styra

Securing Kubernetes Workloads

Why RBAC is not Enough for Intent-Based APIs

Kubernetes has officially arrived - with a recent survey from the CNCF stating that 78% of enterprises are using it in production deployments. Enterprises have moved beyond experimenting, and have placed their trust in containers, and the infrastructure that supports them. Kubernetes is now mission-critical; and all the security and compliance rules and regulations of the old world need to somehow be retrofitted onto this new area. Unfortunately, the old tools for access control like RBAC simply aren't up to the challenge.

The Kubernetes API - What's Different?

The Kubernetes API was designed differently than most modern APIs. It is intent-based, meaning that people using the API think about what they want Kubernetes to be doing, not about how they make that happen. The result is an incredibly extensible, resilient, powerful and popular system.

At the same time, this intent-based API presents challenges for security. None of the standard access control solutions (role-based access control, attribute-based access control, access control lists, or IAM policies) are powerful enough to enforce even basic policies like who can change labels on a pod, or what image repositories are safe.

Kubernetes Admission Control was designed to solve this problem. Kubernetes Admission Controllers don't address access control issues out of the box, but they do allow you to use a WebHook to address authorization challenges with decoupled policy.

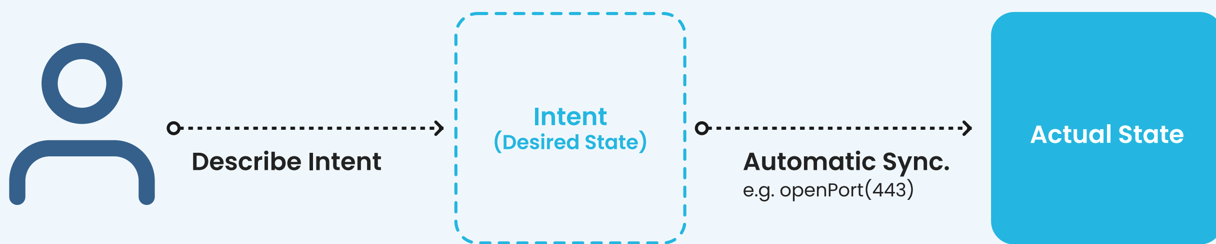
What Needs to Change for Intent-based APIs

The Kubernetes API embraces a fundamentally different paradigm than we're all accustomed to. Most APIs today are what we'll call action-based, meaning that when you think about an API call you're thinking about the action you want to execute to change how the software is running. For example, if you want to expose an application to the internet, you might run the API openport(443) that changes the network settings on their application so port 443 is open

In contrast, Kubernetes has what is known as an intent-based API, meaning that when you think about an API call you're thinking about the state you want that system to be in. You don't care what actions are required to make your desired state into a reality. You simply tell the system what you want (your intentions), and the system figures out how to make that happen—which actions to take to transition the system into

the desired state. For example, you could say that your application should be running version 1.7 of the binary, should be using persistent-storage with encryption, and should be connected to the internet. The system figures out how to upgrade or downgrade the binary, how to turn on encryption, and how to reconfigure the network to allow an internet connection.

The key architectural difference is that an intent-based system understands both the state the system is currently in (sometimes called the actual-state) and your intent for what state the system should be in (the desired-state). The system continually computes the delta between the two and takes whatever action is necessary to make the actual-state into the desired state. Users can directly change the desired-state through API calls and rely on the system itself to change the actual-state.



The Kubernetes API is intent-based. Each API call allows you to specify the desired-state for one of Kubernetes's many objects: pods, services, ingresses, configmaps, etc. To send this desired state to Kubernetes, you specify all the details in a YAML file with the appropriate information. For example, if you want to change the version of nginx, mount an external volume, or provide additional configuration, then you update the nginx-pod.yaml file to whatever the desired state should be and use kubectl apply again.

The key takeaway here is that you're not running an API like updateVersion or mountVolume. You're changing some YAML that describes what state the system should be in and saying "make it so" by running apply.

