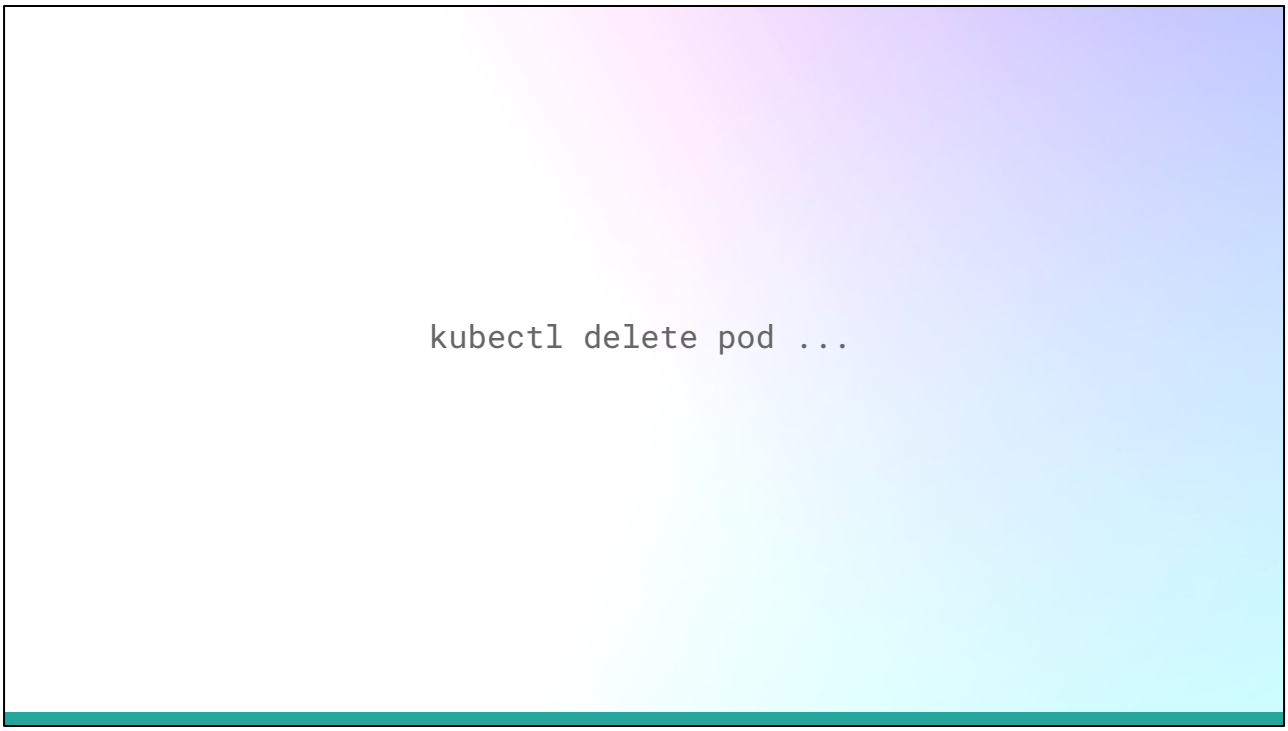
Scalable User Authentication for Kubernetes with OpenID Connect

Nathan Brahms · Shashwat Sehgal · Gergely Dányi

POSECURITY

- Access in Kubernetes
- Authentication trade-offs
- OIDC in Kubernetes
- P0's OSS repo
- Demo
- Authorization


Access in
Kubernetes



Let’s start with what happens when you run a kubectl command.

```
DELETE /api/v1/namespaces/.../pods/... HTTP/1.1
Authorization: Bearer ...
```

Kubectl is just a wrapper for Kubernetes' REST API, so calling `kubectl` will send an HTTP request to the Kubernetes API server. The request will look something like this. It will have a method, a path, and an authorization header. The path namespace and authorization header will be derived by kubectl from your kube config.

