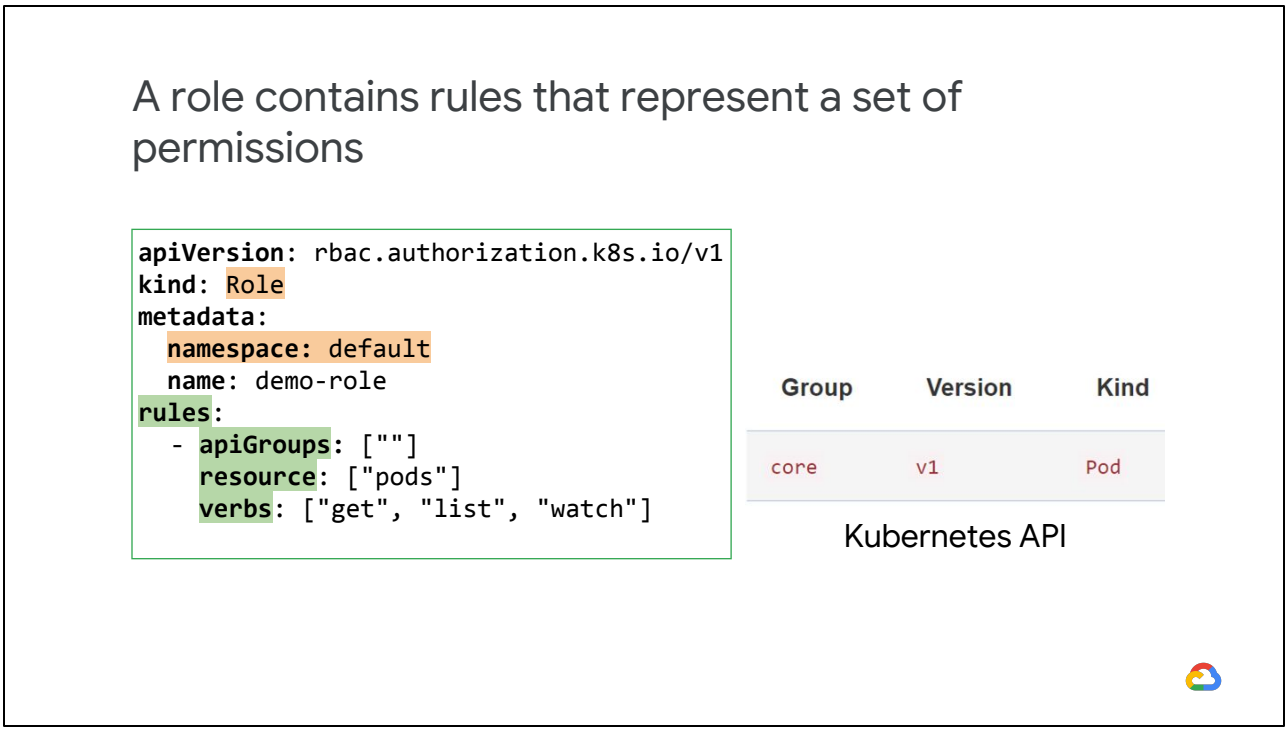
RoleBindings connect roles to subjects. Roles and RoleBindings

are applied at the cluster level or namespace level.



Kubernetes RBAC is helpful if you need to grant users access to specific Kubernetes namespaces. You add users as subjects, configure the roles with the proper permissions, and then bind them to the objects—in this case, the namespaces in the cluster. In Kubernetes, there are two types of roles: Role and ClusterRole. RBAC roles are defined at the namespace level, and RBAC ClusterRoles are defined at the cluster level.

Image Source:
https://cloud.google.com/solutions/prep-kubernetes-engine-for-prod



This example manifest defines an RBAC Role at the namespace level.
The Role type is defined and in this example the default namespace is specified. You can only define a single namespace for a Role.

Under rules, apiGroups in this case is defined as empty to indicate the role applies to the core api group, resource is specified as Pods, and the verbs *get, list,* and *watch* are allocated. If you put these three verbs in a role like this, that grants subjects who have the role to inspect the state of pods. It's common practice to always allocate *get*, *list*, and *watch* together. The apiGroups can be empty, indicating the core API group or it can specify specific Kubernetes API groups such as batch, extensions or apps.

Rules are purely additive; there are no "deny" rules. If no rule

grants a user a verb on a given resource, that user can’t do the action.

A ClusterRole grants permissions at the cluster level

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
   name: demo-clusterrole
rules:
  - apiGroups: [""]
    resource: ["storageclasses"]
    verbs: ["get", "list", "watch"]
```

Here, a ClusterRole is defined. A ClusterRole grants permissions at the cluster level; so there's no need to specify a namespace. ApiGroups is defined in the same way as it is defined when creating a Role. Resources can be cluster-scope, such as nodes, Secrets, and PersistentVolumes, or they can be namespaced-resources such as Pods, Deployments, and StatefulSets.

Examples of how to refer to different resource types

```
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
```

```
rules:
- apiGroups: [""]
  resources: ["pods"]
  resourceNames: ["demo-pod"]
  verbs: ["patch", "update"]
```

```
rules:
- apiGroups: [""]
  resources: ["pods", "services"]
  verbs: ["get", "list", "watch"]
```

```
rules:
- nonResourceURLs: ["metrics/",
"/metrics/*"]
  verbs: ["get", "post"]
```

Here are some examples of rule sets that specify the resources and verbs covered by a Role or ClusterRole.

On the top left is a basic assignment that specifies *get, list* and *watch* operations on all pod resources. On the bottom left, a subresource ‘logs’ is added to the resources list to specify access to Pod logs also. The combination of verbs, *get, list* and *watch,* are the standard read-only verbs. You should typically assign them as a unit, all or none.

On the top right, a resourceName is specified to limit the scope to specific instances of a resource and the verbs specified as *patch* and *update*. These are also a group of verbs that are typically treated as a unit and applied together.

Note that you can only use verbs such as *get, update, delete*, and

patch on named resources.

And bottom right shows how you can specify *get* and *post* actions for non-resource endpoints. This form of rule is unique to ClusterRoles.

Attaching roles to RoleBindings

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: default
  name: demo-rolebinding
subjects
- kind: User
  name: "joe@example.com"
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: demo-role
  apiGroup: rbac.authorization.k8s.io
```

Now that ClusterRoles have been defined, you can attach RoleBindings and ClusterRoleBindings to those roles. Let's attach a RoleBinding first. Define the namespace then the subjects. In Kubernetes, subject kind can be users, groups, or service accounts. Note that subject name is case-sensitive.

With RBAC in GKE, the type of account that you can control access for is either a Google Account, a GCP service account, or a Kubernetes service account, and they are identified using an email address.

You bind a role to subjects in the RoleBinding using roleRef by defining the role kind and the role name. A RoleBinding can refer to roles or ClusterRoles. If a ClusterRole is bound in a RoleBinding, it only grants permissions to the resources of the particular namespaces defined for the RoleBinding, and not to the entire

cluster.