The Kubernetes API model comes with several advantages:

- **Reduced learning curve.** You learn (i) the YAML format for each object and (ii) about a dozen actions, e.g. create, apply, get, describe, delete. You need to learn the YAML configuration format for each object anyway (so you can read it). In contrast, action-based APIs additionally require you to learn what could be 1,000s of actions.
- **Extensibility.** Kubernetes supports custom resource definitions (CRDs). So in addition to all the usual pods, services, ingresses, etc. you can define your own. That's possible BECAUSE the API surface does not need to be extended to handle new resource types. You just write some YAML that describes the resource, and invoke the same dozen actions, e.g. create, apply, get, describe, delete.
- **Distributed systems.** Running large-scale systems on a cloud built using commodity hardware demands incredible resiliency in the face of failures. Kubernetes's intent-based architecture allows it to know what it should be doing, so that when, say, a hardware failure occurs it can try to compensate. Brian Grant (cotech lead on Kubernetes at Google) has written extensively about Declarative Application Management and Kubernetes Resource Management and pointed to the Kubernetes API as a key to solving many distributed-system problems: failures, distribution, auto-scaling, multiple-owners, availability, performance, reversibility.

Why RBAC is Not Enough for K8s API Security

The challenge with the Kubernetes intent-based API comes when you want to secure and safeguard the API, when you want to control which people can do what using that API.

Imagine you're the Kubernetes admin responsible for the operations, security, and compliance of the cluster. Novice Kubernetes developers need guardrails; security teams needs control and visibility; compliance teams need help mapping age-old regulations to this brand new system; and you know from your own experiences which Kubernetes best practices you need to adopt. Ideally you'd enforce those rules, regulations, and best-practices within Kubernetes itself by setting up access control.

Role-Based Access Control (RBAC) has been the solution for decades, enabling admins to control which users can run which APIs on which resources. Kubernetes RBAC (available since late 2017) is the first line of defense. It lets you give read-only access to resources for specific user-groups. It lets you isolate different user groups (though not entirely) by assigning them different portions (aka namespaces) of Kubernetes. It lets you restrict permissions for service-accounts. All of which is valuable.

But in contrast to action-based systems where RBAC handles the vast majority of access control needs, RBAC in Kubernetes provides far less control because of its intent-based API. From the API's perspective there are only about a dozen actions, which means that if (for example) Alice can update a given resource, she can update any portion of that resource.

For example, SREs need to read most of the resources in the cluster so that they can diagnose problems when they arise. But when an SRE finds a problem on a node, e.g. a noisy neighbor, she may need to drain that node in order to move the workloads to a different node and mitigate the problem. Unfortunately, the API does not have drain actions — those are macros provided by the CLI that simply update the annotations on the node. Using RBAC to try to reach this level of granularity is tedious and complex, to the point that it's impractical.

The Intent-based K8s RBAC diagram below shows conceptually what you have to work with using RBAC — you can choose which combinations of the users/actions/resources combinations are permitted.

Intent-based K8s RBAC		
Users	**Actions**	**Resources**
Allison Betty Charlyn Domingo Ethan Felix Georgina Hettie Inga Jess	create delete apply get describe	pod1 pod2 ingress1 ingress2 deployment1 deployment2 deployment3 service1 service2

In contrast, imagine for a moment if Kubernetes were action-based (e.g. it included APIs like cordon, drain, setImage, mountVolume, openPort). Then we could use RBAC to grant read along with cordon and drain but nothing else. Action-based APIs simply have more names that you can use when writing RBAC policies.

Ficticious Action-based K8s RBAC		
Users	**Actions**	**Resources**
Allison Betty Charlyn Domingo Ethan Felix Georgina Hettie Inga Jess	create delete apply get describe setImages openPort mountVolume setReplicas setLabel setAnnotation setRestart setHealth setLiveness setInit setMemLimits setCPULimits ...	pod1 pod2 ingress1 ingress2 deployment1 deployment2 deployment3 service1 service2

In short, the Kubernetes API provides a powerful, extensible, unified resource model, but it is that same resource model that makes RBAC too coarse-grained for many use cases. RBAC is invaluable for the controls it can provide, but far more so than other systems, Kubernetes requires additional controls beyond RBAC.

