EXPERT INSIGHT

Kubernetes – An Enterprise Guide

Effectively containerize applications, integrate enterprise systems, and scale applications in your enterprise

Second Edition

Marc Boorshtein
Scott Surovich

Packt>

This chapter is an excerpt from the book Kubernetes – An Enterprise Guide, Second Edition by Marc Boorshtein and Scott Surovich. With significant updates in each chapter, this revised edition will help you acquire the knowledge and tools required to integrate Kubernetes clusters in an enterprise environment.

5

Integrating Authentication into Your Cluster

Once a cluster has been built, users will need to interact with it securely. For most enterprises, this means authenticating individual users and making sure they can only access what they need in order to do their jobs. With Kubernetes, this can be challenging because a cluster is a collection of APIs, not an application with a frontend that can prompt for authentication.

In this chapter, you'll learn how to integrate enterprise authentication into your cluster using the OpenID Connect protocol and Kubernetes impersonation. We'll also cover several anti-patterns and explain why you should avoid using them.

In this chapter, we will cover the following topics:

- Understanding how Kubernetes knows who you are
- Understanding OpenID Connect
- Configuring KinD for OpenID Connect
- How cloud Kubernetes knows who you are
- Configuring your cluster for impersonation
- Configuring impersonation without OpenUnison
- Authenticating pipelines to your cluster

Let's get started!

Technical requirements

To complete the exercises in this chapter, you will require the following:

- An Ubuntu 20.04 server with 8 GB of RAM
- A KinD cluster running with the configuration from *Chapter 2*, *Deploying Kubernetes Using KinD*

You can access the code for this chapter at the following GitHub repository: `https://github.com/PacktPublishing/Kubernetes---An-Enterprise-Guide-2E/tree/main/chapter5`.

Understanding how Kubernetes knows who you are

In the 1999 sci-fi film *The Matrix*, Neo talks to a child about the Matrix as he waits to see the Oracle. The child explains to him that the trick to manipulating the Matrix is to realize that "*There is no spoon.*"

This is a great way to look at users in Kubernetes, because they don't exist. With the exception of service accounts, which we'll talk about later, there are no objects in Kubernetes called "User" or "Group." Every API interaction must include enough information to tell the API server who the user is and what groups the user is a member of. This assertion can take different forms, depending on how you plan to integrate authentication into your cluster.

In this section, we will get into the details of the different ways Kubernetes can associate a user with a cluster.

External users

Users who are accessing the Kubernetes API from outside the cluster will usually do so using one of two authentication methods:

- **Certificate**: You can assert who you are using a client certificate that has information about you, such as your username and groups. The certificate is used as part of the TLS negotiation process.
- **Bearer token**: Embedded in each request, a bearer token can either be a self-contained token that contains all the information needed to verify itself or a token that can be exchanged by a webhook in the API server for that information.

You can also use service accounts to access the API server outside the cluster, though it's strongly discouraged. We'll cover the risks and concerns around using service accounts in the *Other authentication options* section.

Groups in Kubernetes

Different users can be assigned the same permissions without creating `RoleBinding` objects for each user individually via groups. Kubernetes includes two types of groups:

- **System assigned**: These groups start with the `system:` prefix and are assigned by the API server. An example group is `system:authenticated`, which is assigned to all authenticated users. Another example of system-assigned groups is the `system:serviceaccounts:namespace` group, where `Namespace` is the name of the namespace that contains all the service accounts for the namespace named in the group.
- **User-asserted groups**: These groups are asserted by the authentication system either in the token provided to the API server or via the authentication webhook. There are no standards or requirements for how these groups are named. Just as with users, groups don't exist as objects in the API server. Groups are asserted at authentication time by external users and tracked locally for system-generated groups. When asserting a user's groups, the primary difference between a user's unique ID and groups is that the unique ID is expected to be unique, whereas groups are not.

You may be authorized for access by groups, but all access is still tracked and audited based on your user's unique ID.

Service accounts

Service accounts are objects that exist in the API server to track which pods can access the various APIs. Service account tokens are called **JSON Web Tokens**, or **JWTs**, and depending on how the token was generated, there are two ways to obtain a service account:

- The first is from a secret that was generated by Kubernetes when the service account was created.
- The second is via the `TokenRequest` API, which is used to inject a secret into pods via a mount point or used externally from the cluster. All service accounts are used by injecting the token as a header in the request into the API server. The API server recognizes it as a service account and validates it internally.

Unlike users, service accounts **CANNOT** be assigned to arbitrary groups. Service accounts are members of pre-built groups only; you can't create a group of specific service accounts for assigning roles.

Now that we have explored the fundamentals of how Kubernetes identifies users, we'll explore how this framework fits into the **OpenID Connect** (**OIDC**) protocol. OIDC provides the security most enterprises require and is standards-based, but Kubernetes doesn't use it in a way that is typical of many web applications. Understanding these differences and why Kubernetes needs them is an important step in integrating a cluster into an enterprise security environment.

Understanding OpenID Connect

OpenID Connect is a standard identity federation protocol. It's built on the OAuth2 specification and has some very powerful features that make it the preferred choice for interacting with Kubernetes clusters.

The main benefits of OpenID Connect are as follows:

- **Short-lived tokens**: If a token is leaked, such as via a log message or breach, you want the token to expire as quickly as possible. With OIDC, you're able to specify tokens that can live for 1-2 minutes, which means the token will likely be expired by the time an attacker attempts to use it.
- **User and group memberships**: When we start talking about authorization, we'll see quickly that it's important to manage access by group instead of managing access by referencing users directly. OIDC tokens can embed both the user's identifier and their groups, leading to easier access management.
- **Refresh tokens scoped to timeout policies**: With short-lived tokens, you need to be able to refresh them as needed. The time a refresh token is valid for can be scoped to your enterprise's web application idle timeout policy, keeping your cluster in compliance with other web-based applications.
- **No plugins required for kubectl**: The `kubectl` binary supports OpenID Connect natively, so there's no need for any additional plugins. This is especially useful if you need to access your cluster from a jump box or VM because you're unable to install the **Command-Line Interface** (**CLI**) tools directly onto your workstation.
- **More multi-factor authentication options**: Many of the strongest multi-factor authentication options require a web browser. Examples include FIDO U2F and WebAuthn, which use hardware tokens.

OIDC is a peer-reviewed standard that has been in use for several years and is quickly becoming the preferred standard for identity federation.

Identity federation is the term used to describe the assertion of identity data and authentication without sharing the user's confidential secret or password. A classic example of identity federation is logging into your employee website and being able to access your benefits provider without having to log in again. Your employee website doesn't share your password with your benefits provider. Instead, your employee website *asserts* that you logged in at a certain date and time and provides some information about you. This way, your account is *federated* across two silos (your employee website and benefits portal), without your benefits portal knowing your employee website password.

The OpenID Connect protocol

As you can see, there are multiple components to OIDC. To fully understand how OIDC works, let's begin with understanding the OpenID Connect protocol.

The two aspects of the OIDC protocol we will be focusing on are as follows:

- Using tokens with `kubectl` and the API server
- Refreshing tokens to keep your tokens up to date

We won't focus too much on obtaining tokens. While the protocol to get a token does follow a standard, the login process does not. How you obtain tokens from an identity provider will vary, and it's based on how you choose to implement the OIDC **Identity Provider** (**IdP**).

Three tokens are generated from an OIDC login process:

- `access_token`: This token is used to make authenticated requests to web services your identity provider may provide, such as obtaining user information. It is NOT used by Kubernetes and can be discarded. This token does not have a standard form. It may be a JWT, it may not.
- `id_token`: This token is a JWT that encapsulates your identity, including your unique identifier, groups, and expiration information about you that the API server can use to authorize your access. The JWT is signed by your identity provider's certificate and can be verified by Kubernetes simply by checking the JWT's signature. This is the token you pass to Kubernetes for each request to authenticate yourself.

- refresh_token: kubectl knows how to refresh your id_token for you automatically once it expires. To do this, it makes a call to your IdP's token endpoint using a refresh_token to obtain a new id_token. A refresh_token can only be used once and is opaque, meaning that you, as the holder of the token, have no visibility into its format and it really doesn't matter to you. It either works, or it doesn't. The refresh_token never goes to Kubernetes (or any other application). It is only used in communications with the IdP.

Once you have your tokens, you can use them to authenticate with the API server. The easiest way to use your tokens is to add them to the kubectl configuration using command-line parameters:

```
kubectl config set-credentials username --auth-provider=oidc --auth-provider-arg=idp-issuer-url=https://host/uri --auth-provider-arg=client-id=kubernetes --auth-provider-arg=refresh-token=$REFRESH_TOKEN --auth-provider-arg=id-token=$ID_TOKEN
```

config set-credentials has a few options that need to be provided. We have already explained id-token and refresh_token, but there are two additional options:

- idp-issuer-url: This is the same URL we will use to configure the API server and points to the base URL used for the IdP's discovery URL.
- client-id: This is used by your IdP to identify your configuration. This is unique to a Kubernetes deployment and is not considered secret information.

The OpenID Connect protocol has an optional element, known as a client_secret, that is shared between an OIDC client and the IdP. It is used to "authenticate" the client prior to making any requests, such as refreshing a token. While it's supported by Kubernetes as an option, it's recommended to not use it and instead configure your IdP to use a public endpoint (which doesn't use a secret at all).

The client secret has no practical value since you'd need to share it with every potential user and since it's a password, your enterprise's compliance framework will likely require that it is rotated regularly, causing support headaches. Overall, it's just not worth any potential downsides in terms of security.

Kubernetes requires that your identity provider supports the discovery URL endpoint, which is a URL that provides some JSON to tell you where you can get keys to verify JWTs and the various endpoints available. Take any issuer URL and add /.well-known/openid-configuration to see this information.