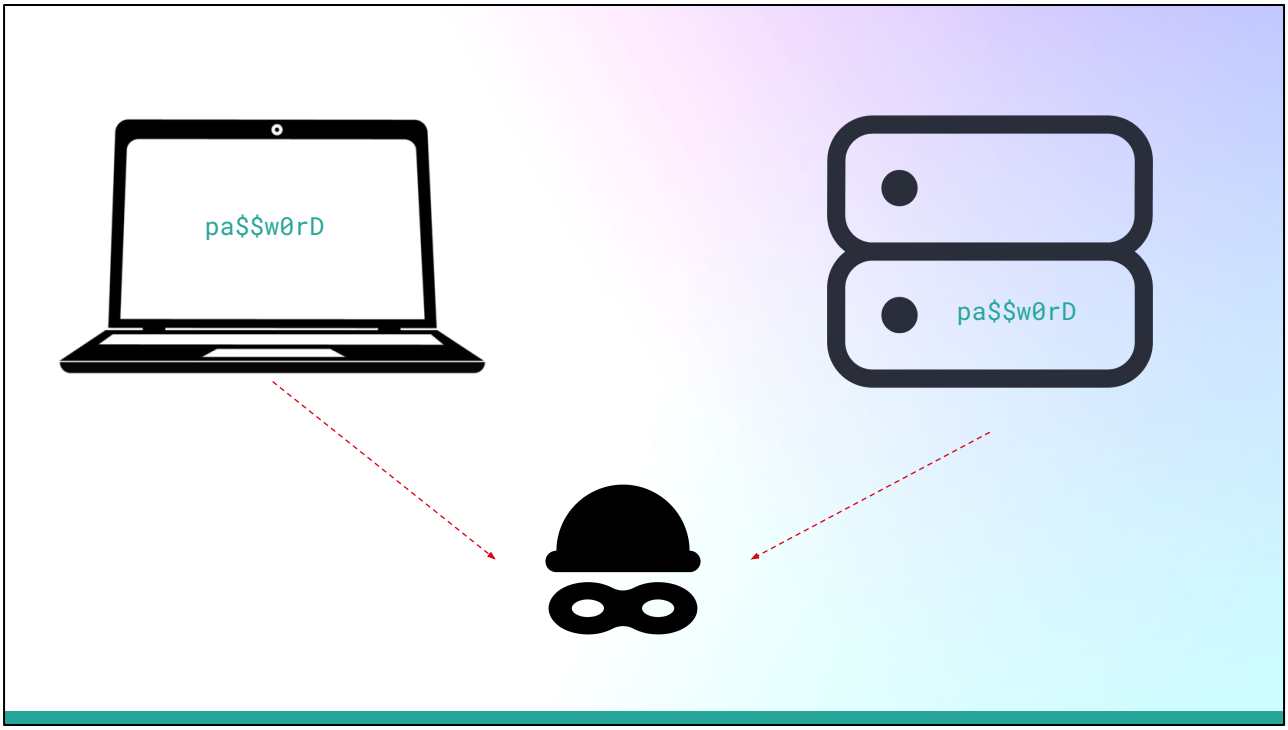
```
DELETE /api/v1/namespaces/.../pods/... HTTP/1.1
Authorization: Bearer ...
```

```
allow(request): boolean {
  who := authn(request.headers.authorization)
  what := resource(request.path,request.path)
  return authz(who, what)
}
```

