



A Zero Trust Reference Architecture


with Linkerd, cert-manager,
Emissary-ingress, and Polaris.





Table of Contents

Introduction	03
Chapter 1: What is Zero Trust?	05
Chapter 2: Zero Trust Reference Architecture	11
Chapter 3: cert-manager Deep Dive	17
Chapter 4: Emissary-ingress Deep Dive	33
Chapter 5: Linkerd Deep Dive	43
Chapter 6: Polaris Deep Dive	60
Bringing it All Together	67
Zero Trust for the Enterprise	68





Introduction

By Flynn, Tech Evangelist at Buoyant

Trust and security have been at the core of computing almost since its inception. We've basically always needed to have ways to talk about who is allowed to access what.

Previously, security models were more straightforward: we put walls around the things we wanted to protect, checked the requester's identity once at that perimeter, and remembered that decision for everything inside our application. This *perimeter security* model worked rather well for the pre-cloud days.

However, in today's cloud-native landscape, the complexity has significantly increased. Microservices are the order of the day, running on hardware owned by a cloud provider and, most likely, running a competitor's code at the same time as our own. Our control over the hardware and the network is sharply limited, but we still need security, and that security has to be capable of managing this complex new world.

The *zero trust* model has emerged as a solution tailored for the cloud-native world to meet these challenges.

In this book, experts in cloud-native computing representing four different projects (cert-manager, Emissary-ingress, Linkerd, and Polaris) will take a deep dive into zero trust itself and how these four projects can work together in a well-defined reference architecture. We'll explain what zero trust is and why it's important, then discuss the reference architecture, then take a deep dive into each project.

How to Use This Book

This book walks you through using four separate products to implement zero trust for a single application. Everything in this book is available in [its GitHub repository](#)-start with [README.md](#) and we encourage you to take advantage of all the code!

Each project has its own chapter, in which we'll walk you through installing and configuring that product. **The chapters are cumulative:** for example, when we install Emissary-ingress in Chapter 4, we assume you've already installed cert-manager as described in Chapter 3. To get started, you need just two things:

1. An empty Kubernetes cluster that can expose a LoadBalancer Service to the internet. [Civo](#) is a great place to go for this.
2. A way to give that LoadBalancer Service's IP address a name in the global DNS. Civo can do this for you, too.

Unfortunately, the need for a globally-addressable LoadBalancer IP (for Emissary) means you can't directly use this with a K3d cluster on your laptop – but Civo makes it easy enough to get set up if you don't already have your own cloud provider that this shouldn't be a barrier.

The purpose of this book isn't just to describe a demo but to give you the resources you need to implement zero trust safely in your own projects – hence the hands-on nature. We look forward to hearing how things go for you at info@kubecrash.io.



What is Zero Trust?

By William Morgan, CEO at Buoyant

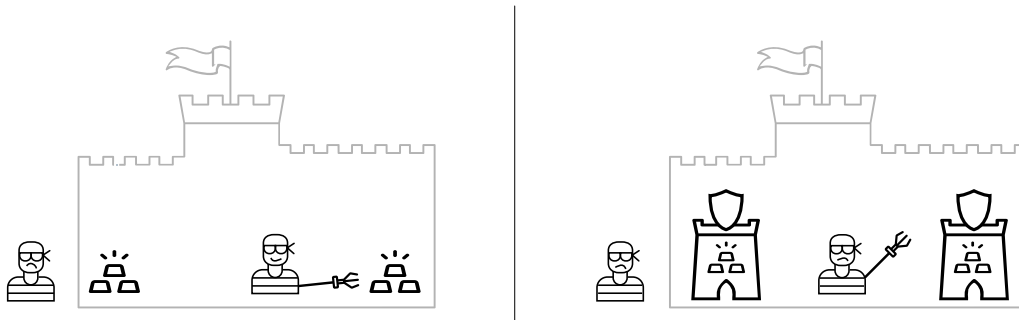
As you might expect, the zero trust model is fundamentally about trust. This model seeks to answer a crucial network security question: 'Should entity X have access to resource Y?'