Following OIDC and the API's interaction

Once `kubectl` has been configured, all of your API interactions will follow the following sequence:

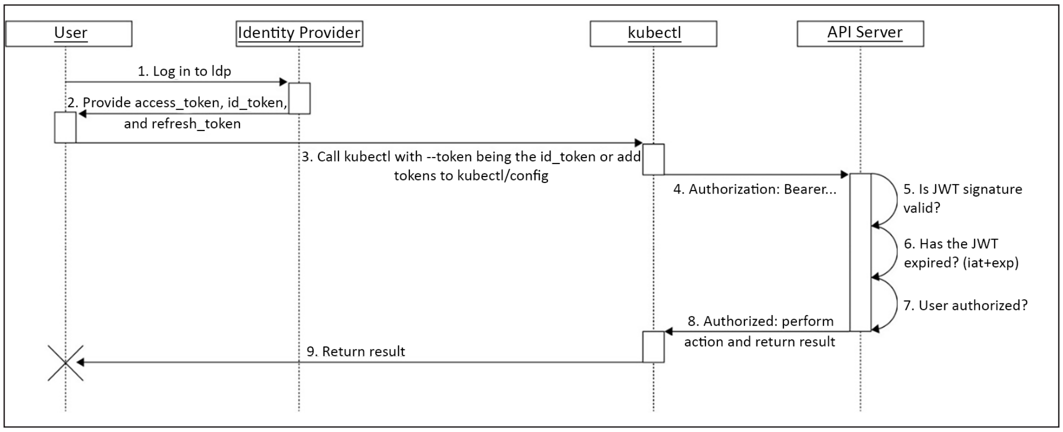

User
Identity Provider
kubectl
API Server
1. Log in to ldp
2. Provide access_token, id_token, and refresh_token
3. Call kubectl with --token being the id_token or add tokens to kubectl/config
4. Authorization: Bearer...
5. Is JWT signature valid?
6. Has the JWT expired? (iat+exp)
7. User authorized?
8. Authorized: perform action and return result
9. Return result

Figure 5.1: Kubernetes/kubectl OpenID Connect sequence diagram

The preceding diagram is from Kubernetes' authentication page at `https://kubernetes.io/docs/reference/access-authn-authz/authentication/#openid-connect-tokens`. Authenticating a request involves doing the following:

1. **Log in to your IdP**: This will be different for each IdP. This could involve providing a username and password to a form in a web browser, a multi-factor token, or a certificate. This will be specific to every implementation.
2. **Provide tokens to the user**: Once authenticated, the user needs a way to generate the tokens needed by `kubectl` to access the Kubernetes APIs. This can take the form of an application that makes it easy for the user to copy and paste them into the configuration file, or can be a new file to download.
3. This step is where `id_token` and `refresh_token` are added to the `kubectl` configuration. If the tokens were presented to the user in the browser, they can be manually added to the configuration. Alternatively, some solutions provide a new `kubectl` configuration to download at this step. There are also `kubectl` plugins that will launch a web browser to start the authentication process and, once completed, generate your configuration for you.
4. **Inject id_token**: Once the `kubectl` command has been called, each API call includes an additional header, called the **Authorization** header, that includes `id_token`.

5. **JWT signature validation**: Once the API server receives `id_token` from the API call, it validates the signature against the public key provided by the identity provider. The API server will also validate whether the issuer matches the issuer for the API server configuration, and also that the recipient matches the client ID from the API server configuration.
6. **Check the JWT's expiration**: Tokens are only good for a limited amount of time. The API server ensures that the token hasn't expired.
7. **Authorization check**: Now that the user has been authenticated, the API server will determine whether the user identified by the provided `id_token` is able to perform the requested action by matching the user's identifier and asserted groups to internal policies.
8. **Execute the API**: All checks are complete and the API server executes the request, generating a response that will be sent back to `kubectl`.
9. **Format the response for the user**: Once the API call is complete (or a series of API calls), the response in JSON is formatted and presented to the user by `kubectl`.

In general terms, authentication is the process of validating you are you. Most of us encounter this when we put our username and password into a website. We're proving who we are. In the enterprise world, authorization then becomes the decision of whether we're allowed to do something. First, we authenticate and then we authorize. The standards built around API security don't assume authentication and go straight to authorization based on some sort of token. It's not assumed that the caller has to be identified. For instance, when you use a physical key to open a door, the door doesn't know who you are, only that you have the right key. This terminology can become very confusing, so don't feel bad if you get a bit lost. You're in good company!

`id_token` is self-contained; everything the API server needs to know about you is in that token. The API server verifies `id_token` using the certificate provided by the identity provider and verifies that the token hasn't expired. As long as that all lines up, the API server will move on to authorizing your request based on its own RBAC configuration. We'll cover the details of that process later. Finally, assuming you're authorized, the API server provides a response.

Notice that Kubernetes never sees your password or any other secret information that you, and only you, know. The only thing that's shared is the `id_token`, and that's ephemeral. This leads to several important points:

- Since Kubernetes never sees your password or other credentials, it can't compromise them. This can save you a tremendous amount of time working with your security team, because all the tasks and controls related to securing passwords can be skipped!
- The `id_token` is self-contained, which means that if it's compromised, there is nothing you can do, short of re-keying your identity provider, to stop it from being abused. This is why it's so important for your `id_token` to have a short lifespan. At 1-2 minutes, the likelihood that an attacker will be able to obtain an `id_token`, realize what it is, and abuse it is very low.

If, while performing its calls, `kubectl` finds that `id_token` has expired, it will attempt to refresh it by calling the IdP's token endpoint using `refresh_token`. If the user's session is still valid, the IdP will generate a new `id_token` and `refresh_token`, which `kubectl` will store for you in the `kubectl` configuration. This happens automatically with no user intervention. Additionally, a `refresh_token` has a one-time use, so if someone tries to use a previously used `refresh_token`, your IdP will fail the refresh process.

It's bound to happen. Someone may need to be locked out immediately. It may be that they're being walked out or that their session has been compromised. This is dependent on your IdP, so when choosing an IdP, make sure it supports some form of session revocation.

Finally, if the `refresh_token` has expired or the session has been revoked, the API server will return a `401 Unauthorized` message to indicate that it will no longer support the token.

We've spent a considerable amount of time examining the OIDC protocol. Now, let's take an in-depth look at the `id_token`.

id_token

An `id_token` is a JSON web token that is Base64-encoded and is digitally signed. The JSON contains a series of attributes, known as claims, in OIDC. There are some standard claims in the `id_token`, but for the most part, the claims you will be most concerned with are as follows:

- `iss`: The issuer, which MUST line up with the issuer in your `kubectl` configuration
- `aud`: Your client ID
- `sub`: Your unique identifier

- `groups`: Not a standard claim, but should be populated with groups specifically related to your Kubernetes deployment

> Many deployments attempt to identify you by your email address. This is an anti-pattern as your email address is generally based on your name, and names change. The sub claim is supposed to be a unique identifier that is immutable and will never change. This way, it doesn't matter if your email changes because your name changes. This can make it harder to debug "who is cd25d24d-74b8-4cc4-8b8c-116bf4abbd26?" but will provide a cleaner and easier-to-maintain cluster.

There are several other claims that indicate when an `id_token` should no longer be accepted. These claims are all measured in seconds from epoch (January 1, 1970) UTC time:

- `exp`: When the `id_token` expires
- `iat`: When the `id_token` was created
- `nbf`: The absolute earliest an `id_token` should be allowed

Why doesn't a token just have a single expiration time?

It's unlikely that the clock on the system that created the `id_token` has the exact same time as the system that is evaluating it. There's often a skew and, depending on how the clock is set, it may be a few minutes. Having a not-before in addition to an expiration gives some room for standard time deviation.

There are other claims in an `id_token` that don't really matter but are there for additional context. Examples include your name, contact information, organization, and so on.

While the primary use for tokens is to interact with the Kubernetes API server, they are not limited to only API interaction. In addition to going to the API server, webhook calls may also receive your `id_token`.

You may have deployed OPA as a validating webhook on a cluster. When someone submits a pod creation request, the webhook will receive the user's `id_token`, which can be used for other decisions.

> OPA, the Open Policy Agent, is a tool for validating and authorizing requests. It's often deployed in Kubernetes as an admission controller webhook. If you haven't worked with OPA or admission controllers, we cover both in depth starting in *Chapter 8, Extending Security Using Open Policy Agent*.

One example is that you want to ensure that the PVCs are mapped to specific PVs based on the submitter's organization. The organization is included in the `id_token`, which is passed to Kubernetes, and then onto the OPA webhook. Since the token has been passed to the webhook, the information can then be used in your OPA policies.

Other authentication options

In this section, we focused on OIDC and presented reasons why it's the best mechanism for authentication. It is certainly not the only option, and we will cover the other options in this section and when they're appropriate.

Certificates

This is generally everyone's first experience authenticating to a Kubernetes cluster.

Once a Kubernetes installation is complete, a pre-built `kubectl config` file that contains a certificate and private key is created and ready to be used. Where this file is created is dependent on the distribution. This file should only be used in "break glass in case of emergency" scenarios, where all other forms of authentication are not available. It should be controlled by your organization's standards for privileged access. When this configuration file is used, it doesn't identify the user and can easily be abused since it doesn't allow for an easy audit trail.

While this is a standard use case for certificate authentication, it's not the only use case for certificate authentication. Certificate authentication, when done correctly, is one of the strongest recognized credentials in the industry.

Certificate authentication is used by the US Federal Government for its most important tasks. At a high level, certificate authentication involves using a client key and certificate to negotiate your HTTPS connection to the API server. The API server can get the certificate you used to establish the connection and validate it against a **Certificate Authority** (**CA**) certificate. Once verified, it maps attributes from the certificate to a user and groups the API server can recognize.

To get the security benefits of certificate authentication, the private key needs to be generated on isolated hardware, usually in the form of a smartcard, and never leave that hardware. A certificate signing request is generated and submitted to a CA that signs the public key, thus creating a certificate that is then installed on the dedicated hardware. At no point does the CA get the private key, so even if the CA were compromised, you couldn't gain the user's private key. If a certificate needs to be revoked, it's added to a revocation list that can either be pulled from an LDAP directory or a file, or it can be checked using the OCSP protocol.

This may look like an attractive option, so why shouldn't you use certificates with Kubernetes?

- Smartcard integration uses a standard called PKCS11, which neither `kubectl` nor the API server support
- The API server has no way of checking certificate revocation lists or using OCSP, so once a certificate has been minted, there's no way to revoke it so that the API server can use it

Additionally, the process to correctly generate a key pair is rarely used. It requires a complex interface to be built that is difficult for users to use combined with command-line tools that need to be run. To get around this, the certificate and key pair are generated for you and you download them or they're emailed to you, negating the security of the process.

The other reason you shouldn't use certificate authentication for users is that it's difficult to leverage groups. While you can embed groups into the subject of the certificate, you can't revoke a certificate. So, if a user's role changes, you can give them a new certificate but you can't keep them from using the old one. While you could reference users directly in your RoleBindings and ClusterRoleBindings, this is an anti-pattern that will make it difficult to keep track of access across even small clusters.

As stated in the introduction to this section, using a certificate to authenticate in "break glass in case of emergency" situations is a good use of certificate authentication. It may be the only way to get into a cluster if all other authentication methods are experiencing issues.

Service accounts

Service accounts appear to provide an easy access method. Creating them is easy. The following command creates a service account object and a secret to go with it that stores the service account's token:

```
kubectl create sa mysa -n default
```

Next, the following command will retrieve the service account's token in JSON format and return only the value of the token. This token can then be used to access the API server:

```
kubectl get secret $(kubectl get sa mysa -n default -o json | jq -r
'.secrets[0].name') -o json | jq -r '.data.token' | base64 -d
```

To show an example of this, let's call the API endpoint directly, without providing any credentials (make sure you use the port for your own local control plane):

```
curl -v --insecure https://0.0.0.0:32768/api
```

You will receive the following:

```
.
.
.
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {
  },
  "status": "Failure",
  "message": "forbidden: User \"system:anonymous\" cannot get path
\"/api\"",
  "reason": "Forbidden",
  "details": {
  },
  "code": 403
* Connection #0 to host 0.0.0.0 left intact
```

By default, most Kubernetes distributions do not allow anonymous access to the API server, so we receive a *403 error* because we didn't specify a user.