You can identify namespaced / non namespaced resources using the command kubectl api-resources --namespaced=true

ClusterRoleBinding has a cluster-wide scope

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: demo-clusterrolebinding
subjects
- kind: User
  name: "admin@example.com"
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: demo-clusterrole
  apiGroup: rbac.authorization.k8s.io
```

Here's an example of a ClusterRoleBinding. Note that namespace isn't mentioned, because a ClusterRoleBinding always has a cluster-wide scope. Note that a ClusterRoleBinding can only refer to a ClusterRole, not to a role.

Examples of how to refer to different subject types

```
subjects
- kind: User
  name: "joe@demo.com"
  apiGroup: rbac.authorization...
```

```
subjects
- kind: Group
  name: system.serviceaccounts
  apiGroup: rbac.authorization...
```

```
subjects
- kind: Group
  name: "Developers"
  apiGroup: rbac.authorization...
```

```
subjects
- kind: Group
  name: system.serviceaccounts:qa
  apiGroup: rbac.authorization...
```

```
subjects
- kind: ServiceAccount
  name: default
  namespace: kube-system
```

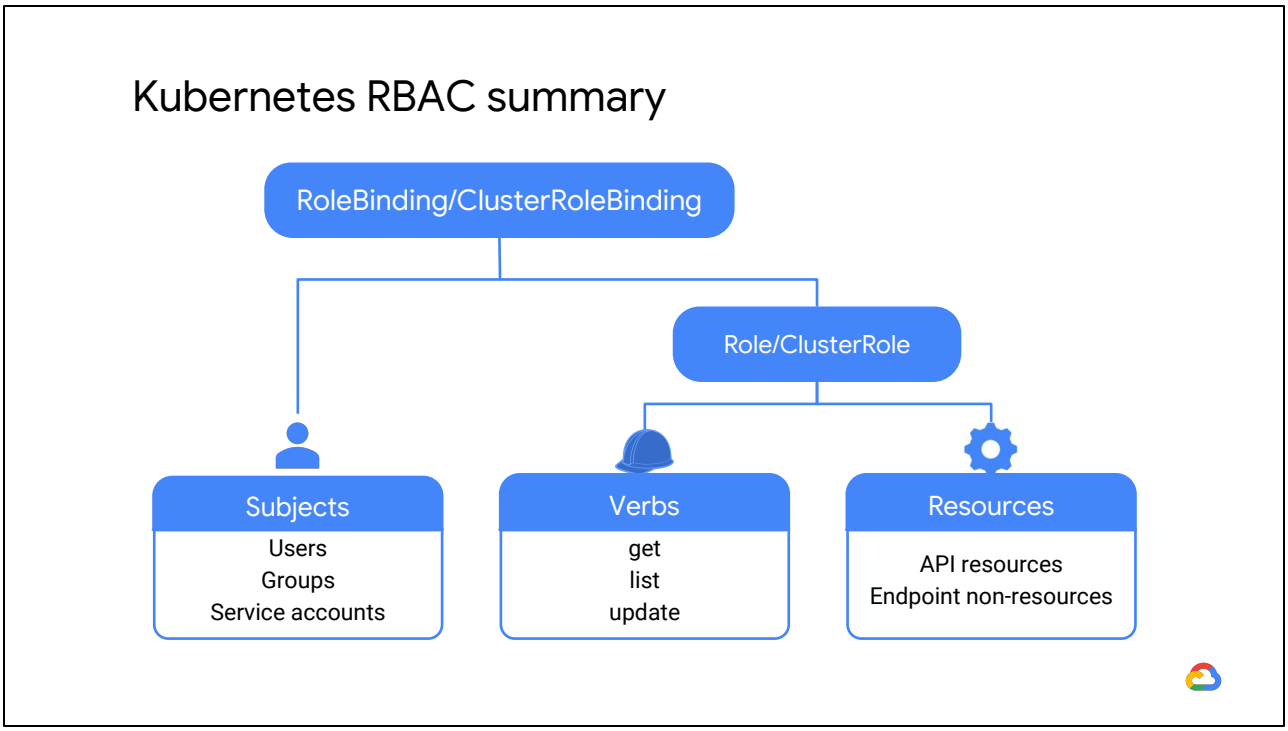
```
subjects
- kind: Group
  name: system.authenticated
  apiGroup: rbac.authorization...
```

Here are some examples of how to refer to different subject types including users, service accounts and groups.

Working from the top left, you see examples specifying a user and a group. On the bottom left you have an example of a ServiceAccount subject where the ServiceAccount name is ‘default’ and the namespace is specified as ‘kube-system.’

Assigning Kubernetes RBAC permissions to Google Groups is a beta feature that is not currently supported and is not enabled by default. You can assign Cloud IAM permissions to a Google Group. However, Kubernetes can support groups provided the authentication provider is compatible, and you can assign Kubernetes RBAC permissions to Kubernetes groups if a supported identity provider is configured. In this example, you can see that there is a group available called “Developers”.

In the right column you have more group types. The example includes all service accounts on all namespaces. Note that this is a system group, so it can be used for RBAC in GKE environments. The middle example on the right includes only the service accounts in the qa namespace, and the example on the bottom right includes all authenticated users.



With Kubernetes RBAC, you can manage granular permissions for the people (using users and groups), and for containers (using service accounts) at both the namespace level and the cluster level. Resources and verbs are bound using either Roles or ClusterRoles. Roles and ClusterRoles are then bound to subjects using RoleBinding or ClusterRoleBinding.

A variety of ClusterRoles and ClusterRoleBindings are also predefined within the RBAC system. The Kubernetes documentation offers more information.

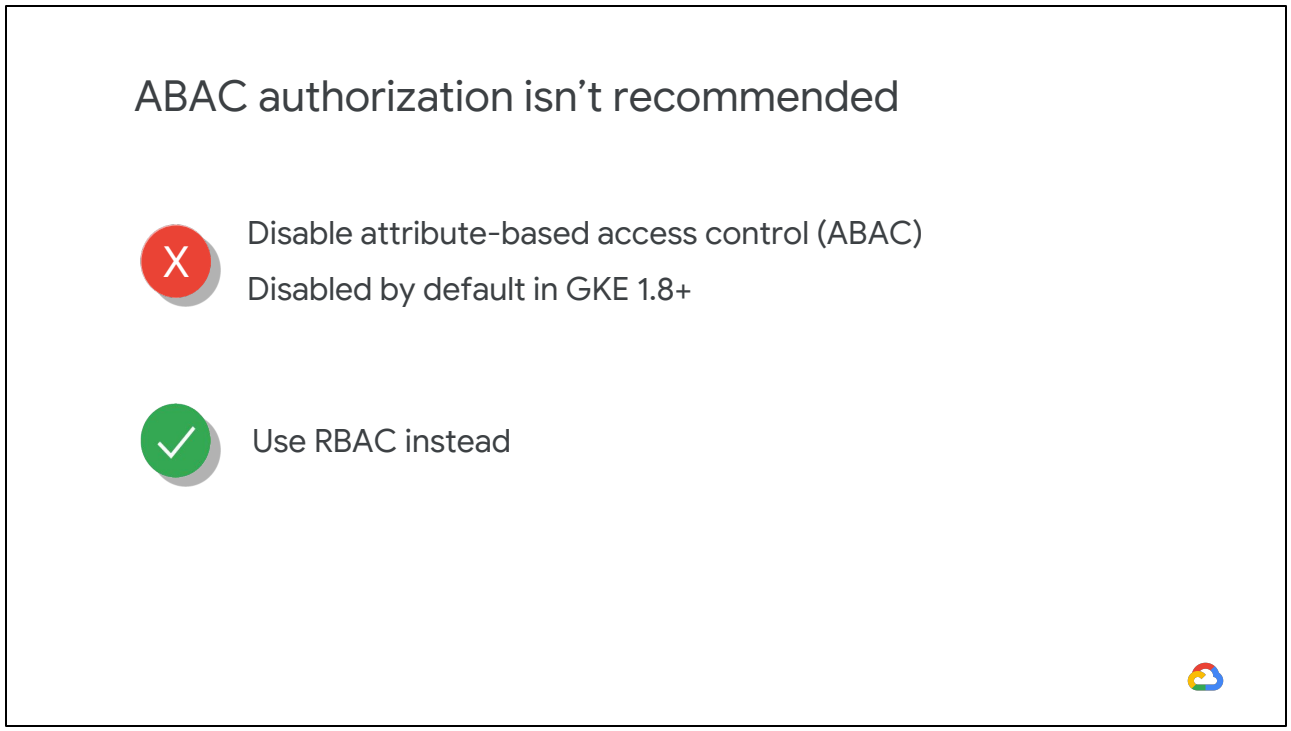
Not all resources are NameSpaced