What Do We Need for K8s API Security?

So if RBAC doesn't provide enough control, what do we need? Let's look at an example: "all pods must only use images from trusted repositories" (say, hooli.com). Anytime someone runs, for example, kubectl apply, then the access control system needs to make a decision based on the user, the action apply, and the YAML that describes that pod.

To make the right decision, the access control system needs to extract the list of image names (e.g. nginx and hooli.com/frontend) and do string manipulation to extract the name of the repository (e.g. the default repo and hooli.com).

One option is to build a bunch of knowledge about Kubernetes resources into the access control system itself. Then the admin could write a policy about who can, for example, update labels, what the permitted-image-registries are, and so forth. That's what most systems do—invent a bunch of entitlements and build a custom access control system on top.

But building a custom access control system won't work for Kubernetes because it allows users and vendors to invent their own YAML formats (Custom Resource Definitions) and install code that implements them. So Kubernetes's resource extensibility requires any custom Kubernetes access control system to be extensible itself.

What we really need is an access control system that lets the administrator write policies that:

- Descend through the hierarchical structure of a YAML file.
- Iterate over elements in an array.
- Manipulate strings, IPs, numbers, etc.

As you may have guessed, none of the standard access control paradigms meet these requirements. That includes role-based access control (RBAC), attribute-based access control (ABAC), access control lists (ACLs), and even IAM-style policies.

Admission Control to the Rescue

Fortunately, the Kubernetes team foresaw this problem and created an Admission Control mechanism where you can put controls that go far beyond RBAC and the standard access control mechanisms. The Kubernetes API server provides a pipeline of access controls, broken into Authorization (e.g. RBAC), and Admission.

Authorization happens on every API call, and Admission happens only on updates (creates, updates, and deletes). With Authorization you're provided the following information to make a decision:

- User: user, groups, extra attributes provided by authentication
- Action:path, API verb, HTTP verb
- Resource: resource, subresource, namespace, API group

With Admission you're handed an AdmissionReview object in YAML. It includes all the information about the resource being modified to make whatever decision you want. You can , of course, build whatever logic you need to secure your API by writing, deploying, and maintaining custom code that implements the Admission Control webhook protocol (a simple HTTP/json API).

But if you don't want to support and maintain custom code, you can use the Open Policy Agent as a Kubernetes admission controller and leverage its declarative policy language. That language includes the required expressiveness outlined above: iteration, dot-notation, and 50+ builtins for string-manipulation and the like, as well as a community of practitioners with best practices built from hundreds of production deployments.

New Technologies Need New Solutions

The Kubernetes API changes can be summarized as follows:

- Kubernetes's intent-based API lets users focus on what state they want Kubernetes to be in, not how to achieve it.
- One of the core benefits of the intent-based approach is that it enables Kubernetes to be resilient in the face of failures. Because the system knows what it should be doing, when failures happen, Kubernetes knows how to recover.
- Kubernetes's API also provides tremendous extensibility. Users can create their own custom resources without having to extend the API.
- The challenge with Kubernetes's API is that an access control decision may need to analyze an arbitrary YAML document, e.g. using dot-notation, iteration, and string-manipulation. Standard access control systems like RBAC, ABAC, ACLs and IAM simply aren't expressive enough.

The Kubernetes team introduced Admission Control to give users additional power to control the API. Creating effective admission control policies to secure your workloads can be simplified with declarative authorization solutions like Styra, or Open Policy Agent. Using these solutions as Kubernetes Admission Controllers, gives you the needed expressiveness to overcome new access challenges with the granularity needed to be truly effective.

About Styra

We are reinventing policy and authorization for cloud-native. Today's cloud app infrastructure has evolved. Access, security, and compliance must also evolve. It's time for a new paradigm. It's time for authorization-as-code.

Learn more at www.styra.com

©2022 Styra, Inc. All rights reserved. **styra_wp_securing-kubernetes-workloads_042822**