Now, let's add our service account to an API request:

```
export KUBE_AZ=$(kubectl get secret $(kubectl get sa mysa -n default -o
json | jq -r '.secrets[0].name') -o json | jq -r '.data.token' | base64
-d)
curl  -H "Authorization: Bearer $KUBE_AZ" --insecure
https://0.0.0.0:32768/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "172.17.0.3:6443"
    }
    ]
  }
```

Success! This was an easy process, so you may be wondering, "Why do I need to worry about all the complicated OIDC mess?" This solution's simplicity brings multiple security issues:

- **Secure transmission of the token**: Service accounts are self-contained and need nothing to unlock them or verify ownership, so if a token is taken in transit, you have no way of stopping its use. You could set up a system where a user logs in to download a file with the token in it, but you now have a much less secure version of OIDC.
- **No expiration**: When you decode a legacy service account token, there is nothing that tells you when the token expires. That's because the token never expires. You can revoke a token by deleting the service account and recreating it, but that means you need a system in place to do that. Again, you've built a less capable version of OIDC.
- **Auditing**: The service account can easily be handed out by the owner once the key has been retrieved. If there are multiple users using a single key, it becomes very difficult to audit the use of the account.

In addition to these issues, you can't put a service account into arbitrary groups. This means that RBAC bindings have to either be direct to the service account or use one of the pre-built groups that service accounts are a member of. We'll explore why this is an issue when we talk about authorization, so just keep it in mind for now.

Finally, service accounts were never designed to be used outside of the cluster. It's like using a hammer to drive in a screw. With enough muscle and aggravation, you will drive it in, but it won't be pretty and no one will be happy with the result.

TokenRequest API

The `TokenRequest` API is the future of service account integration. It went into beta in 1.12 and will be GA in 1.22. This API eliminates the use of static legacy service accounts and instead projects accounts into your pods. These projected tokens are short-lived and unique for each individual pod. Finally, these tokens become invalid once the pods they're associated with are destroyed. This makes service account tokens embedded into a pod much more secure.

This API provides another great feature: you can use it with third-party services. One example is using HashiCorp's Vault secret management system to authenticate pods without having to do a token review API call against the APIs server to validate it.

This feature makes it much easier, and more secure, for your pods to call external APIs.

The `TokenRequest` API lets you request a short-lived service account for a specific scope. While it provides slightly better security since it will expire and has a limited scope, it's still bound to a service account, which means no groups, and there's still the issue of securely getting the token to the user and auditing its use.

Starting in 1.21, all service account tokens are projected into pods via the `TokenRequest` API by default. The new tokens are good for a year though, so not very short-lived! That said, even if a token is set up to expire quickly, the API server won't reject it. It will log that someone is using an expired token. This is intended to make the transition from unlimited-life tokens to short-lived tokens easier.

Some people may be tempted to use tokens for user authentication; however, tokens generated by the `TokenRequest` API are still built for pods to talk to your cluster or for talking to third-party APIs; they are not meant to be used by users.

Custom authentication webhooks

If you already have an identity platform that doesn't use an existing standard, a custom authentication webhook will let you integrate it without having to customize the API server. This feature is commonly used by cloud providers who host managed Kubernetes instances.

You can define an authentication webhook that the API server will call with a token to validate it and get information about the user. Unless you manage a public cloud with a custom IAM token system that you are building a Kubernetes distribution for, don't do this. Writing your own authentication is like writing your own encryption – just don't do it. Every custom authentication system we've seen for Kubernetes boils down to either a pale imitation of OIDC or "pass the password." Much like the analogy of driving a screw in with a hammer, you could do it, but it will be very painful. This is mostly because instead of driving the screw through a board, you're more likely to drive it into your own foot.

Keystone

Those familiar with OpenStack will recognize the name Keystone as an identity provider. If you are not familiar with Keystone, it is the default identity provider used in an OpenStack deployment.

Keystone hosts the API that handles authentication and token generation. OpenStack stores users in Keystone's database.

While using Keystone is more often associated with OpenStack, Kubernetes can also be configured to use Keystone for username and password authentication, with some limitations:

- The main limitation of using Keystone as an IdP for Kubernetes is that it only works with Keystone's LDAP implementation. While you could use this method, you should consider that only username and password are supported, so you're creating an identity provider with a non-standard protocol to authenticate to an LDAP server, which pretty much any OIDC IdP can do out of the box.
- You can't leverage SAML or OIDC with Keystone, even though Keystone supports both protocols for OpenStack, which limits how users can authenticate, thus cutting you off from multiple multi-factor options.
- Few, if any, applications know how to use the Keystone protocol outside of OpenStack. Your cluster will have multiple applications that make up your platform, and those applications won't know how to integrate with Keystone.

Using Keystone is certainly an appealing idea, especially if you're deploying on OpenStack, but ultimately, it's very limiting and you will likely put in just as much work getting Keystone integrated as just using OIDC.

The next section will take the details we've explored here and apply them to integrating authentication into a cluster. As you move through the implementation, you'll see how `kubectl`, the API server, and your identity provider interact to provide secure access to the cluster. We'll tie these features back to common enterprise requirements to illustrate why the details for understanding the OpenID Connect protocol are important.

Configuring KinD for OpenID Connect

For our example deployment, we will use a scenario from our customer, FooWidgets. FooWidgets has a Kubernetes cluster that they would like integrated using OIDC. The proposed solution needs to address the following requirements:

- Kubernetes must use our central authentication system, Active Directory
- We need to be able to map Active Directory groups into our RBAC `RoleBinding` objects
- Users need access to the Kubernetes Dashboard
- Users need to be able to use the CLI
- All enterprise compliance requirements must be met

- Additional cluster management applications need to be managed centrally as well

Let's explore each of these in detail and explain how we can address the customer's requirements.

Addressing the requirements

Our enterprise's requirements require multiple moving parts, both inside and outside our cluster. We'll examine each of these components and how they relate to building an authenticated cluster.

Using LDAP and Active Directory with Kubernetes

Most enterprises today use Active Directory from Microsoft™ to store information about users and their credentials. Depending on the size of your enterprise, it's not unusual to have multiple domains or forests where users' data is stored. We'll need a solution that knows how to talk to each domain. Your enterprise may have one of many tools and products for OpenID Connect integration, or you may just want to connect via LDAP. **LDAP**, the **Lightweight Directory Access Protocol**, is a standard protocol that has been used for over thirty years and is still the standard way to talk directly to Active Directory. Using LDAP, you can look up users and validate their passwords. It's also the simplest way to start because it doesn't require integration with an identity provider. All you need is a service account and credentials!

For FooWidgets, we're going to connect directly to our Active Directory for all authentication.

Don't worry – you don't need Active Directory ready to go to run this exercise. We'll walk through deploying a demo directory into our KinD cluster.

Mapping Active Directory groups to RBAC RoleBindings

This will become important when we start talking about authorization. Active Directory lists all the groups a user is a member of in the `memberOf` attribute. We can read this attribute directly from our logged-in user's account to get their groups. These groups will be embedded into our `id_token` in the `groups` claim and can be referenced directly in RBAC bindings.

Kubernetes Dashboard access

The dashboard is a powerful way to quickly access information about your cluster and make quick updates. When deployed correctly, the dashboard does not create any security issues. The proper way to deploy the dashboard is with no privileges, instead relying on the user's own credentials. We'll do this with a reverse proxy that injects the user's OIDC token on each request, which the dashboard will then use when it makes calls to the API server. Using this method, we'll be able to constrain access to our dashboard the same way we would with any other web application.

There are a few reasons why using the `kubectl` built-in proxy and port-forward aren't a great strategy for accessing the dashboard. Many enterprises will not install CLI utilities locally, forcing you to use a jump box to access privileged systems such as Kubernetes, meaning port forwarding won't work. Even if you can run `kubectl` locally, opening a port on loopback (`127.0.0.1`) means anything on your system can use it, not just you from your browser. While browsers have controls in place to keep you from accessing ports on loopback using a malicious script, that won't stop anything else on your workstation. Finally, it's just not a great user experience.

We'll dig into the details of how and why this works in *Chapter 7*, *Deploying a Secured Kubernetes Dashboard*.

Kubernetes CLI access

Most developers want to be able to access `kubectl` and other tools that rely on the `kubectl` configuration. For instance, the Visual Studio Code Kubernetes plugin doesn't require any special configuration. It just uses the `kubectl` built-in configuration. Most enterprises tightly constrain what binaries you're able to install, so we want to minimize any additional tools and plugins we want to install.

Enterprise compliance requirements

Being cloud-native doesn't mean you can ignore your enterprise's compliance requirements. Most enterprises have requirements such as having 20-minute idle timeouts, may require multi-factor authentication for privileged access, and so on. Any solution we put in place has to make it through the control spreadsheets needed to go live. Also, this goes without saying, but everything needs to be encrypted (and I do mean everything).

Pulling it all together

To fulfill these requirements, we're going to use OpenUnison. It has prebuilt configurations to work with Kubernetes, the Dashboard, the CLI, and Active Directory.

It's also pretty quick to deploy, so we don't need to concentrate on provider-specific implementation details and instead focus on Kubernetes' configuration options. Our architecture will look like this:

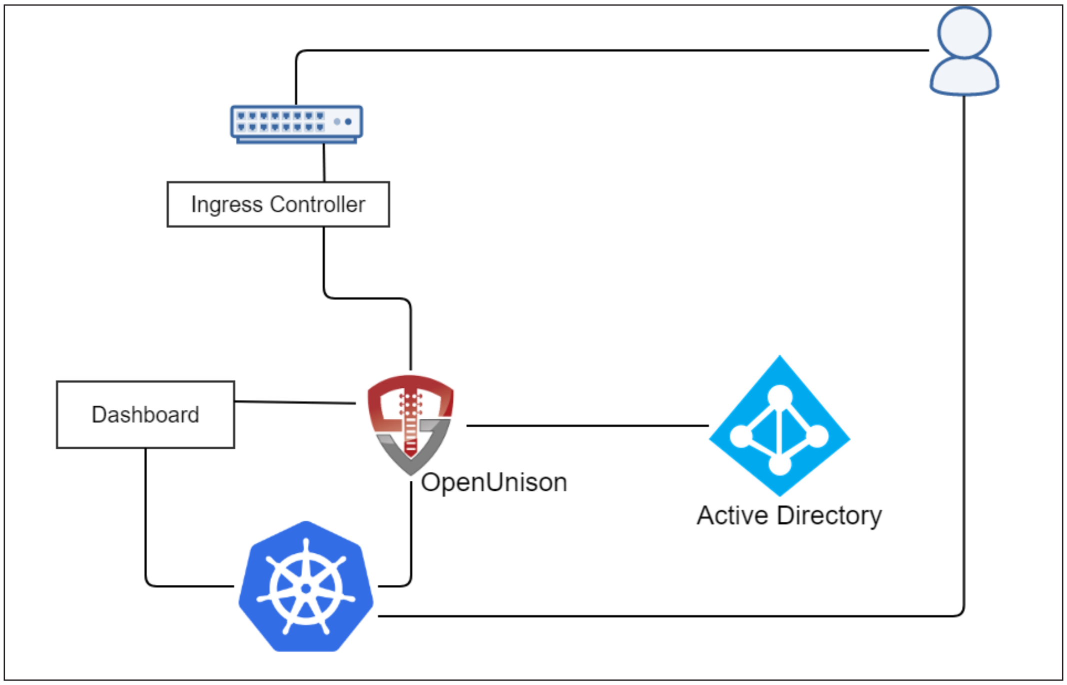


Figure 5.2: Authentication architecture

For our implementation, we're going to use two hostnames:

- `k8s.apps.X-X-X-X.nip.io`: Access to the OpenUnison portal, where we'll initiate our login and get our tokens
- `k8sdb.apps.X-X-X-X.nip.io`: Access to the Kubernetes dashboard

> As a quick refresher, `nip.io` is a public DNS service that will return an IP address from the one embedded in your hostname. This is really useful in a lab environment where setting up DNS can be painful. In our examples, `X-X-X-X` is the IP of your Docker host.