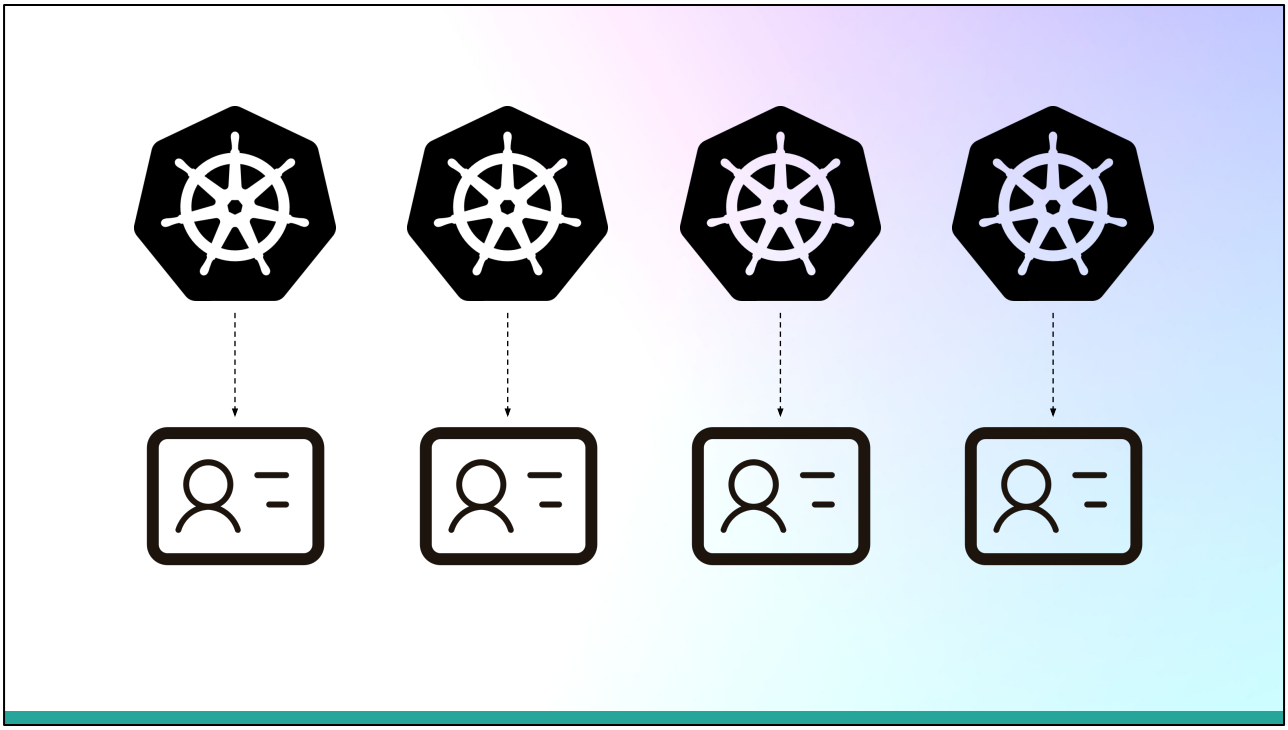
When Kubernetes receives this web request, it decides whether or not to allow you to execute the web request. In oversimplified terms, pseudocode for this decision follows this form of an access allow function. First, Kubernetes uses an authentication module to figure out _WHO_ is making the request. It combines that with _WHAT_ is being requested, then passes that to its RBAC engine to make an authorization decision.

The majority of this talk will be about configuring Kubernetes to answer this first “WHO” question securely, scalably, and with the least amount of effort on your part as the Kubernetes maintainer. At the end of the talk, Shashwat will briefly discuss how to apply similar principals to authorization.

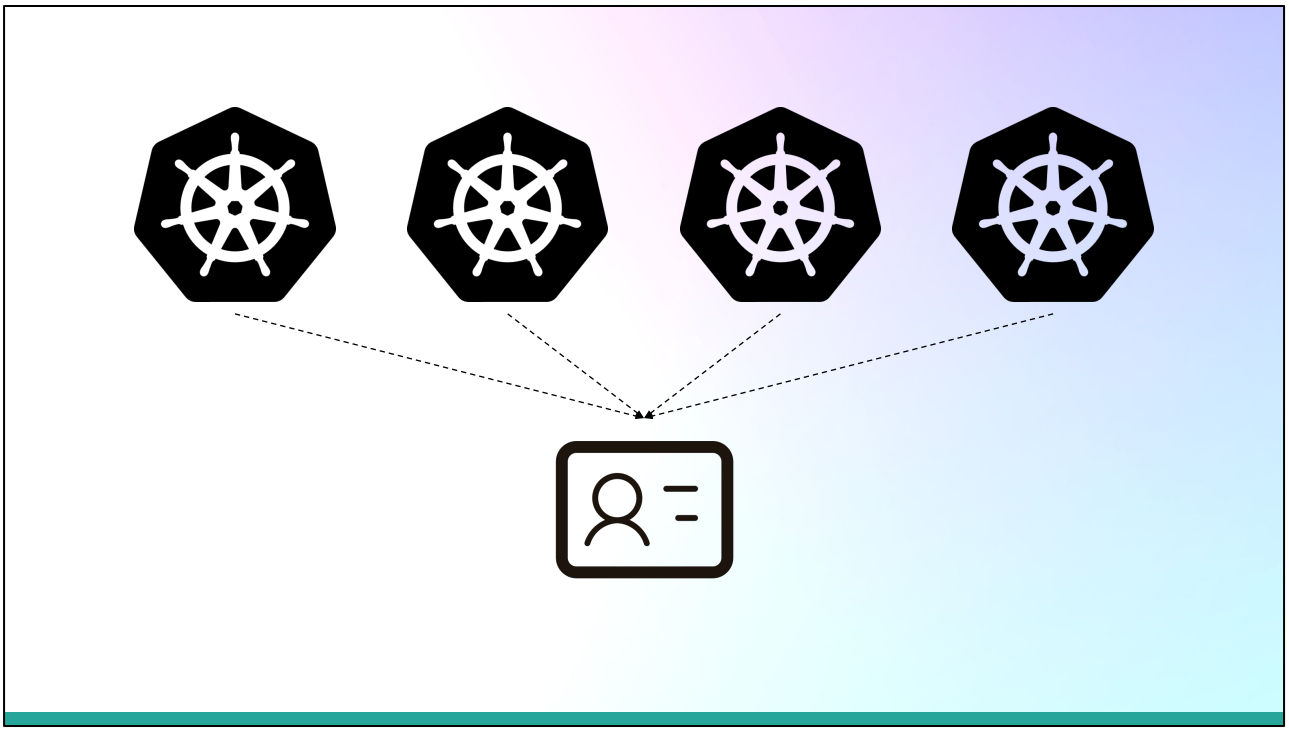
In order to figure out how we'd like to set up authentication, let's go through some of the concerns you may have, or outcomes you want to achieve:



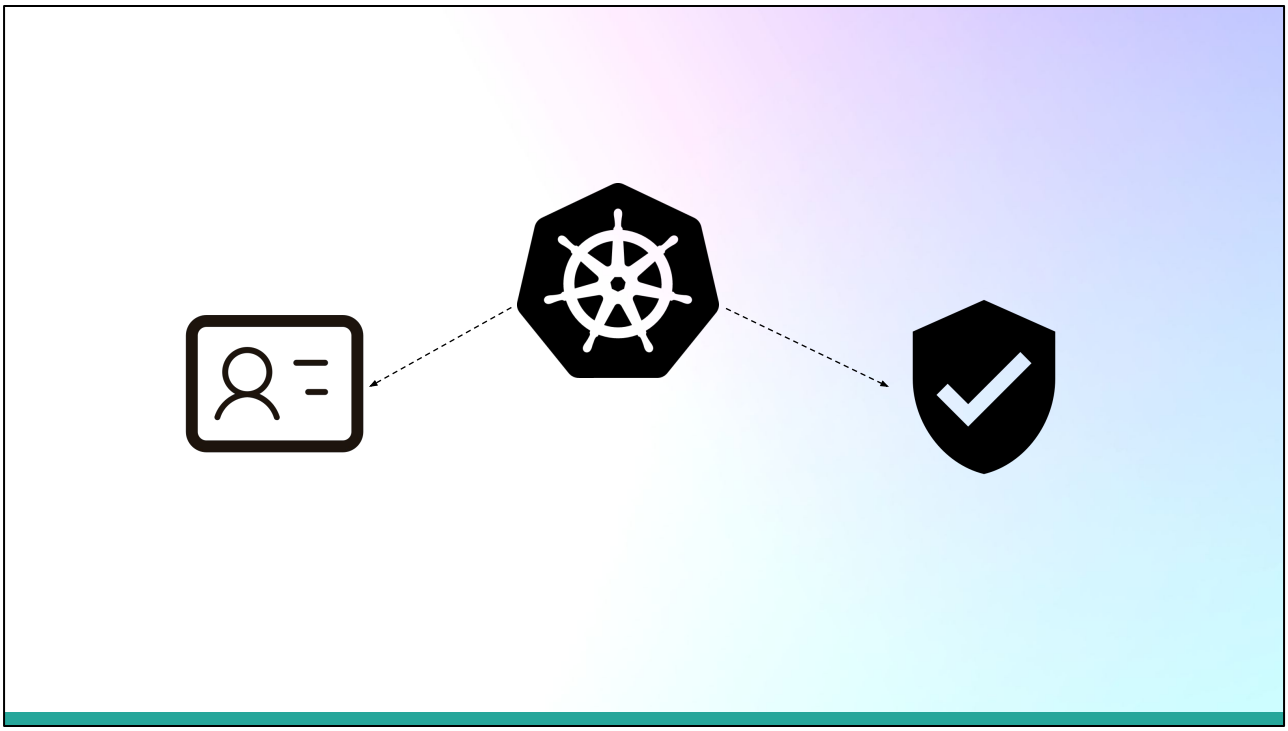
First, you'd like authentication to be secure against credential leaks. We want to avoid clear-text credentials, like passwords, API keys, or private SSH keys, from living on either the user's client machine, or the Kubernetes server. So if an adversary gains access to the filesystems of either of these, they won't be able to impersonate users.