```
$ kubectl api-resources
```

```
NAME                ... NAMESPACED
bindings                true
componentstatuses       false
configmaps              true
endpoints               true
...
namespaces              false
nodes                   false
...
```

When you create Roles and ClusterRoles, you should take into account whether a resource is associated with a namespace or defined at the cluster level. You can use the kubectl api-resources command to list which resources are “NameSpaced,” as the command’s output phrases it, and which are not.

Typically, cluster level resources should be managed with ClusterRoles, and NameSpaced resources should be managed using Roles. However, if you need to define RBAC permissions across multiple NameSpaces then you will need to use a ClusterRole.



There is another, older, authorization option in Kubernetes — attribute-based access control, or ABAC. RBAC is simpler and easier to work with than ABAC. Google recommends that you use RBAC and disable ABAC. ABAC is disabled by default in GKE clusters running Kubernetes 1.8 and later versions. If you upgrade a cluster from an earlier Kubernetes version, you will have to disable ABAC yourself.

Agenda

Authentication and Authorization

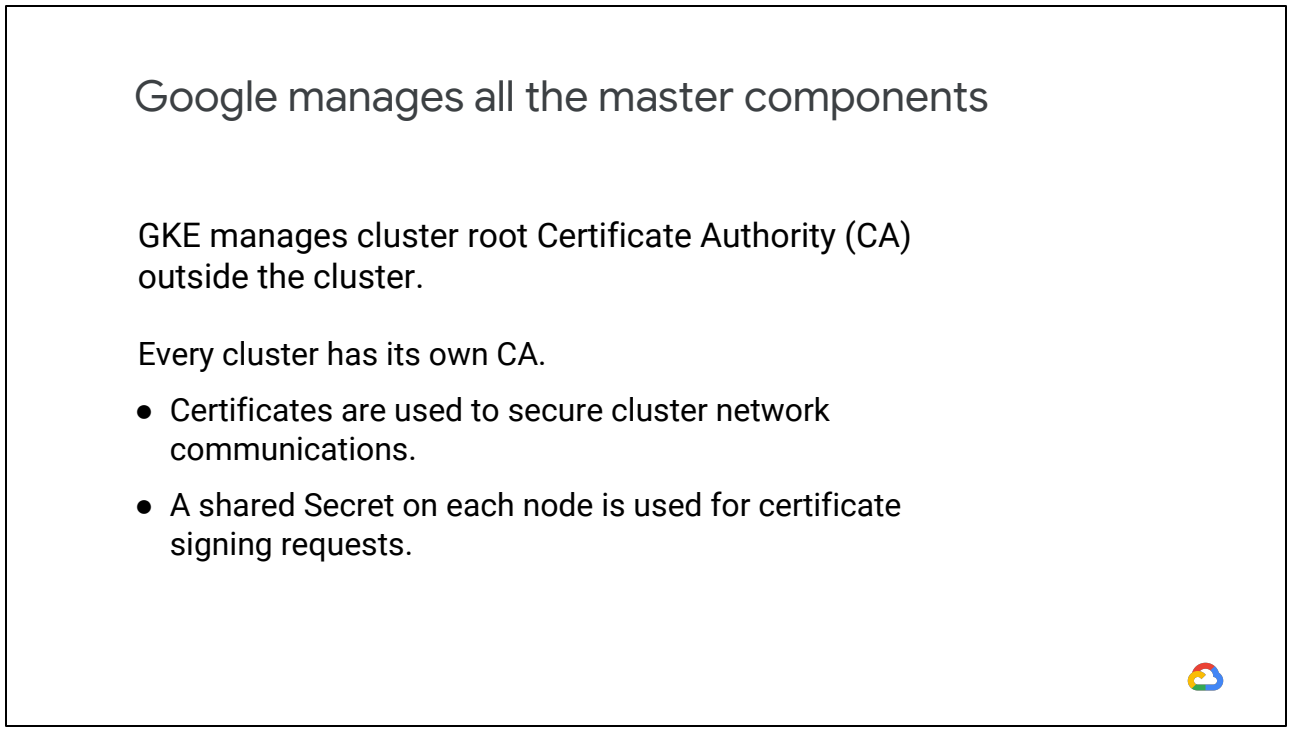
Cloud IAM

Kubernetes RBAC

Kubernetes Control Plane Security

Pod Security

Next, let’s learn more about Kubernetes Control Plane Security.



In GKE, all the master components, such as the API server, etcd database, and controller manager, are managed by Google. Each cluster has its own root certificate authority (“CA” for short). An internal Google service manages the root keys for this CA.

Secure communications between the master and nodes in a cluster relies on the shared root of trust provided by the certificates issued by this CA.

GKE uses a separate per-cluster CA to provide certificates for the etcd databases within a cluster. You don’t have to do anything to turn on this feature; it’s automatic. It’s beneficial because it means there doesn’t have to be shared trust between the services running in the cluster and the databases that keep the cluster working.

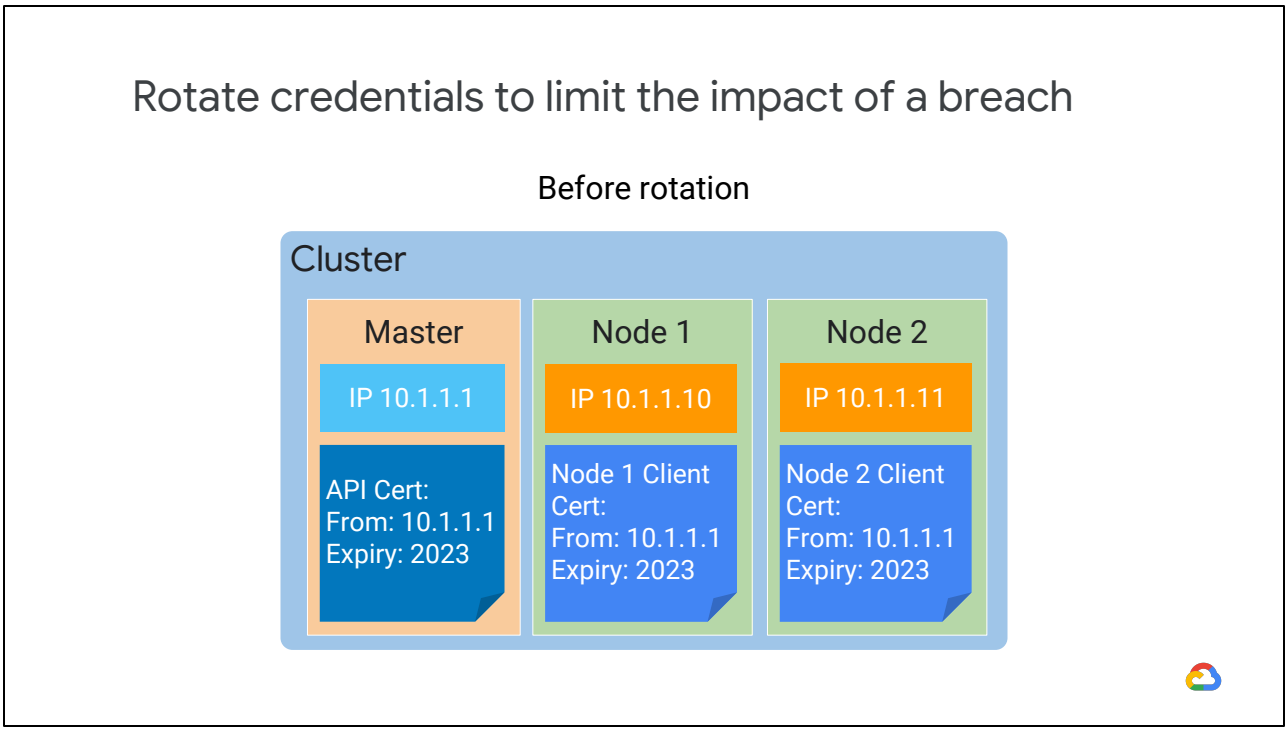
Remember kubelets, which are the primary Kubernetes agents on

the nodes? The Kubernetes API server and the kubelets use secure network communications protocols -- namely TLS and SSH -- when they communicate with each other. They use certificates issued by the cluster's root CA to support those protocols.

Because separate CAs are used for each separate cluster, a compromised CA of one cluster cannot be used to compromise other clusters.