When a user attempts to access `https://k8s.apps.X-X-X-X.nip.io/`, they'll be asked for their username and password. After the user hits submit, OpenUnison will look up the user against Active Directory, retrieving the user's profile information. At that point, OpenUnison will create user objects in the OpenUnison namespace to store the user's information and create OIDC sessions.

Earlier, we described how Kubernetes doesn't have user objects. Kubernetes lets you extend the base API with **Custom Resource Definitions** (**CRDs**). OpenUnison defines a User CRD to help with high availability and to avoid needing a database to store state in. These user objects can't be used for RBAC.

Once the user is logged into OpenUnison, they can get their `kubectl` configuration to use the CLI or use the Kubernetes Dashboard, `https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/`, to access the cluster from their browser. Once the user is ready, they can log out of OpenUnison, which will end their session and invalidate their `refresh_token`, making it impossible for them to use `kubectl` or the dashboard until after they log in again. If they walk away from their desk for lunch without logging out, when they return, their `refresh_token` will have expired, so they'll no longer be able to interact with Kubernetes without logging back in.

Now that we have walked through how users will log in and interact with Kubernetes, we'll deploy OpenUnison and integrate it into the cluster for authentication.

Deploying OpenUnison

The Dashboard is a popular feature for many users. It provides a quick view of resources without needing to use the `kubectl` CLI. Over the years, it has received some bad press for being insecure, but when deployed correctly, it is very secure. Most of the stories you may have read or heard about come from a Dashboard deployment that was not set up correctly. We will cover this topic in *Chapter 7*, *Deploying a Secured Kubernetes Dashboard*:

1. First, we'll deploy the dashboard from `https://github.com/kubernetes/dashboard`:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/
dashboard/v2.2.0/aio/deploy/recommended.yaml

namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard
created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard
```

```
created
clusterrolebinding.rbac.authorization.k8s.io/Kubernetes
dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```

2. Second, deploy our testing "Active Directory." This is an ApacheDS instance that has the same schema as Active Directory, so you'll be able to see how Kubernetes would integrate with Active Directory without needing to deploy it yourself! After running the following `kubectl` command, wait until the pod is running in the `activedirectory` namespace:

   ```
   kubectl create -f chapter5/apacheds.yaml
   ```

3. Next, we need to add the repository that contains OpenUnison to our Helm list. To add the Tremolo chart repository, use the `helm repo add` command:

   ```
   helm repo add tremolo https://nexus.tremolo.io/repository/helm/
   https://nexus.tremolo.io/repository/helm/"tremolo" has been
   added to your repositories
   ```

4. Once added, you need to update the repository using the `helm repo update` command:

   ```
   helm repo update
   Hang tight while we grab the latest from your chart
   repositories...
   ...Successfully got an update from the "tremolo" chart
   repository
   Update Complete. Happy Helming!
   ```

   You are now ready to deploy the OpenUnison operator using the Helm chart.

5. First, we want to deploy OpenUnison in a new namespace called `openunison`. We need to create the namespace before deploying the Helm chart:

   ```
   kubectl create ns openunison
   ```

6. Next, we need to add a ConfigMap that will tell OpenUnison how to talk to our "Active Directory":

   ```
   kubectl create -f chapter5/myvd-book.yaml
   ```

7. With the namespace created, you can deploy the chart into the namespace using Helm. To install a chart using Helm, use helm install <name> <chart> <options>:

   ```
   helm install openunison tremolo/openunison-operator --namespace
   ```

```
openunison
NAME: openunison
LAST DEPLOYED: Fri Apr 17 15:04:50 2020
NAMESPACE: openunison
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

The operator will take a few minutes to finish deploying.

> An operator is a concept that was pioneered by CoreOS with the goal of encapsulating many of the tasks an administrator may perform that can be automated. Operators are implemented by watching for changes to a specific CRD and acting accordingly. The OpenUnison operator looks for objects of the OpenUnison type and will create any objects that are needed. A secret is created with a PKCS12 file; Deployment, Service, and Ingress objects are all created too. As you make changes to an OpenUnison object, the operator makes updates to the Kubernetes object as needed. For instance, if you change the image in the OpenUnison object, the operator updates the Deployment, which triggers Kubernetes to roll out new pods. For SAML, the operator also watches metadata so that if it changes, the updated certificates are imported.

8. Once the operator has been deployed, we need to create a secret that will store passwords used internally by OpenUnison. Make sure to use your own values for the keys in this secret (remember to Base64-encode them):

```
kubectl create -f - <<EOF
apiVersion: v1
type: Opaque
metadata:
   name: orchestra-secrets-source
   namespace: openunison
data:
   K8S_DB_SECRET: c3RhcnQxMjM=
   unisonKeystorePassword: cGFzc3dvcmQK
   AD_BIND_PASSWORD: c3RhcnQxMjM=
kind: Secret
EOF
secret/orchestra-secrets-source created
```

From here on out, we'll assume you're using the testing "Active Directory." You can customize values and examples, though, to work with your own Active Directory if you want.

9. Now, we need to create a `values.yaml` file that will be used to supply configuration information when we deploy OpenUnison. This book's GitHub repository contains the file to customize in `chapter5/openunison-values.yaml`:

```
network:
  openunison_host: "k8sou.apps.XX-XX-XX-XX.nip.io"
  dashboard_host: "k8sdb.apps.XX-XX-XX-XX.nip.io"
  api_server_host: ""
  session_inactivity_timeout_seconds: 900
  k8s_url: https://0.0.0.0:6443
```

   You need to change the following values for your deployment:

   - Network: `openunison_host`: This value should use the IP address of your cluster, which is the IP address of your Docker host; for example, `k8sou.apps.192-168-2-131.nip.io`
   - Network: `dashboard_host`: This value should also use the IP address of your cluster, which is the IP address of your Docker host; for example, `k8sdb.apps.192-168-2-131.nip.io`

   After you've edited or created the file using your own entries, save the file and move on to deploying OpenUnison provider

10. To deploy OpenUnison using your `openunison-values.yaml` file, execute a `helm install` command that uses the `-f` option to specify the `openunison-values.yaml` file:

```
helm install orchestra tremolo/orchestra --namespace openunison
-f ./chapter5/openunison-values.yaml
NAME: orchestra
LAST DEPLOYED: Tue Jul 6 16:20:00 2021
NAMESPACE: openunison
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

11. In a few minutes, OpenUnison will be up and running. Check the deployment status by getting the pods in the `openunison` namespace:

```
kubectl get pods -n openunison
NAME                                    READY   STATUS    RESTARTS
AGE
openunison-operator-858d496-zzvvt       1/1     Running   0
5d6h
openunison-orchestra-57489869d4-88d2v   1/1     Running   0
85s
```