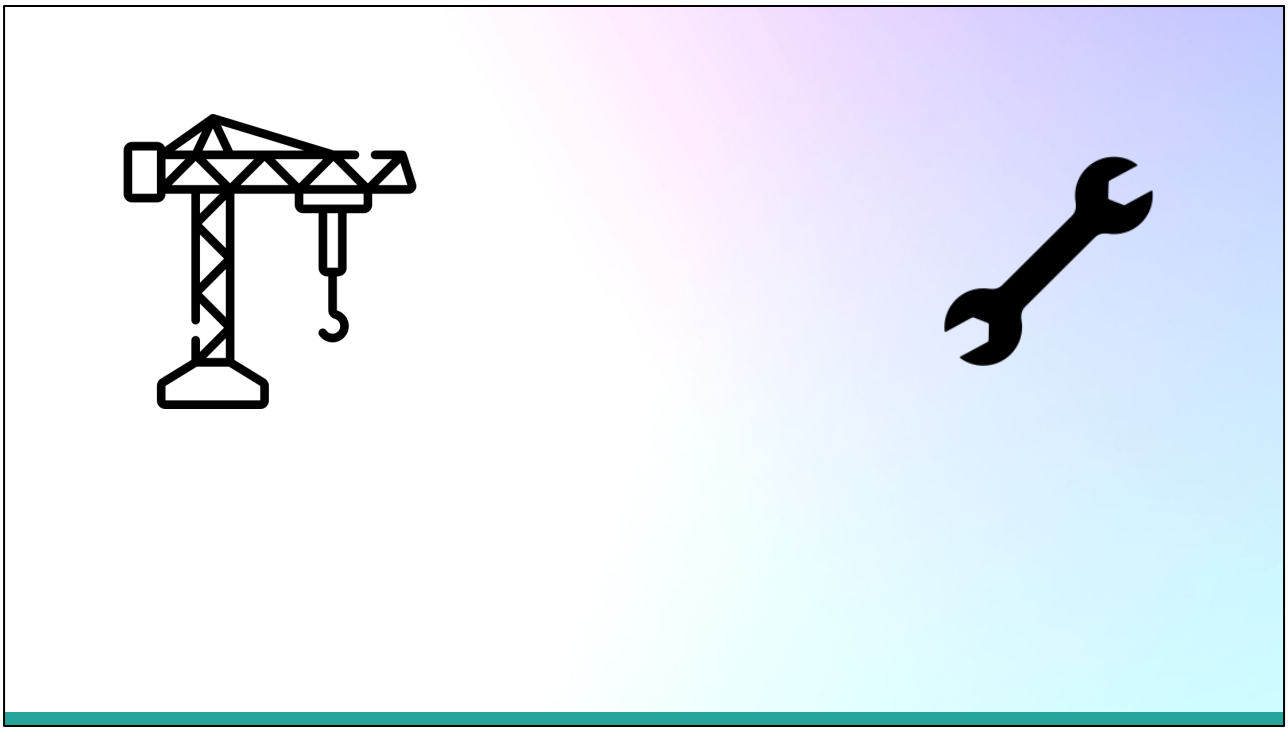
Next, you want your authentication mechanism to scale as you add Kubernetes clusters. Ideally, you have a single identity provider, linked to your organization, that is the single source of identity.



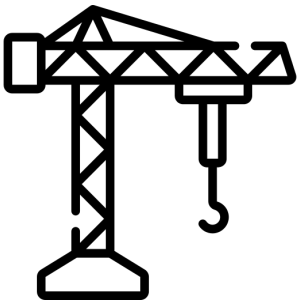
This way you can easily add, remove, and update users, and monitor their activity and security in one place.