When a new node of a Kubernetes cluster is created, the node is injected with a shared Secret as part of its creation. This Secret is then used by its kubelet to submit certificate signing requests to the cluster root CA.

That way, it can get client certificates when the node is created, and new certificates when they need to be renewed or rotated. And finally, Kubelets use these client certificates to communicate securely with the API server. Note that this Secret can be accessed by Pods, and by extension their containers, unless metadata concealment is enabled.

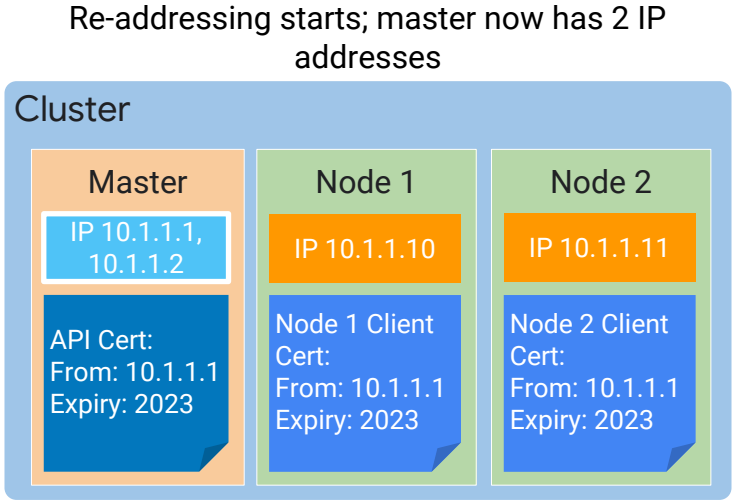


It's essential that these credentials are rotated periodically to improve the security posture of your cluster and limit the impact of potential security breaches that compromise credentials.

How often should you rotate credentials? It's a tradeoff: If a cluster manages high-value assets, you probably want to rotate credentials often. But whenever you do, the cluster's Kube-APIserver is unavailable for a short time. So you need to make the tradeoff that works for your organization's security policies.

You'll learn how to manually rotate the credentials used by the API server and clients during a lab activity. You can't manually rotate etcd certificates in GKE, though, because they're managed entirely by Google.

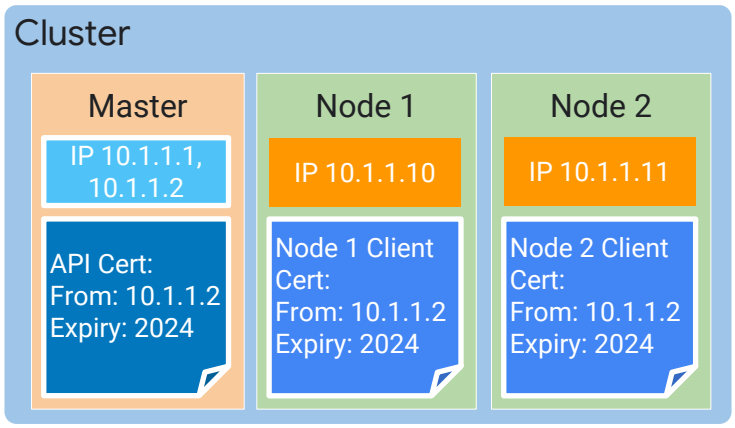
Rotate credentials to limit the impact of a breach



Let's look at an overview of the rotation process. The process starts by creating a new IP address for the cluster master that will be available along with its existing IP address. New certificate credentials are issued to the control plane with this new address. Note that the API server will not be available during this period, although Pods continue to run.

Rotate credentials to limit the impact of a breach

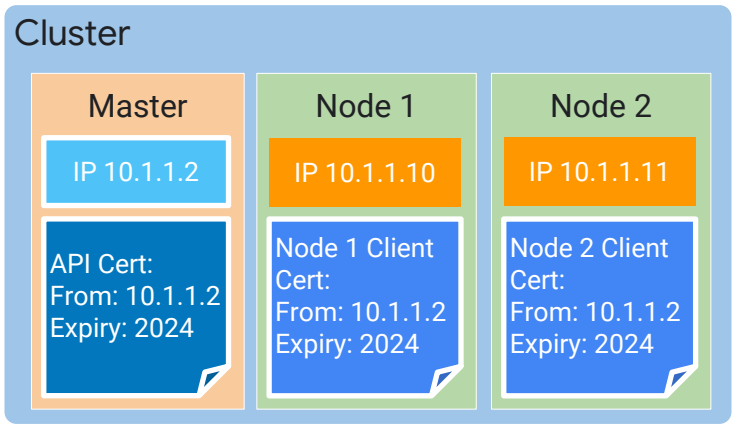
Control plane credentials are updated to use the new IP



After the master is reconfigured, the nodes are automatically updated by GKE to use the new IP and credentials. This causes GKE to also automatically upgrade the node version to the closest supported version. All of your API clients outside the cluster must also be updated to use the new credentials.

Rotate credentials to limit the impact of a breach

Old master IP address is retired



Address rotation must be completed for the cluster master to start serving with the new IP address and new credentials and remove the old IP address and old credentials. If the rotation is not completed manually, GKE will automatically complete the rotation after seven days.

You can also rotate the IP address for your cluster

Initiating credential rotation