12. The last step is to deploy our portal configuration:

```
helm install orchestra-login-portal tremolo/orchestra-login-
portal --namespace openunison -f ./chapter5/openunison-values.
yaml
NAME: orchestra-login-portal
LAST DEPLOYED: Tue Jul 6 16:22:00 2021
NAMESPACE: openunison
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

You can log into the OIDC provider using any machine on your network by using the assigned nip.io address. Since we will test access using the dashboard, you can use any machine with a browser. Navigate your browser to `network.openunison_host` in your `openunison-values.yaml` file. When prompted, use the username `mmosley` and the password `start123` and click on **Sign in**.

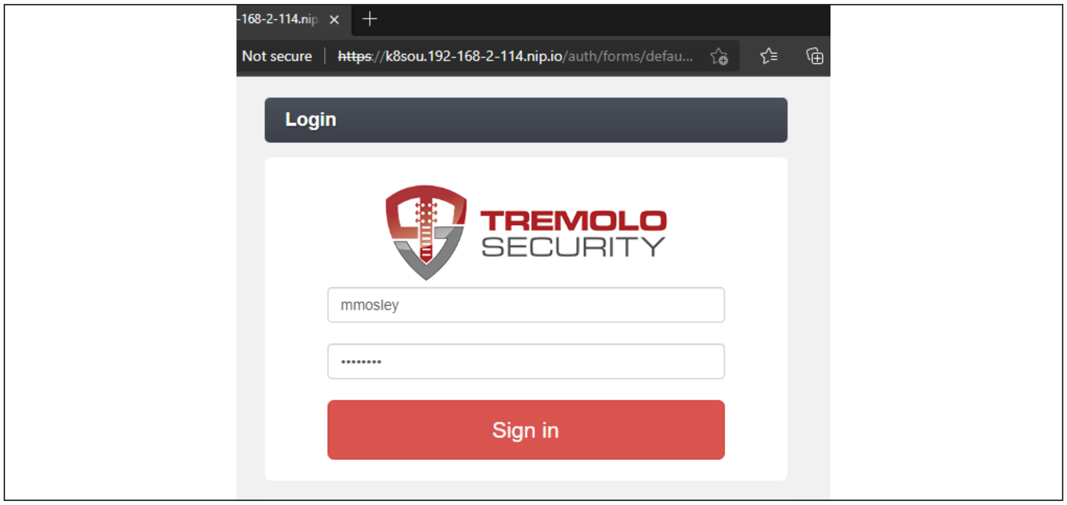


Figure 5.3: OpenUnison login screen

When you do, you'll see this screen:

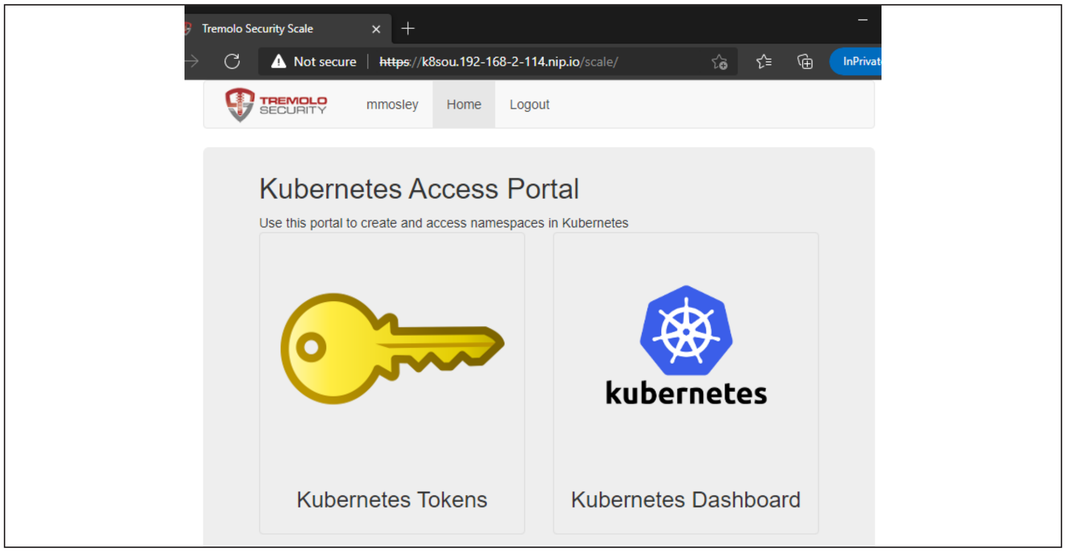


Figure 5.4: OpenUnison home screen

Let's test the OIDC provider by clicking on the **Kubernetes Dashboard** link. Don't panic when you look at the initial dashboard screen – you'll see something like the following:

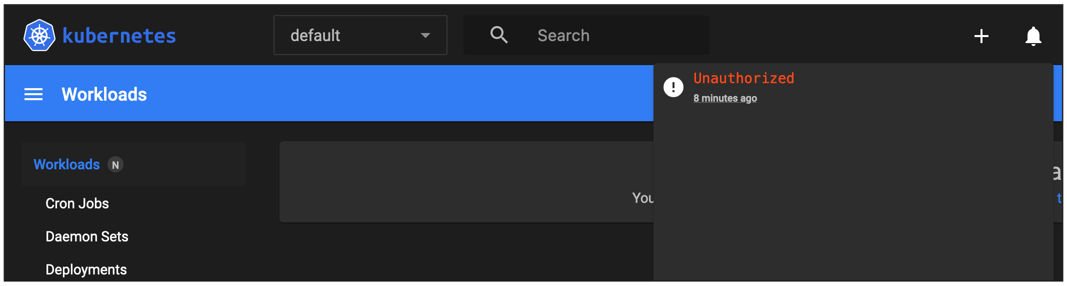


Figure 5.5: Kubernetes Dashboard before SSO integration has been completed with the API server

That looks like a lot of errors! We're in the dashboard, but nothing seems to be authorized. That's because the API server doesn't trust the tokens that have been generated by OpenUnison yet. The next step is to tell Kubernetes to trust OpenUnison as its OpenID Connect Identity Provider.

Configuring the Kubernetes API to use OIDC

At this point, you have deployed OpenUnison as an OIDC provider and it's working, but your Kubernetes cluster has not been configured to use it as a provider yet.

To configure the API server to use an OIDC provider, you need to add the OIDC options to the API server and provide the OIDC certificate so that the API will trust the OIDC provider.

Since we are using KinD, we can add the required options using a few `kubectl` and `docker` commands.

To provide the OIDC certificate to the API server, we need to retrieve the certificate and copy it over to the KinD master server. We can do this using two commands on the Docker host:

1. The first command extracts OpenUnison's TLS certificate from its secret. This is the same secret referenced by OpenUnison's Ingress object. We use the `jq` utility to extract the data from the secret and then Base64-decode it:

   ```
   kubectl get secret ou-tls-certificate -n openunison -o json | jq
   -r '.data["tls.crt"]' | base64 -d > ou-ca.pem
   ```

2. The second command will copy the certificate to the master server into the `/etc/kubernetes/pki` directory:

   ```
   docker cp ou-ca.pem cluster01-control-plane:/etc/kubernetes/pki/
   ou-ca.pem
   ```

3. As we mentioned earlier, to integrate the API server with OIDC, we need to have the OIDC values for the API options. To list the options we will use, describe the `api-server-config` ConfigMap in the `openunison` namespace:

   ```
   kubectl describe configmap api-server-config -n openunison
   Name:         api-server-config
   Namespace:    openunison
   Labels:       <none>
   Annotations:  <none>
   Data
   ====
   oidc-api-server-flags:
   ----
   --oidc-issuer-url=https://k8sou.apps.192-168-2-131.nip.io/auth/
   idp/k8sIdp
   --oidc-client-id=Kubernetes
   --oidc-username-claim=sub
   --oidc-groups-claim=groups
   --oidc-ca-file=/etc/kubernetes/pki/ou-ca.pem
   ```

4. Next, edit the API server configuration. OpenID Connect is configured by changing flags on the API server. This is why managed Kubernetes generally doesn't offer OpenID Connect as an option, but we'll cover that later in this chapter. Every distribution handles these changes differently, so check with your vendor's documentation. For KinD, shell into the control plane and update the manifest file:

```
docker exec -it cluster01-control-plane bash
apt-get update
apt-get install vim
vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

5. Add the flags from the output of the ConfigMap under `command`. Make sure to add spacing and a dash (-) in front. It should look something like this when you're done:

```
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Ho
stname
    - --oidc-issuer-url=https://k8sou.apps.192-168-2-131.nip.io/
auth/idp/k8sIdp
    - --oidc-client-id=Kubernetes
    - --oidc-username-claim=sub
    - --oidc-groups-claim=groups
    - --oidc-ca-file=/etc/kubernetes/pki/ou-ca.pem
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-
client.crt
```

6. Exit vim and the Docker environment (*Ctrl* + *D*) and then take a look at the `api-server` pod:

```
kubectl get pod kube-apiserver-cluster01-control-plane -n kube-
system
NAME                                       READY   STATUS
RESTARTS  AGE
kube-apiserver-cluster-auth-control-plane  1/1     Running      0
73s
```

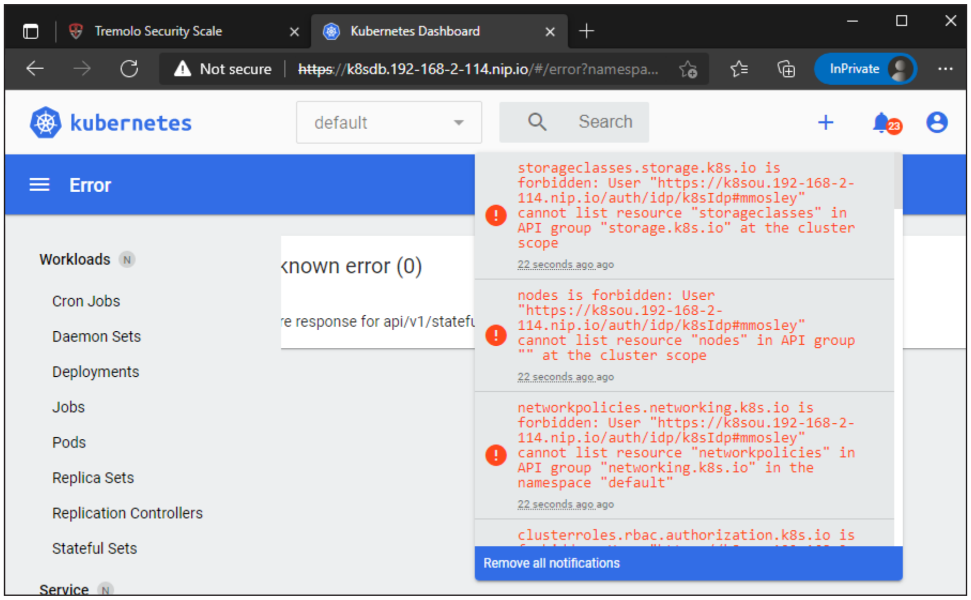
Notice that it's only `73s` old. That's because KinD saw that there was a change in the manifest and restarted the API server.

> The API server pod is known as a static pod. This pod can't be changed directly; its configuration has to be changed from the manifest on disk. This gives you a process that's managed by the API server as a container, but without giving you a situation where you need to edit pod manifests in etcd directly if something goes wrong.

Verifying OIDC integration

Once OpenUnison and the API server have been integrated, we need to test that the connection is working:

1. To test the integration, log back into OpenUnison and click on the **Kubernetes Dashboard** link again.
2. Click on the bell in the upper right and you'll see a different error:



known error (0)

e response for api/v1/statefu

```
storageclasses.storage.k8s.io is
forbidden: User "https://k8sou.192-168-2-
114.nip.io/auth/idp/k8sIdp#mmosley"
cannot list resource "storageclasses" in
API group "storage.k8s.io" at the cluster
scope
```
22 seconds ago ago

```
nodes is forbidden: User
"https://k8sou.192-168-2-
114.nip.io/auth/idp/k8sIdp#mmosley"
cannot list resource "nodes" in API group
"" at the cluster scope
```
22 seconds ago ago

```
networkpolicies.networking.k8s.io is
forbidden: User "https://k8sou.192-168-2-
114.nip.io/auth/idp/k8sIdp#mmosley"
cannot list resource "networkpolicies" in
API group "networking.k8s.io" in the
namespace "default"
```
22 seconds ago ago

```
clusterroles.rbac.authorization.k8s.io is
```

Figure 5.6: SSO enabled but the user is not authorized to access any resources

SSO between OpenUnison and Kubernetes is working! However, the new error, `service is forbidden: User https://...`, is an authorization error, **not** an authentication error. The API server knows who we are, but isn't letting us access the APIs.