Moreover, you may want the authentication system to yield more information than just the user's name, so you can use that information in authorization decisions. For instance, you may want to configure Kubernetes so that every member of an organization group gets the same access. In this case, you need your authentication system to provide each user's group memberships as part of authentication.



Finally, but importantly, your authentication method should be easy both to set up and to maintain



Method

Token file

SA tokens

X509

Webhook / Proxy

Provider

OIDC

Now let’s list the various authentication methods in Kubernetes…

	Theft-resistant?				Ease-of-use	
Method	**Client**	**Server**	**Scaling?**	**Authz data?**	**Setup**	**Maintenance**
Token file						
SA tokens						
X509						
Webhook / Proxy						
Provider						
OIDC						

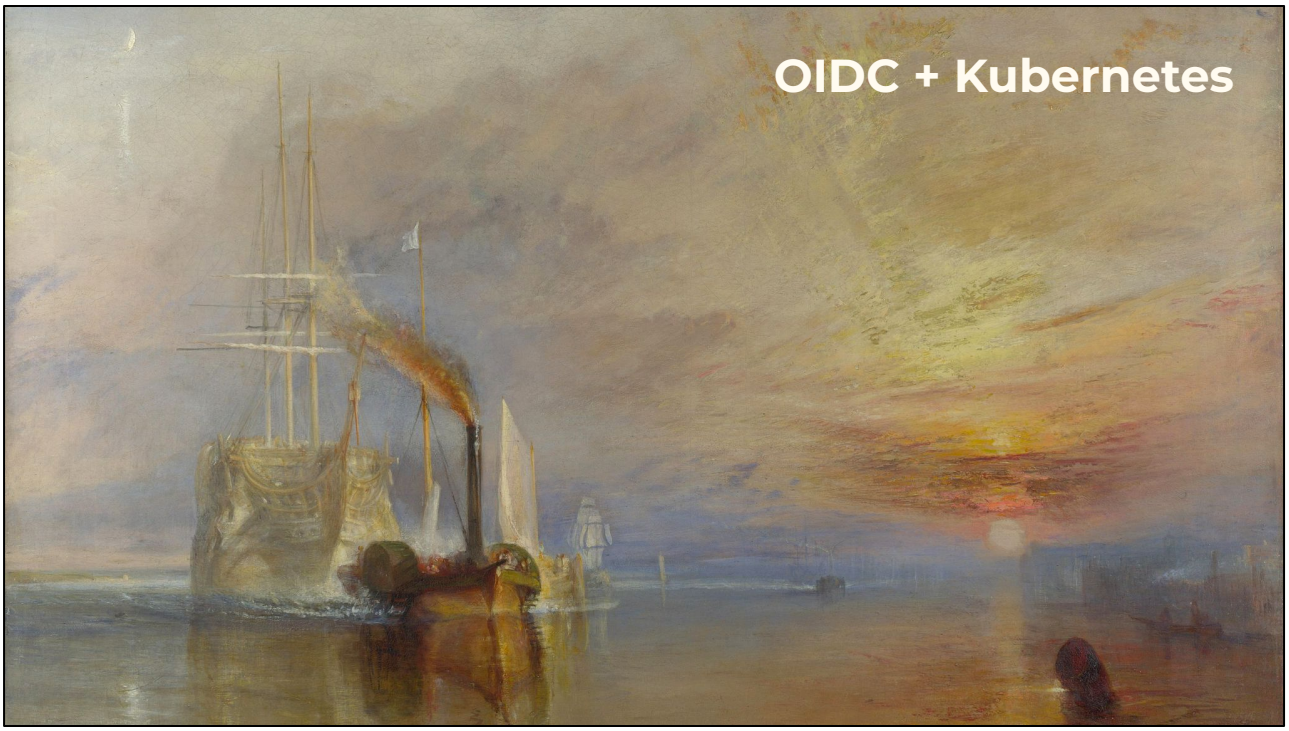
… and score them against each of these criteria. Here I'm going to show ratings for each of these, but I'll only talk in depth about the two most easily implementable secure methods, authentication via your cloud-service provider, and OIDC. With provider authentication, you map identities in your CSP's IAM to Kubernetes identities. With OIDC, you use an OIDC plugin in your identity provider: Okta, Azure AD, Google Workspace, or the like, to create identities in Kubernetes.