```
gcloud container clusters update [NAME] \
    --start-credential-rotation
```

Completing credential rotation

```
gcloud container clusters update [NAME] \
    --complete-credential-rotation
```

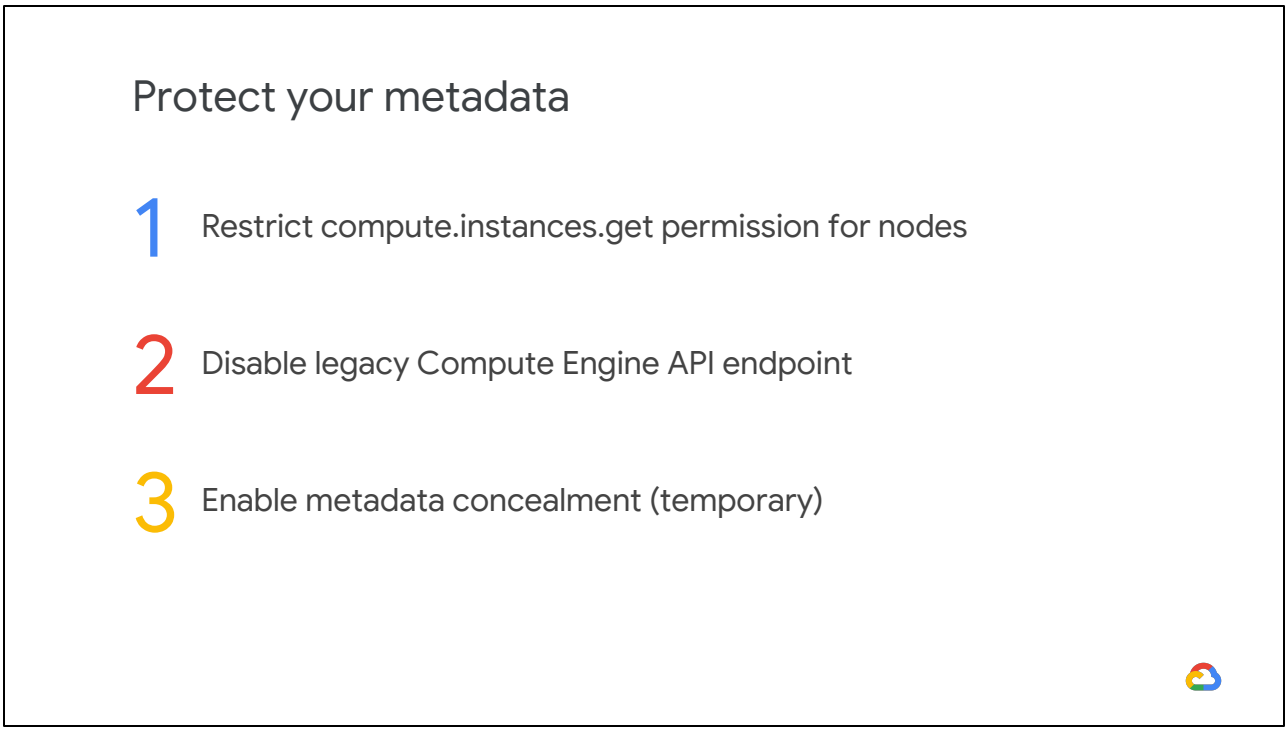
Initiating IP rotation

```
gcloud container clusters update [NAME] \
    --start-ip-rotation
```

Completing IP rotation