3. We'll dive into the details of RBAC and authorizations in the next chapter, but for now, create this RBAC binding:

```
kubectl create -f - <<EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

```
metadata:
   name: ou-cluster-admins
subjects:
- kind: Group
  name: cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
EOF
clusterrolebinding.rbac.authorization.k8s.io/ou-cluster-admins
created
```

4. Finally, go back to the Dashboard and you'll see that you have full access to your cluster and all the error messages are gone.

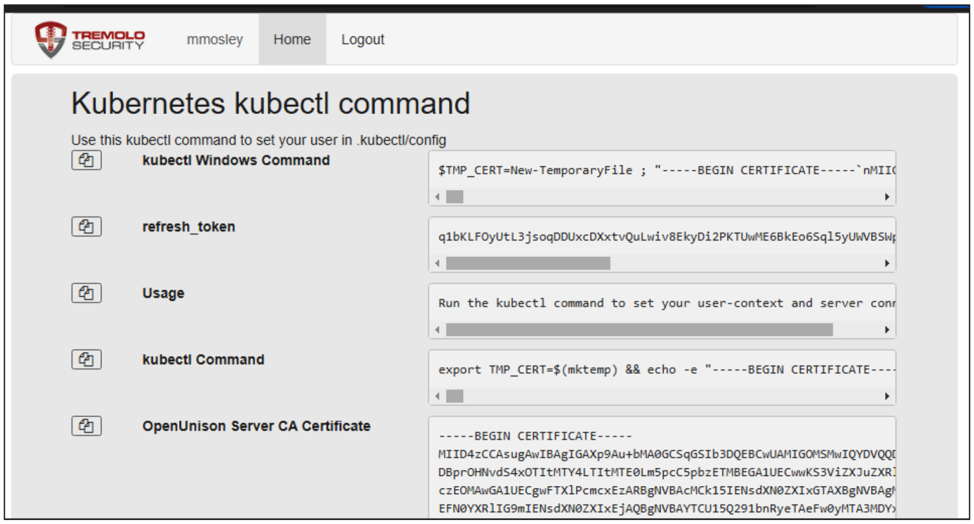
The API server and OpenUnison are now connected. Additionally, an RBAC policy has been created to enable our test user to manage the cluster as an administrator. Access was verified by logging into the Kubernetes dashboard, but most interactions will take place using the `kubectl` command. The next step is to verify we're able to access the cluster using `kubectl`.

Using your tokens with kubectl

> This section assumes you have a machine on your network that has a browser and `kubectl` running.

Using the Dashboard has its use cases, but you will likely interact with the API server using `kubectl`, rather than the Dashboard, for the majority of your day. In this section, we will explain how to retrieve your JWT and how to add it to your Kubernetes config file:

1. You can retrieve your token from the OpenUnison dashboard. Navigate to the OpenUnison home page and click on the key that says **Kubernetes Tokens**. You'll see a screen that looks as follows:



Kubernetes kubectl command

Use this kubectl command to set your user in .kubectl/config

kubectl Windows Command
```
$TMP_CERT=New-TemporaryFile ; "-----BEGIN CERTIFICATE-----`nMII
```

refresh_token
```
q1bKLFOyUtL3jsoqDDUxcDXxtvQuLwiv8EkyDi2PKTUwME6BkEo6Sql5yUWVBSW
```

Usage
```
Run the kubectl command to set your user-context and server con
```

kubectl Command
```
export TMP_CERT=$(mktemp) && echo -e "-----BEGIN CERTIFICATE---
```

OpenUnison Server CA Certificate
```
-----BEGIN CERTIFICATE-----
MIID4zCCAsugAwIBAgIGAXp9Au+bMA0GCSqGSIb3DQEBCwUAMIGOMSMwIQYDVQQ
DBprOHNvdS4xOTItMTY4LTItMTE0Lm5pcC5pbzETMBEGA1UECwwKS3ViZXJuZXR
czEOMAwGA1UECgwFTXlPcmcxEzARBgNVBAcMCk15IENsdXN0ZXIxGTAXBgNVBAg
EFN0YXRlIG9mIENsdXN0ZXIxEjAQBgNVBAYTCU15Q291bnRyeTAeFw0yMTA3MDY
```

Figure 5.7: OpenUnison kubectl configuration tool

OpenUnison provides a command line that you can copy and paste into your host session that adds all the required information to your config.

2. First, click on the double documents button next to the `kubectl` command (or `kubectl` Windows command if you're on Windows) to copy your `kubectl` command into your buffer. Leave the web browser open in the background.
3. You may want to back up your original config file before pasting the `kubectl` command from OpenUnison:

```
cp .kube/config .kube/config.bak
export KUBECONFIG=/tmp/k
kubectl get nodes
W0423 15:46:46.924515    3399 loader.go:223] Config not found:
/tmp/k error: no configuration has been provided, try setting
KUBERNETES_MASTER environment variable
```

4. Then, go to your host console and paste the command into the console (the following output has been shortened, but your paste will start with the same output):

```
export TMP_CERT=$(mktemp) && echo -e "-----BEGIN CER. . .
Cluster "kubernetes" set.
Context "kubernetes" modified.
User "mmosley@kubernetes" set.
```

```
Switched to context "kubernetes".
```

5. Now, verify that you can view the cluster nodes using `kubectl get nodes`:

```
kubectl get nodes

NAME                          STATUS   ROLES    AGE   VERSION
cluster-auth-control-plane    Ready    master   60m   v1.21.1
cluster-auth-worker           Ready    <none>   61m   v1.21.1
```

6. You are now using your login credentials instead of the master certificate! As you work, the session will refresh. Log out of OpenUnison and watch the list of nodes. Within a minute or two, your token will expire and no longer work:

```
kubectl get nodes
Unable to connect to the server: failed to refresh token:
oauth2: cannot fetch token: 401 Unauthorized
```

Congratulations! You've now set up your cluster so that it does the following:

- Authenticates using LDAP using your enterprise's existing authentication system
- Uses groups from your centralized authentication system to authorize access to Kubernetes (we'll get into the details of how in the next chapter)
- Gives access to your users to both the CLI and the dashboard using the centralized credentials
- Maintains your enterprise's compliance requirements by having short-lived tokens that provide a way to time out
- Everything uses TLS, from the user's browser, to the Ingress Controller, to OpenUnison, the Dashboard, and finally, the API server

Next, you'll learn how to integrate centralized authentication into your managed clusters.

Introducing impersonation to integrate authentication with cloud-managed clusters

It's very popular to use managed Kubernetes services from cloud vendors such as Google, Amazon, Microsoft, and DigitalOcean (among many others).

When it comes to these services, they are generally very quick to get up and running, and they all share a common thread: they mostly don't support OpenID Connect (Amazon's EKS does support OpenID Connect now, but the cluster must be running on a public network and have a commercially signed TLS certificate).

Earlier in this chapter, we talked about how Kubernetes supports custom authentication solutions through webhooks and that you should never, ever, use this approach unless you are a public cloud provider or some other host of Kubernetes systems. It turns out that pretty much every cloud vendor has its own approach to using these webhooks that uses their own identity and access management implementations. In that case, why not just use what the vendor provides? There are several reasons why you may not want to use a cloud vendor's IAM system:

- **Technical**: You may want to support features not offered by the cloud vendor, such as the dashboard, in a secure fashion.
- **Organizational**: Tightly coupling access to managed Kubernetes with that cloud's IAM puts an additional burden on the cloud team, which means that they may not want to manage access to your clusters.
- **User experience**: Your developers and admins may have to work across multiple clouds. Providing a consistent login experience makes it easier on them and requires learning fewer tools.
- **Security and compliance**: The cloud implementation may not offer choices that line up with your enterprise's security requirements, such as short-lived tokens and idle timeouts.

All that being said, there may be reasons to use the cloud vendor's implementation. You'll need to balance out the requirements, though. If you want to continue to use centralized authentication and authorization with hosted Kubernetes, you'll need to learn how to work with Impersonation.

What is Impersonation?

Kubernetes Impersonation is a way of telling the API server who you are without knowing your credentials or forcing the API server to trust an OpenID Connect IdP.

When you use `kubectl`, instead of the API server receiving your `id_token` directly, it will receive a service account or identifying certificate that will be authorized to impersonate users, as well as a set of headers that tell the API server who the proxy is acting on behalf of:

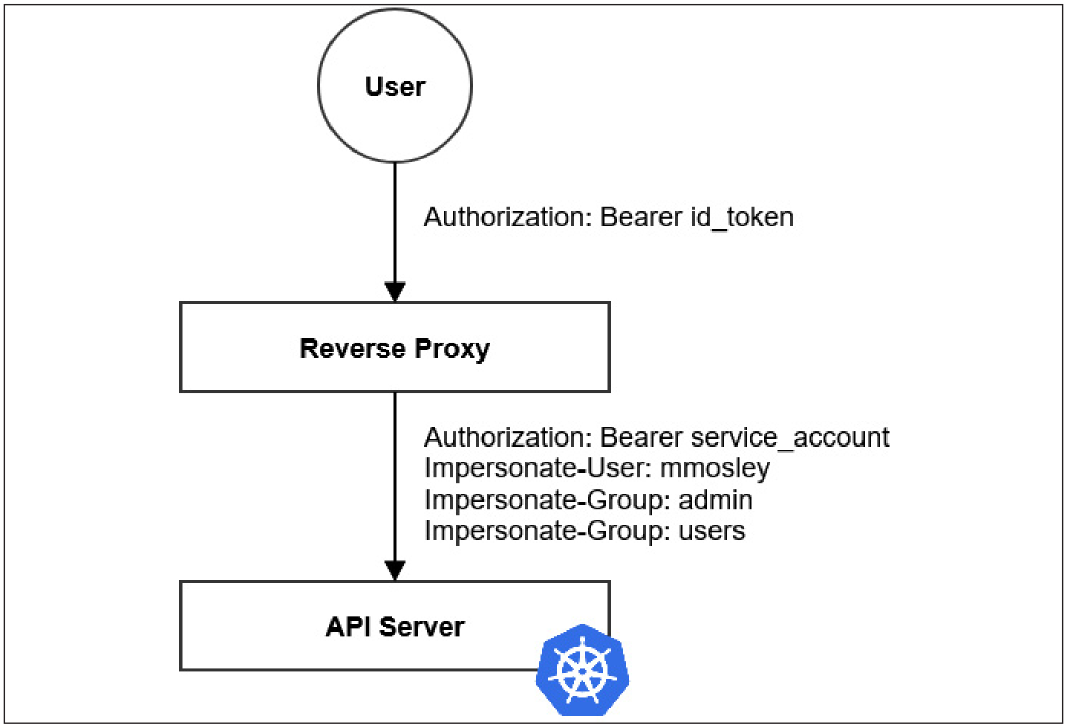

User
Authorization: Bearer id_token
Reverse Proxy
Authorization: Bearer service_account
Impersonate-User: mmosley
Impersonate-Group: admin
Impersonate-Group: users
API Server

Figure 5.8: Diagram of how a user interacts with the API server when using Impersonation

The reverse proxy is responsible for determining how to map from the `id_token`, which the user provides (or any other token, for that matter), to the `Impersonate-User` and `Impersonate-Group` HTTP headers. The dashboard should never be deployed with a privileged identity, which the ability to impersonate falls under.

To allow Impersonation with the 2.x dashboard, use a similar model, but instead of going to the API server, you go to the dashboard:

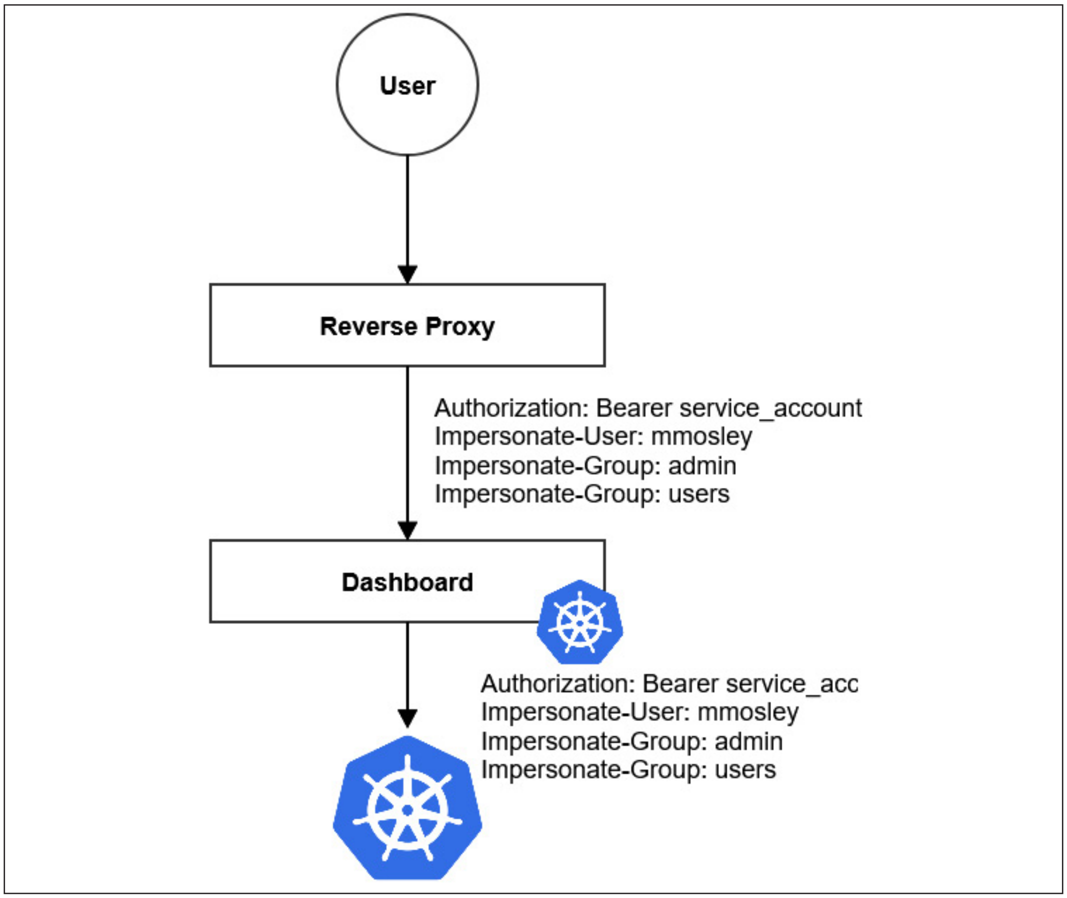


Figure 5.9: Kubernetes Dashboard with Impersonation

The user interacts with the reverse proxy just like any web application. The reverse proxy uses its own service account and adds the impersonation headers. The dashboard passes this information through to the API server on all requests. The dashboard never has its own identity.

Security considerations

The service account has a certain superpower: it can be used to impersonate **anyone** (depending on your RBAC definitions). If you're running your reverse proxy from inside the cluster, a service account is OK, especially if combined with the `TokenRequest` API to keep the token short-lived.

Earlier in the chapter, we talked about the legacy tokens for `ServiceAccount` objects having no expiration. That's important here because if you're hosting your reverse proxy off cluster, then if it were compromised, someone could use that service account to access the API service as anyone. Make sure you're rotating that service account often. If you're running the proxy off cluster, it's probably best to use a shorter-lived certificate instead of a service account.

When running the proxy on a cluster, you want to make sure it's locked down. It should run in its own namespace at a minimum. Not `kube-system` either. You want to minimize the number of people who have access. Using multi-factor authentication to get to that namespace is always a good idea, as is using network policies that control what pods can reach out to the reverse proxy.

Based on the concepts we've just learned about regarding impersonation, the next step is to update our cluster's configuration to use impersonation instead of using OpenID Connect directly. You don't need a cloud-managed cluster to work with impersonation.

Configuring your cluster for impersonation

Let's deploy an impersonating proxy for our cluster. Assuming you're reusing your existing cluster, we'll upgrade our existing orchestra Helm deployment with an updated `openunison-values.yaml` file:

1. First, delete the current TLS secret for OpenUnison since it doesn't have the right configuration for Impersonation. When we update the orchestra Helm chart, the operator will generate a new certificate for us.

   ```
   kubectl delete secret ou-tls-certificate -n openunison
   ```

2. Next, update our Helm chart to use impersonation. Edit the `openunison-values.yaml` file, update `network.api_server_host` as shown in the following snippet, and set `enable_impersonation` to `true`:

   ```
   network:
     openunison_host: "k8sou.apps.192-168-2-131.nip.io"
     dashboard_host: "k8sdb.apps.192-168-2-131.nip.io"
     api_server_host: "k8sapi.apps.192-168-2-131.nip.io"
     session_inactivity_timeout_seconds: 900
     k8s_url: https://192.168.2.131:32776
   enable_impersonation: true
   ```

We have made two changes here:

- Added a host for the API server proxy
- Enabled impersonation

These changes enable OpenUnison's impersonation features and generate an additional RBAC binding to enable impersonation on OpenUnison's service account.

3. Upgrade the orchestra Helm chart with the new `openunison-values.yaml` file:

```
helm upgrade orchestra tremolo/orchestra -n openunison -f
chapter5/openunison-values.yaml
NAME: orchestra
LAST DEPLOYED: Wed Jul  7 02:45:36 2021
NAMESPACE: openunison
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

4. Once the `openunison-orchestra` pod is running, upgrade the `orchestra-login-portal` Helm chart with the new `openunison-values.yaml` file too:

```
helm upgrade orchestra-login-portal tremolo/orchestra-login-
portal -n openunison -f chapter5/openunison-values.yaml
NAME: orchestra-login-portal
LAST DEPLOYED: Wed Jul  7 02:47:03 2021
NAMESPACE: openunison
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

The new OpenUnison deployment is configured as a reverse proxy for the API server and is still integrated with our Active Directory. There are no cluster parameters to set because Impersonation doesn't need any cluster-side configuration. The next step is to test the integration.

Testing Impersonation

Now, let's test our Impersonation setup. Follow these steps:

1. In a browser, enter the URL for your OpenUnison deployment. This is the same URL you used for your initial OIDC deployment.

2. Log into OpenUnison and then click on the dashboard. You should recall that the first time you opened the dashboard on your initial OpenUnison deployment, you received a lot of errors until you created the new RBAC role, which granted access to the cluster.

   After you've enabled impersonation and opened the dashboard, you shouldn't see any error messages, even though you were prompted for new certificate warnings and didn't tell the API server to trust the new certificates you're using with the dashboard.
3. Click on the little circular icon in the upper right-hand corner to see who you're logged in as.
4. Next, go back to the main OpenUnison dashboard and click on the **Kubernetes Tokens** badge.

   Notice that the `--server` flag being passed to `kubectl` no longer has an IP. Instead, it has the hostname from `network.api_server_host` in the `values.yaml` file. This is Impersonation. Instead of interacting directly with the API server, you're now interacting with OpenUnison's reverse proxy.
5. Finally, let's copy and paste our `kubectl` command into a shell:

```
export TMP_CERT=$(mktemp) && echo -e "-----BEGIN CERTIFI...
Cluster "kubernetes" set.
Context "kubernetes" created.
User "mmosley@kubernetes" set.
Switched to context "kubernetes".
```

6. To verify you have access, list the cluster nodes:

```
kubectl get nodes
NAME                         STATUS   ROLES    AGE    VERSION
cluster-auth-control-plane   Ready    master   6h6m   v1.21.1
cluster-auth-worker          Ready    <none>   6h6m   v1.21.1
```

7. Just like when you integrated the original deployment of OpenID Connect, once you've logged out of the OpenUnison page, within a minute or two, the tokens will expire and you won't be able to refresh them:

```
kubectl get nodes
Unable to connect to the server: failed to refresh token:
oauth2: cannot fetch token: 401 Unauthorized
```

You've now validated that your cluster is working correctly with Impersonation. Instead of authenticating directly to the API server, the impersonating reverse proxy (OpenUnison) is forwarding all requests to the API server with the correct impersonation headers. You're still meeting your enterprise's needs by providing both a login and logout process and integrating your Active Directory groups.

You'll also notice that you can now access your cluster from any system on your network! This might make doing the rest of the examples throughout the book easier.

Configuring Impersonation without OpenUnison

The OpenUnison operator automated a couple of key steps to get impersonation working. There are other projects designed specifically for Kubernetes, such as Jetstack's OIDC proxy (`https://github.com/jetstack/kube-oidc-proxy`), that are designed to make using Impersonation easier. You can use any reverse proxy that can generate the correct headers. There are two critical items to understand when doing this on your own.

Impersonation RBAC policies

RBAC will be covered in the next chapter, but for now, the correct policy to authorize a service account for Impersonation is as follows:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: impersonator
rules:
- apiGroups:
  - ""
  resources:
  - users
  - groups
  verbs:
  - impersonate
```

To constrain what accounts can be impersonated, add `resourceNames` to your rule.

Default groups

When impersonating a user, Kubernetes does not add the default group, `system:authenticated`, to the list of impersonated groups. When using a reverse proxy that doesn't specifically know to add the header for this group, configure the proxy to add it manually. Otherwise, simple acts such as calling the `/api` endpoint will fail as this will be unauthorized for anyone except cluster administrators.

We've focused the bulk of this chapter on authenticating users who will interact with the API server. A major advantage of Kubernetes and the APIs it provides is to automate your systems. Next, we'll look at how you apply what we've learned so far to authenticating those automated systems.

Authenticating from pipelines

This chapter so far has focused exclusively on authentication to Kubernetes by users. Whether an operator or a developer, a user will often interact with a cluster to update objects, debug issues, view logs, and so on. This doesn't quite handle all use cases, though. Most Kubernetes deployments are partnered with pipelines, a process by which code is moved from source to binaries to containers and ultimately into a running cluster. We'll cover pipelines in more detail in *Chapter 14*, *Provisioning a Platform*. For now, the main question is "How will your pipeline talk to Kubernetes securely?"

If your pipeline runs in the same cluster as being updated, this is a simple question to answer. You would grant access to the pipeline's service account via RBAC to do what it needs to do. This is why service accounts exist, to provide identity to processes inside the cluster.

What if your pipeline runs outside of the cluster? Kubernetes is an API, and all the options presented in this chapter apply to a pipeline as they would to a user. Service account tokens don't provide an expiration and can easily be abused. The `TokenRequest` API could give you a short-lived token, but you still need to be authenticated to get it. If your cluster is running on the same cloud provider as your pipeline, you may be able to use its integrated IAM system. For instance, you can generate an IAM role in Amazon CodeBuild that can talk to an EKS cluster without having a static service account. The same is true for Azure DevOps and AKS.

If a cloud's IAM capabilities won't cover your needs, there are two options. The first is to dynamically generate a token for a pipeline the same way you would for a user by authenticating to an identity provider and then using the returned `id_token` with your API calls. The second is to generate a certificate that can be used with your API server. Let's look at both options and see how our pipelines can use them.

Using tokens

Kubernetes doesn't distinguish between an API call from a human or a pipeline. A short-lived token is a great way to interact with your API server as a pipeline. Most of the client SDKs for Kubernetes know how to refresh these tokens. The biggest issue is how do you get a token your pipeline can use?