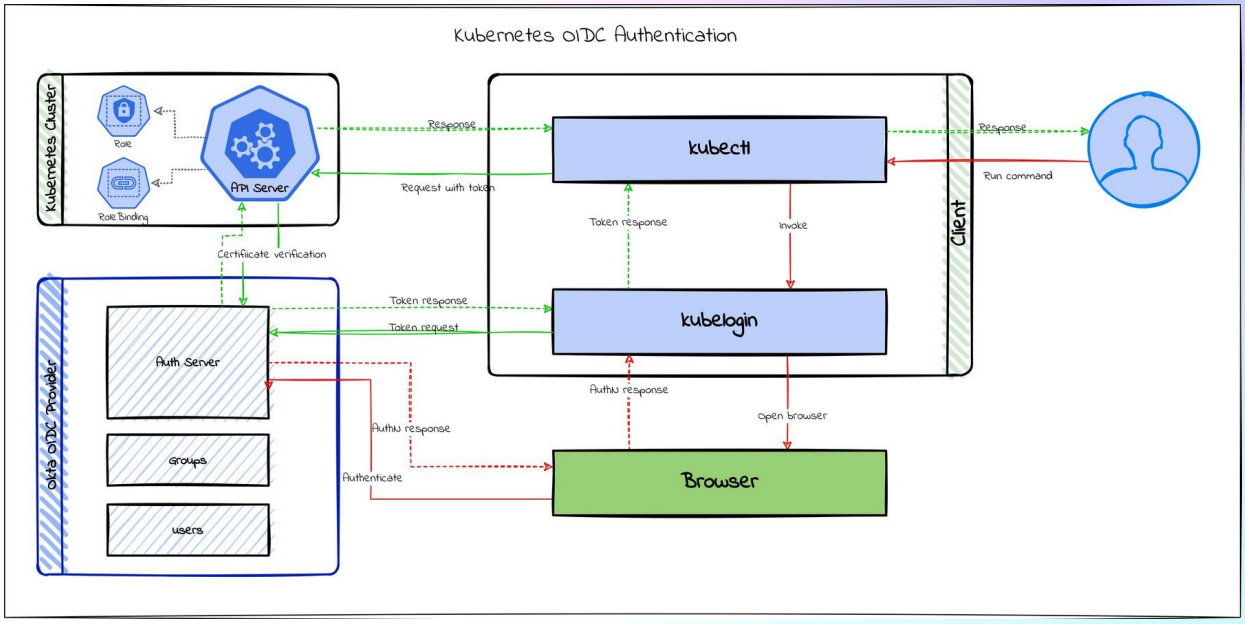
	Theft-resistant?				Ease-of-use	
Method	**Client**	**Server**	**Scaling?**	**Authz data?**	**Setup**	**Maintenance**
Token file	❌	❌	❌	❌	😁	😐
SA tokens	❌	❌	❌	❌	😁	😁
X509	❌	✅	✅	⚠️	🙂	🙂
Webhook / Proxy	⚠️	⚠️	✅	✅	😓	😓
Provider	✅	✅	⚠️	✅	🙂	😓
OIDC	✅	✅	✅	✅	🙂*	😁

Here are those rankings. Of the scalable methods, I'm ignoring X509 because it's vulnerable to credential leakage, and ignoring webhook and proxy methods as they tend to require a lot of implementation overhead.

Let's focus on provider and OIDC authentication. How scalable the method is depends on the provider. But, in the most widely used provider, AWS, each account is treated separately. So you'll scale to multiple clusters in an account, but not clusters across multiple accounts. And, if you have clusters in multiple cloud providers, you'll never scale with provider authentication. In addition, provider authentication can require a large amount of maintenance if you want to add access for additional user groups in the future.

OIDC authentication is both easy to maintain and scales naturally. Instead, OIDC authentication can be intimidating at setup time, as I'll show you in a couple slides. But, in this talk I'll show you some open source tools that can make this setup relatively painless.