```
gcloud container clusters update [NAME] \
    --complete-ip-rotation
```

Note that you can also rotate the IP address for your cluster. This essentially goes through the same process, because the certificates must be renewed when the master ip-address is changed, but with different commands as shown here. In these examples, NAME refers to the cluster name.



Compromised applications running on Pods can potentially access the metadata such as the Node Secret that is used for Node configuration. To prevent such exposure, there are a few steps that you can take.

First, you should always configure the Cloud IAM service account for the node with minimal permissions. Don't confuse this Google service account with the Kubernetes service account; this is the Cloud IAM Service Account used by the Node VM itself. The idea here is to not use the compute.instances.get permission through a service account, Compute Instance admin role, or any custom roles. With permissions set in this way, the metadata on the GKE nodes cannot be simply retrieved by making direct Compute Engine API calls to these nodes.

The second step you should take is to disable legacy metadata

APIs. Compute Engine API endpoints using /0.1/ and /v1beta1/ support querying of metadata. The /v1/ APIs restrict the retrieval of metadata. Starting from GKE version 1.12, the legacy Compute Engine metadata endpoints are disabled by default. With earlier versions, they can only be disabled by creating a new cluster or adding a new node pool to an existing cluster.

One further important security step is to enable metadata concealment. This is basically a firewall that prevents Pods from accessing a node's metadata. It does this by restricting access to kube-env (which contains kubelet credentials) and the virtual machine's instance identity token. Note that this is a temporary solution that will be deprecated as better security improvements are developed in the future. There's a link associated with this lesson to Google Cloud's documentation on protecting cluster metadata.

Agenda

Authentication and Authorization

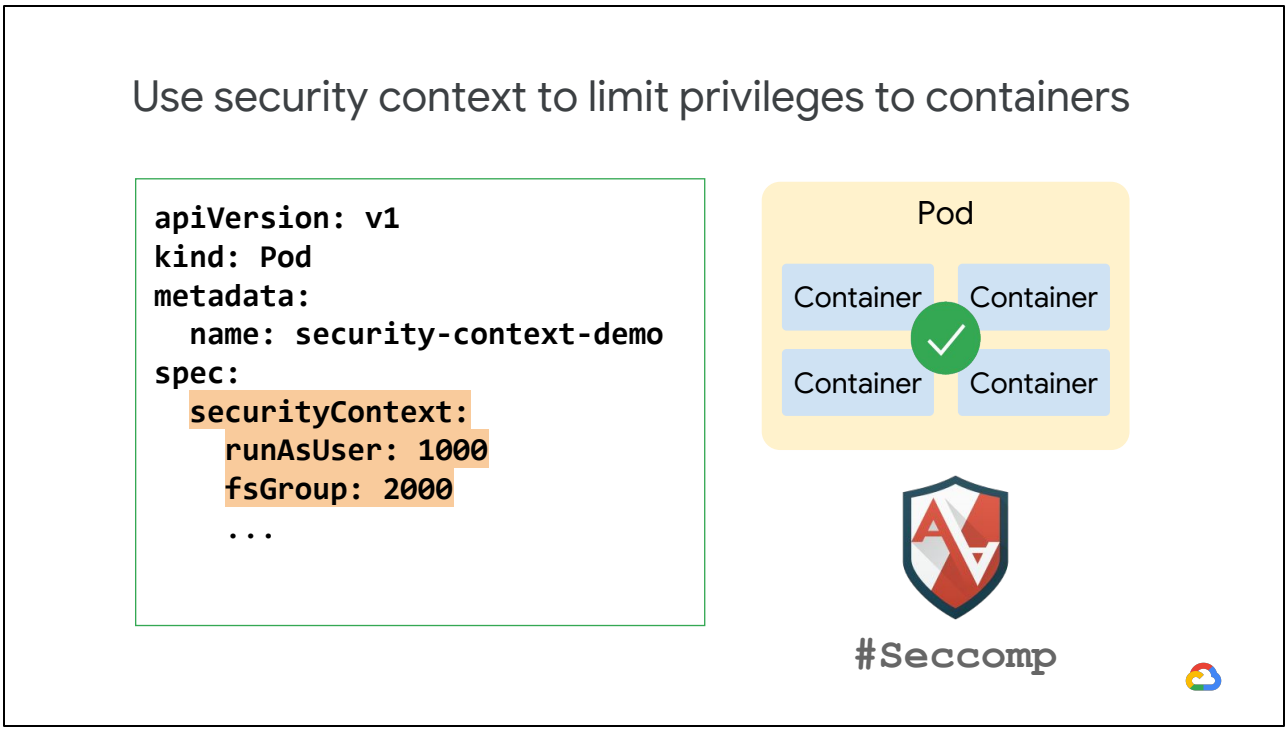
Cloud IAM

Kubernetes RBAC

Kubernetes Control Plane Security

Pod Security

Finally, let's discuss Pod security. In this lesson, you will learn how to control what sort of things a Pod, or more specifically the containers inside a Pod, are allowed to do. You will learn about the best practices you should adhere to in order to increase the security of your GKE environment.



By default, users can deploy containers inside a Pod that will allow privilege elevation and can access the host file system and the host network. Sometimes those capabilities are convenient, and sometimes they're undesirable from a security perspective. You can define restrictions on what the containers in a Pod can do by using security contexts. You can also enforce the use of specific security measures.

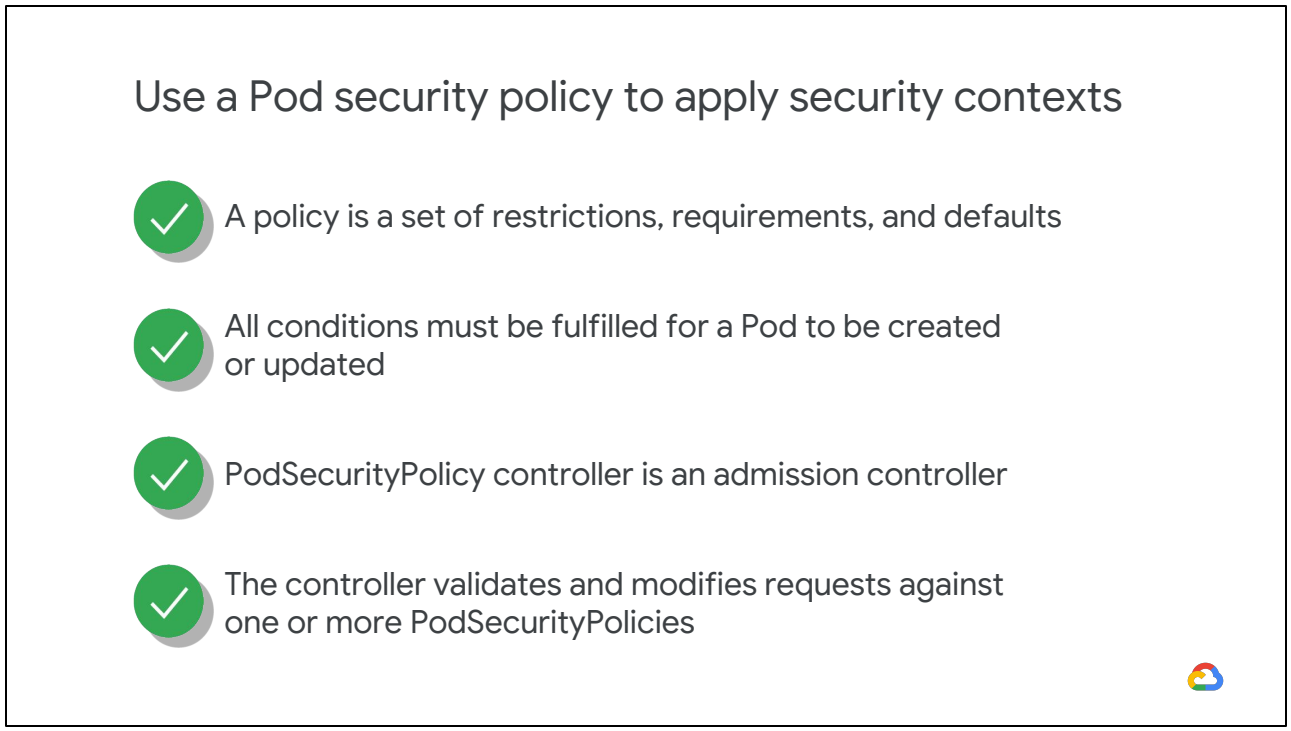
A security context is a set of security settings defined in a Pod specification. Here's an extract of a Pod manifest YAML. The securityContext definition inside the Pod specifies that the first process runs with user ID 1000 for any containers in the Pod, and group ID 2000 is associated with all containers in the Pod. This provides a specific user and group context for the containers. The most important thing about this context is that 1000 is not 0. In a Linux system, 0 is the privileged root user's user ID. Taking away

root privilege from the code running inside the container decreases what it can do if it is compromised.

If you define a security context at the Pod level, it is applied to all of the Pod's containers.

Using security contexts in a Pod definition, you can exercise a lot of control: over the use of the host namespace, over networking, over filesystem and Volume types, over whether privileged containers can run, and whether code in a container can escalate to root privileges.

You can also control other security settings. For example, you can enable Seccomp to block code running in containers from making system calls you know they should not legitimately make. You can enable AppArmor, which uses security profiles to restrict individual programs' actions. You can also limit access to some Linux capabilities, by granting certain but not all privileges to a process.



What if you have dozens or hundreds of Pods? Directly configuring security contexts in each individual Pod can become a lot of work. It's easier to define and manage security configurations separately, and then apply them to the Pods that need them. By defining Pod Security Policies, you create reusable security contexts. You can apply Pod Security Policies to multiple Pods without having to specify and manage all those details in each Pod definition.

What's in a Pod Security Policy? Each consists of an object and an admission controller.

The Pod security policy object is a set of restrictions, requirements, and defaults that are defined in the same way as the security context inside a Pod and can be used to control the same security features.

For a Pod to be created or updated, it must fulfil all of the security conditions that are defined in the Pod Security Policy. These rules are only applied when a Pod is being created or updated.

The PodSecurityPolicy controller is an admission controller that validates or modifies requests to create or update Pods against one or more PodSecurityPolicies.

You saw earlier in this module that a request to perform a specific action has to be authenticated and then authorized. But there's an extra step called *admission control*. A validating or non-mutating Admission controller just validates requests. A mutating Admission controller can modify requests if necessary and can also validate requests. A request can be passed through multiple controllers, and if the request fails at any point, the entire request is rejected immediately and the end user is notified through an error.

The Pod Security Policy admission controller acts on the creation and modification of pods and determines whether the Pod should be admitted based on the requested security context and the available Pod Security Policies. Note that these policies are enforced during the creation or update of a Pod, but a security context is enforced by the container runtime.

Pod security policy example

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: demo-psp
spec:
  privileged: false
  allowPrivilegeEscalation: false
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'MustRunAsNonRoot'
  seLinux:
    rule: 'RunAsAny'
  readOnlyRootFilesystem: false
```