Most enterprises already have some kind of service account management system. Here, the term "service account" is generic and means an account used by a service of some kind instead of being the `ServiceAccount` object in Kubernetes. These service account management systems often have their own way of handling tasks, such as credential rotation and authorization management. They also have their own compliance tools, making it easier to get through your security review processes!

Assuming you have an enterprise service account for your pipeline, how do you translate that credential into a token? We are generating tokens based on credentials in our OIDC integrated identity provider; it would be great to use that from our pipelines too! With OpenUnison, this is pretty easy because the page that gave us our token is just a frontend for an API. The next question to answer is how to authenticate to OpenUnison. We could write some code to simulate a browser and reverse engineer the login process, but that's just ugly. And if the form changes, our code would break. It would be better to configure the API to authenticate with something that is more API friendly, such as HTTP Basic authentication.

OpenUnison can be extended by creating configuration custom resources. In fact, most of OpenUnison is configured using these custom resources. You deployed them in the third Helm chart we used to stand up OpenUnison. The current token service assumes you are authenticating using the default OpenUnison form login mechanism, instead of a basic authentication that would be helpful from a pipeline. In order to tell OpenUnison to support API authentication, we need to tell it to:

1. Enable authentication via HTTP Basic authentication by defining an authentication mechanism
2. Create an authentication chain that uses the basic authentication mechanism to complete the authentication process
3. Define an application that can provide the token API, authenticating using the newly created chain

We won't go through the details of how to make this work in OpenUnison, instead focusing on the end results. The `chapter5` folder contains a Helm chart that was created for you to configure this API. Run it using the same `openunison-values.yaml` file you used to deploy OpenUnison:

```
helm install orchestra-token-api chapter5/token-login -n openunison -f
chapter5/openunison-values.yaml
Release "orchestra-token-api" has been installed. Happy Helming!
NAME: orchestra-token-api
LAST DEPLOYED: Wed Jul  7 15:04:21 2021
NAMESPACE: openunison
STATUS: deployed
```

```
REVISION: 5
TEST SUITE: None
```

Once deployed, we can test using `curl`:

```
export KUBE_AZ=$(curl --insecure -u 'pipeline_svc_account:start123'
https://k8sou.192-168-2-114.nip.io/k8s-api-token/token/user | jq -r
'.token.id_token')
curl --insecure   -H "Authorization: Bearer $KUBE_AZ"
https://0.0.0.0:6443/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "172.18.0.2:6443"
    }
  ]
}
```

Now, wait a minute or two and try the `curl` command again, and you'll see you're not authenticated anymore. This example is great if you're running a single command, but most pipelines run multiple steps and a single token's lifetime isn't enough. We could write code to make use of the `refresh_token`, but most of the SDKs will do that for us. Instead of getting just the `id_token`, let's generate an entire `kubectl` configuration:

```
export KUBECONFIG=/tmp/r
kubectl get nodes
W0707 15:18:50.646390 1512400 loader.go:221] Config not found: /tmp/r
The connection to the server localhost:8080 was refused - did you
specify the right host or port?
curl --insecure -u 'pipeline_svc_account:start123' https://
k8sou.192-168-2-114.nip.io/k8s-api-token/token/user 2>/dev/null | jq -r
'.token["kubectl Command"]' | bash
Cluster "kubernetes" set.
Context "kubernetes" created.
User "pipelinex-95-xsvcx-95-xaccount@kubernetes" set.
Switched to context "kubernetes".
kubectl get nodes
NAME                      STATUS   ROLES                  AGE
```

```
VERSION
cluster01-control-plane   Ready    control-plane,master   2d15h
v1.21.1
cluster01-worker          Ready    <none>                 2d15h
v1.21.1
```

We're getting a short-lived token securely, while also interacting with the API server using our standard tools! This solution only works if your service accounts are stored and accessed via an LDAP directory. If that's not the case, you can extend OpenUnison's configuration to support any number of configuration options. To learn more, visit OpenUnison's documentation at `https://openunison.github.io/`.

This solution is specific to OpenUnison because there is no standard to convert a user's credentials into an `id_token`. That is a detail left to each identity provider. Your identity provider may have an API for generating an `id_token` easily, but it's more likely you'll need something to act as a broker since an identity provider won't know how to generate a full `kubectl` configuration.

Using certificates

The preceding process works well, but requires OpenUnison or something similar. If you wanted to take a vendor-neutral approach you could use certificates as your credential instead of trying to generate a token. Earlier in the chapter, I said that certificate authentication should be avoided for users because of Kubernetes' lack of revocation support and the fact that most certificates aren't deployed correctly. Both of these issues are generally easier to mitigate with pipelines because the deployment can be automated.

If your enterprise requires you to use a central store for service accounts, this approach may not be possible. Another potential issue with this approach is that you may want to use an enterprise CA to generate the certificates for service accounts, but Kubernetes doesn't know how to trust third-party CAs. There are active discussions about enabling the feature but it's not there yet.

Finally, you can't generate certificates for many managed clusters. Most managed Kubernetes distributions, such as EKS, do not make the private keys needed to sign requests via the built-in API available to clusters directly. In that case, you'll be unable to mint certificates that will be accepted by your cluster.

With all that said, let's walk through the process:

1. First, we'll generate a keypair and **certificate signing request** (**CSR**):

   ```
   openssl req -out sa_cert.csr -new -newkey rsa:2048 -nodes
   -keyout sa_cert.key -subj '/O=k8s/O=sa-cluster-admins/CN=sa-
   ```

```
cert/'
Generating a RSA private key
..........+++++
.................................+++++
writing new private key to 'sa_cert.key'
-----
```

Next, we'll submit the CSR to Kubernetes:

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: sa-cert
spec:
  request: $(cat sa_cert.csr | base64 | tr -d '\n')
  signerName: kubernetes.io/kube-apiserver-client
  usages:
  - digital signature
  - key encipherment
  - client auth
EOF
```

2. Once the CSR is submitted to Kubernetes, we need to approve the submission:

```
kubectl certificate approve sa-cert
certificatesigningrequest.certificates.k8s.io/sa-cert approved
```

After being approved, we download the minted certificate into a `pem` file:

```
kubectl get csr sa-cert -o jsonpath='{.status.certificate}' |
base64 --decode > sa_cert.crt
```

3. Next, we'll configure `kubectl` to use our newly approved certificate

```
cp ~/.kube/config ./sa-config
export KUBECONFIG=./sa-config
kubectl config set-credentials kind-cluster01 --client-key=./
sa_cert.key --client-certificate=./sa_cert.crt
kubectl get nodes
Error from server (Forbidden): nodes is forbidden: User "sa-
cert" cannot list resource "nodes" in API group "" at the
cluster scope
```

The API server has accepted our certificate, but has not authorized it. Our CSR had an "o" in the subject called `sa-cluster-admins`, which Kubernetes translates to "the user `sa-cert` is in the group `sa-cluster-admins`". We need to authorize that group to be a cluster admin next:

```
export KUBECONFIG=
kubectl create -f chapter5/sa-cluster-admins.yaml
export KUBECONFIG=./sa-config
kubectl get nodes
NAME                       STATUS   ROLES                   AGE
VERSION
cluster01-control-plane    Ready    control-plane,master    2d17h
v1.21.1
cluster01-worker           Ready    <none>                  2d17h
v1.21.1
```

You now have a key pair that can be used from your pipelines with your cluster! Beware while automating this process. The CSR submitted to the API server can set any groups it wants, including `system:masters`. If a certificate is minted with `system:masters` as an "o" in the subject it will not only be able to do anything on your cluster, it will bypass all RBAC authorization. It will bypass all authorization!

If you're going to go down the certificate route, think about potential alternatives, such as using certificates with your identity provider instead of going directly to the API server. This is similar to our token-based authentication, but instead of using a username and password in HTTP Basic authentication, you use a certificate. This gives you a strong credential that can be issued by your enterprise certificate authority while avoiding having to use passwords.

Having discussed how to properly authenticate to your cluster from your pipeline, let's examine some anti-patterns with pipeline authentication.

Avoiding anti-patterns

It turns out most of the anti-patterns that apply to user authentication also apply to pipeline authentication. Given the nature of code that authenticates, there are some specific things to look out for.

First, don't use a person's account for a pipeline. It will likely violate your enterprise's policies and can expose your account, and maybe your employment, to issues. Your enterprise account (and that assigned to everyone else in the enterprise) generally has several rules attached to it. Simply using it in code can breach these rules. The other anti-patterns we'll discuss add to the risk.

Next, never put your service account's credentials into Git, even when encrypted. It's popular to include credentials directly in objects stored in Git because you now have change control, but it's just so easy to accidentally push a Git repository out to a public space. Much of security is protecting users from accidents that can leak sensitive information. Even encrypted credentials in Git can be abused if the encryption keys are also stored in Git. Every cloud provider has a secret management system that will synchronize your credentials into Kubernetes Secret objects. You can do this with Vault as well. This is a much better approach as these tools are specifically designed to manage sensitive data. Git is meant to make it easy to share and collaborate, which makes for poor secret management.

Finally, don't use legacy service account tokens from outside of your cluster. I know, I've said this a dozen times in this chapter, but it's incredibly important. When using a bearer token, anything that carries that token is a potential attack vector. There have been network providers that leak tokens as an example. It's a common anti-pattern. If a vendor tells you to generate a service account token, push back: you're putting your enterprise's data at risk.

Summary

This chapter detailed how Kubernetes identifies users and what groups their members are in. We detailed how the API server interacts with identities and explored several options for authentication. Finally, we detailed the OpenID Connect protocol and how it's applied to Kubernetes.

Learning how Kubernetes authenticates users and the details of the OpenID Connect protocol is an important part of building security into a cluster. Understanding the details and how they apply to common enterprise requirements will help you decide the best way to authenticate to clusters, and also provide justification regarding why the anti-patterns we explored should be avoided.

In the next chapter, we'll apply our authentication process to authorizing access to Kubernetes resources. Knowing who somebody is isn't enough to secure your clusters. You also need to control what they have access to.

Questions

1. OpenID Connect is a standard protocol with extensive peer review and usage.
    a. True
    b. False

2. Which token does Kubernetes use to authorize your access to an API?
    a. `access_token`
    b. `id_token`
    c. `refresh_token`
    d. `certificate_token`
3. In which situation is certificate authentication a good idea?
    a. Day-to-day usage by administrators and developers
    b. Access from external CI/CD pipelines and other services
    c. Break glass in case of emergency when all other authentication solutions are unavailable
4. How should you identify users accessing your cluster?
    a. Email address
    b. Unix login ID
    c. Windows login ID
    d. An immutable ID not based on a user's name
5. Where are OpenID Connect configuration options set in Kubernetes?
    a. Depends on the distribution
    b. In a ConfigMap object
    c. In a secret
    d. Set as flags on the Kubernetes API server executable
6. When using Impersonation with your cluster, the groups your user brings are the only ones needed.
    a. True
    b. False
7. The dashboard should have its own privileged identity to work properly.
    a. True
    b. False