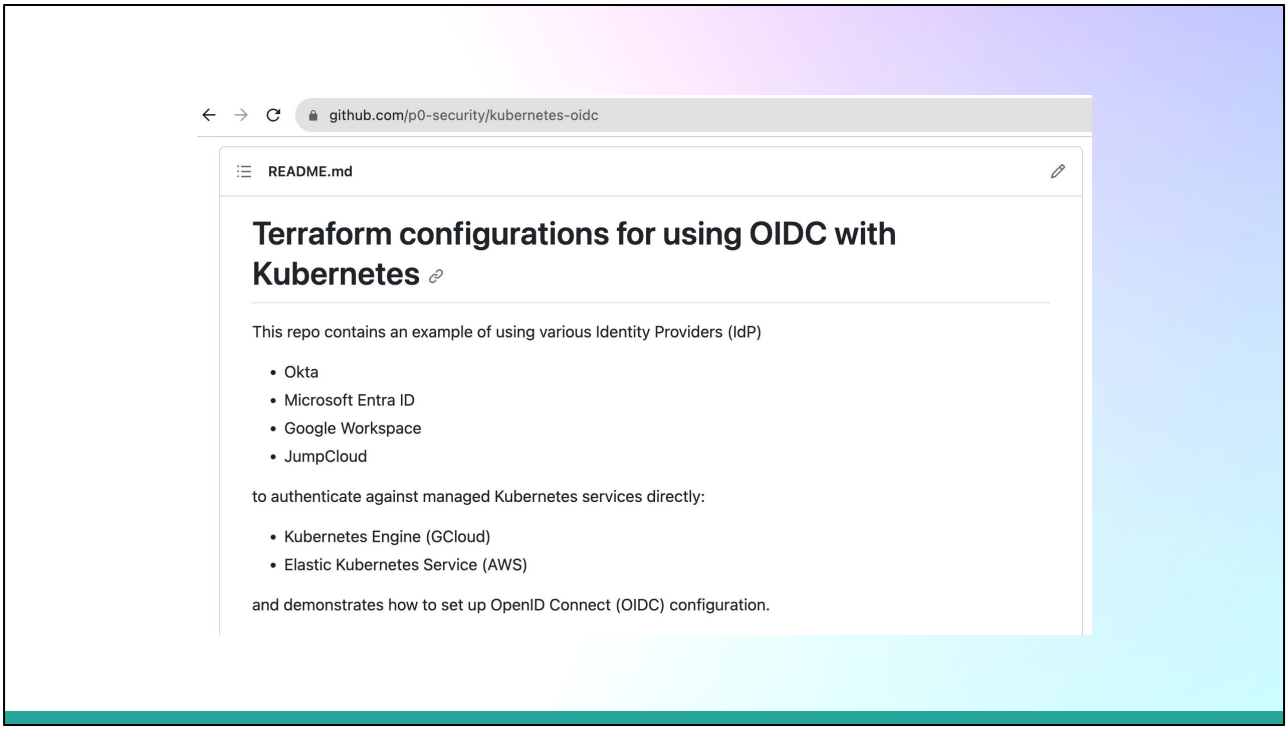

OIDC + Kubernetes



Source: Okta

Here's a quick overview of OIDC authentication in Kubernetes, with this graphic shamelessly borrowed from Okta. The system involves three major components that you as the Kubernetes maintainer need to set up: The Kubernetes authentication backend, the OIDC application itself, and the client environment. These can be done manually, but it's much less work with the open-source tools I'm about to show you.

As a companion to this talk, we've created an open-source repo that you can use, in combination with Terraform, to automatically provision the entire Kubernetes - OIDC authentication stack.



The repo is public, just go to p0-security/kubernetes-oidc on GitHub, clone the repo, and follow the README to get setup.

In fact, let's just do that now as a demo.

It’s

DEMO TIME!!!

We've talked about setting up scalable authentication using OIDC. But how do you securely scale authorization? In the last couple minutes of this talk, we'd like to show you how we solve this problem at P0.

Authorization is important..

- Limit blast radius: Principle of Least Privilege
- Compliance

But HARD to implement...Why?

- High cardinality causes group sprawl
- Knowledge of IAM

P0 makes Privileged Access easy

(included in our free tier for small teams)

- Intent Based Access Control
- Common Use Cases
  - Restart a deployment
  - Delete a pod
  - Port forward traffic
  - Namespace admin
  - Shell (Kubectl exec)

It’s

DEMO TIME!!!

Thank you!

Nathan Brahms

Shashwat Sehgal

Gergely Dányi

POSECURITY

mastodon.social/@nathanbrahms

nathan-brahms-a9787251

@shashwatsehgal

shashwatsehgal

gergely-danyi

p0-security