This is an example of a PodSecurityPolicy. You'll learn how to create a PodSecurityPolicy in the lab.

Authorize your Pod security policy

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: psp-clusterrole
rules:
- apiGroups:
  - policy
  resources:
  - podsecuritypolicies
  resourceNames:
  - demo-psp
  verbs:
  - use
```

After you've defined a PodSecurityPolicy, you need to authorize it; otherwise it will prevent any Pods from being created.

You can authorize a policy using Kubernetes role-based access control. Here, a ClusterRole allows the PodSecurityPolicy called *demo-psp* to be used.

Define a RoleBinding to bind the ClusterRole to users or groups

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: psp-rolebinding
  namespace: demo
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: psp-clusterrole
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:serviceaccounts
```

```
- kind: ServiceAccount
  name: service@example.com
  namespace: demo
```

Next, define a RoleBinding to bind the previous ClusterRole to users or groups.

In this example, two subjects for the RoleBinding are specified. The first is a group containing all service accounts within the demo namespace. The other is a specific service account in the demo namespace.

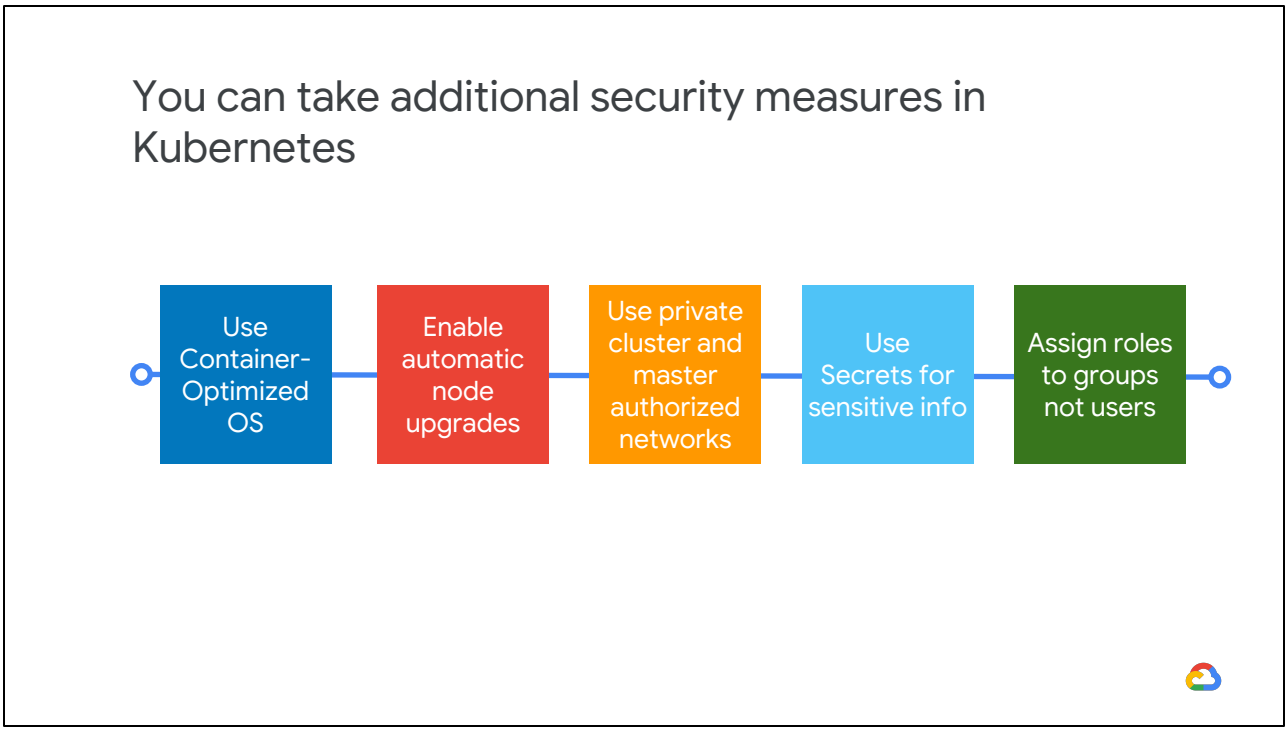
The role binding can grant permission to the creator of the Pod, which might be a Deployment, ReplicaSet, or other templated controller. Or it can grant permission to the created pod's service account. Granting the controller access to the policy would grant access for all pods created by that the controller, so the preferred method for authorizing policies is to grant access to the pod's service account.

A PodSecurityPolicy controller must be enabled

```
gcloud container clusters update [NAME] \
    --enable-pod-security-policy
```

Without a PodSecurityPolicy controller, PodSecurityPolicies mean nothing. You need BOTH to define policies AND to enable the PodSecurityPolicy controller. Careful: The order matters. If you enable the PodSecurityPolicy controller before defining any policies, you have just commanded that nothing is allowed to be deployed.

In GKE, the PodSecurityPolicy controller is disabled by default. If you choose to use PodSecurityPolicies, first define them, and then enable the controller with the gcloud command shown here. NAME represents the name of your cluster.



Many of these are enabled by default in GKE, especially if you choose to run recent versions of Kubernetes in your GKE cluster.

For example, GKE's default policy of using Google's Container-optimized OS for nodes improves node security automatically. Unlike a general-purpose Linux distribution, the Container-optimized OS implements a minimal read-only file system, performs system integrity checks, and implements firewalls, audit logging, and automatic updates.

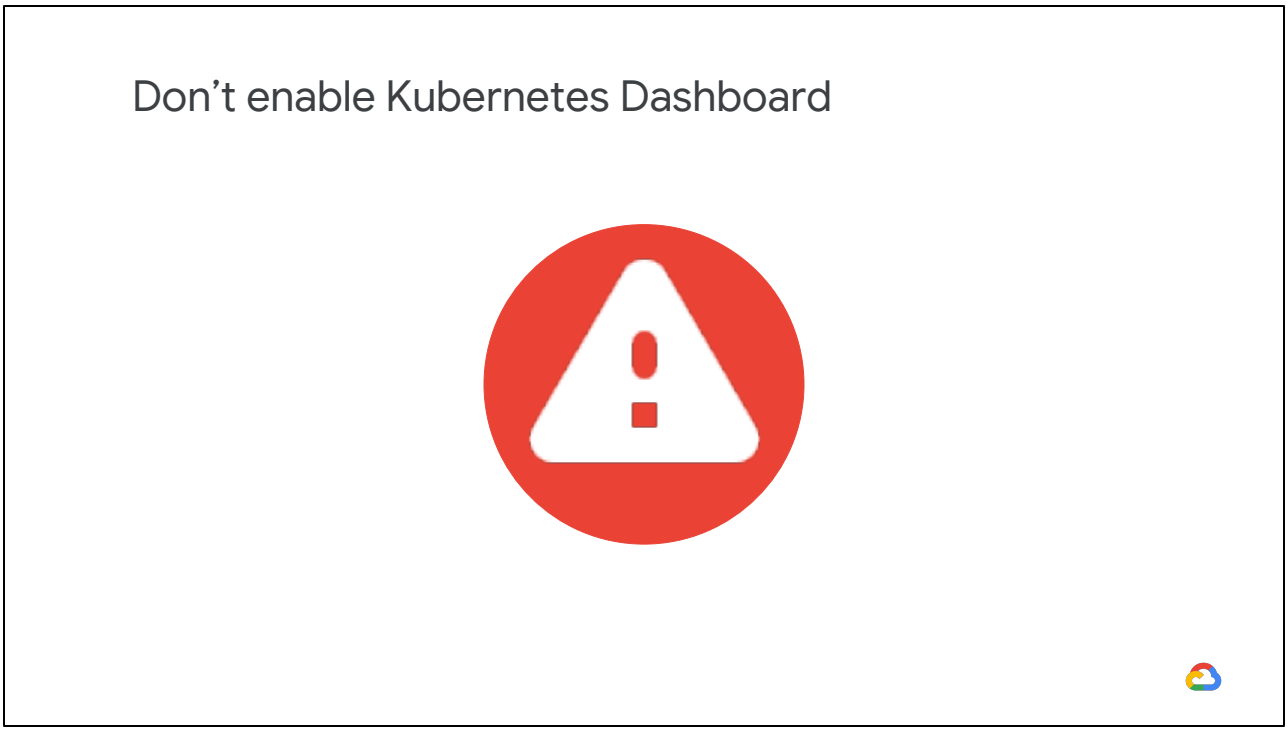
You can enable automatic node upgrades to keep all your nodes updated to the latest version of Kubernetes.

You can choose to run private clusters, which contain nodes without external IP addresses. You can also choose to to run the cluster master for a private cluster without a publicly-reachable

endpoint. By default, private clusters do not allow GCP IP addresses to access the cluster master endpoint. Using private clusters with authorized networks makes your cluster master reachable only by the specific address ranges you choose. Nodes within your cluster's VPC network can still access the master, and so can Google's internal production jobs that manage it for you.

Don’t forget to take advantage of encrypted Secrets to store sensitive configuration information, rather than storing them in ConfigMaps.

Whenever possible, grant privilege to groups rather to than individual users. This applies both to Cloud IAM, which lets you grant roles to Google groups, as well as to Kubernetes RBAC, which lets you bind roles to Kubernetes groups. Suppose you grant privileges to an administrator named “Pat” in many places, and then Pat leaves your company. You now must track down all the places where Pat has privileges and remove them. That’s tedious and error-prone. If you have followed the best practice of always grant privileges to groups rather than to users, you can remove Pat’s access simply by taking Pat out of the administrator group.



Kubernetes Dashboard is a traditional add-on to Kubernetes. It was originally intended to provide a convenient web-based way for administrators to manage a cluster.

In the past, the Kubernetes Dashboard was backed by a highly privileged Kubernetes Service Account by default. Kubernetes Dashboards were also often left accessible from the internet. This was a serious problem. The default configuration exposed an interface that might be vulnerable to remote attacks.

GKE takes steps to reduce this risk. GKE clusters running with Kubernetes version 1.7 or higher do not give the Kubernetes Dashboard admin access. GKE clusters running with Kubernetes version 1.10 or higher completely disable the Kubernetes Dashboard by default. Instead of using the Kubernetes Dashboard, use the GCP Console’s built in GKE dashboard or the kubectl

commands instead, just as you have done throughout this specialization. These interfaces provide all the old Dashboard's functionality and more, without exposing an additional attack surface. If you must run any earlier versions of Kubernetes, Google recommends that you disable the Kubernetes Dashboard, or at least restrict its permissions.

Lab

Securing Google Kubernetes Engine with Cloud IAM and Pod Security Policies

In this lab, you'll control access to GKE clusters using Cloud IAM. You'll create a Pod security policy to restrict privileged Pod creation, test that policy, and perform IP address and credential rotation.

Lab

Implementing Role-Based Access Control with Google Kubernetes Engine

In this lab, you'll create Namespaces within a GKE cluster, and then use role-based access control to permit a non-admin user to work with Pods in a specific Namespace. You'll learn how to create Namespaces for users to control access to cluster resources and create Roles and RoleBindings to control access within a Namespace.

Summary

Use Kubernetes authentication and authorization for access control

Define Kubernetes RBAC roles and role bindings for accessing resources in namespaces

Define Kubernetes RBAC cluster roles and cluster role bindings for accessing cluster-scoped resources

Define Kubernetes pod security policies to only allow pods with specific security-related attributes to run

That concludes Access Control and Security in Kubernetes and Google Kubernetes Engine. In this module, you learned how to understand the structure of GCP Identity and Access Management access controls and define Cloud IAM roles and policies for Kubernetes Engine cluster administration. You learned how to use Kubernetes authentication and authorization to control who is able to do which things with the objects in your clusters. You learned how to manage access to resources in namespaces using Kubernetes RBAC Roles. You learned how to control access to cluster-scoped resources using Kubernetes RBAC cluster roles and cluster role bindings, and you learned how to configure admission control to enforce specific security-related attributes for Pods using Kubernetes Pod security policies.

cloud.google.com