The “zero” in zero trust, of course, is a bit of a conceit. For software to work, you need some trust. So zero trust isn't about removing trust entirely so much as reducing it to the bare minimum necessary and making the trust explicit rather than implicit.

Phrased that way, zero trust may not sound particularly revolutionary. Of course we should use the bare minimum trust and make things explicit. So what's all the fuss about?

Like many technological innovations, comprehending zero trust involves understanding the traditional models it challenges. In short, zero trust is the rejection of the perimeter security approach that has dominated network security in the past. In perimeter security, you implement a hard shell around your sensitive components—for example, a firewall around your data center. This model is sometimes called the “castle approach” because the firewall acts like castle walls. Under this model, entities outside the castle are potential threats requiring strict scrutiny, whereas those inside are considered trusted and safe.

Perimeter Security vs Zero trust security



The zero trust model says that this model is no longer enough. According to zero trust, even within any security perimeter established by a firewall, you must still treat users, systems, and network traffic as fundamentally untrustworthy. The DoD's Zero Trust Reference Architecture sums it up nicely:

“[N]o actor, system, network, or service operating outside or within the security perimeter is trusted. Instead, we must verify anything and everything attempting to establish access. It is a dramatic paradigm shift in philosophy of how we secure our infrastructure, networks, and data, from verify once at the perimeter to continual verification of each user, device, application, and transaction.”

Of course, zero trust doesn't mean throwing away your firewalls. Defense in depth is an important component of any security strategy. Nor does it mean we get to ignore all the other important components of security like event logging and supply chain management.

However, zero trust necessitates a significant shift in our approach to security we must move our trust checking from “once, at the perimeter,” to “everywhere, every time.”

Why is zero trust suddenly important?

The reason zero trust has become so important is that it addresses some of the security challenges in modern cloud software. To see why, let's compare the "good ol' days" of running software with the modern, cloud way.

In the good ol' days, we had:

1. **Physical machines, which we owned.** These machines were in a data center, sitting in locked cages, behind locked doors, and staffed by security personnel.
2. **A physical network, which we owned.** Our physical network cabling was also within those secure data centers, protected by cages, doors, and guards.
3. **100% control over the machines and the network.** Everything that ran on those machines was there because we put it there.
4. **Low expectations for our software!** We could get away with quarterly releases. We could have regular planned downtime. There weren't even that many people on the Internet!

Life was so simple back then! And in that world, the perimeter security approach was, well... not perfect, but in many ways sufficient to meet those constraints. Within our firewall, we could trust our machines and network, and all we had to do was make sure bad actors couldn't get in.

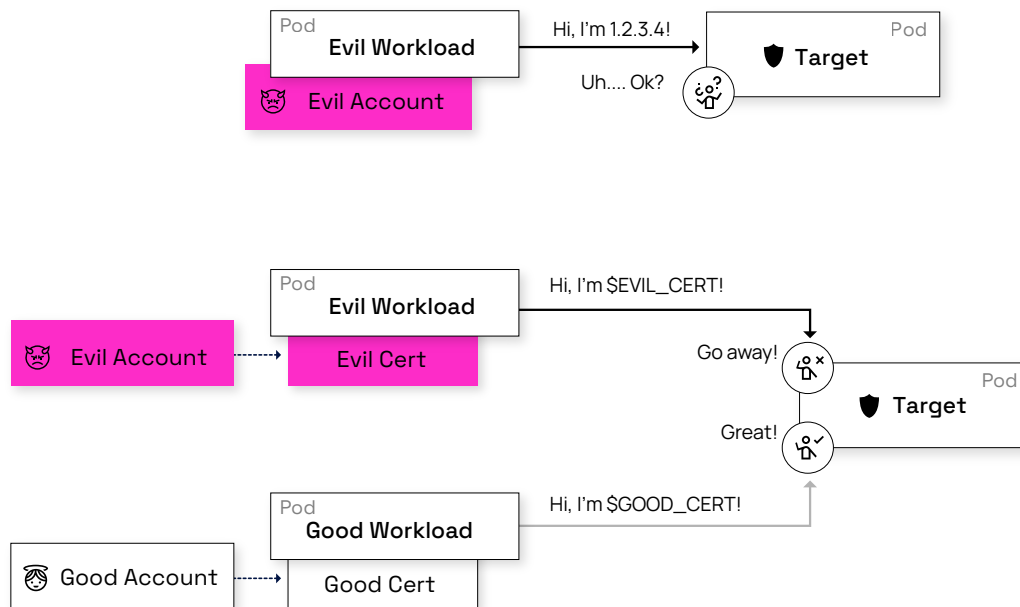
Fast forward to the modern world. We now have:

1. **No ownership of physical machines,** which are instead rented from a cloud provider and provided through a layer of virtualization.
2. **No ownership of the network,** which is also rented and provided to us virtually.
3. **No control over the actual network or machines,** which is shared with all of our cloud providers' other tenants.
4. **Extremely high demands on our software,** including daily releases, zero downtime, and scaling to massive amounts of traffic.

The trust once placed in the physical layers of our infrastructure can no longer be relied upon in the same way. Instead, we can only regain this trust through what we control—our software. That's why the zero trust model is important today. This shift to zero trust has profound implications for how we think about identity, policy, and enforcement.

What is identity?

Zero trust requires that we rework how we think about identity, especially system identity.



In the perimeter model, your network location was effectively your identity. If you were inside the firewall, you were trusted; if you were outside it, you weren't trusted. Perimeter-based systems could thus allow access to sensitive systems based on the IP address of the client.

In the zero trust world, we can no longer trust the network. At all. This means that your IP address now indicates location, nothing more. (And even that cannot really be trusted—there are many ways that IP addresses can be spoofed and forged!) For zero trust, we need another form of identity: one tied to a workload, user, or system in some intrinsic way. And this identity must also be verifiable in some way that doesn't require trusting the network.

This is a big requirement with many implications. Even systems that provide security but rely on network identifiers such as IP addresses, such as IPSec or Wireguard, are not sufficient for zero trust.

What is policy?

Armed with our new identity model, we also need a way of capturing the access each identity has. In the perimeter approach, it was common to grant full access

to a sensitive resource to a range of IP addresses. For example, we might set up IP address filtering to ensure that only IP addresses from within the firewall are allowed to access a sensitive service. In zero trust, we instead need to enforce the minimum level of access necessary. Access to a resource should be as restricted as possible, based on identity as well as any other relevant factors.

While our application code could make these authorization decisions itself, we typically instead capture it with some form of policy specified outside the application. Having an explicit policy allows us to audit and change access without modifying the application code.

In service of our zero trust goals, these policies can be very sophisticated. We may have a policy restricting access to a service to only those calling services

Account 1	
PUT /crown-jewels	✗
GET /crown-jewels	✓
GET /boring-stuff	✓
Account 2	
...	

that need access (i.e., using the workload identity on both sides). We may refine that further and allow only access to certain interfaces (HTTP routes, gRPC methods) on that service. We may refine that even further and restrict access based on the user identity responsible for the request. In all cases, the goal is the “least privilege” principle—systems and data should be accessible only when absolutely necessary.

Enforcement

Finally, zero trust requires that we perform both authentication (confirmation of identity) and authorization (validating that the policy allows the action) at the most granular level possible. Every system granting access to data or computation should enforce a security boundary, from the perimeter on down to individual components.

Similar to policy, this enforcement is ideally done uniformly across the stack. Rather than each component using its own custom enforcement code, using a uniform enforcement layer allows for auditing and decouples the concerns of application developers from those of operators and security teams.

Zero trust for Kubernetes

Faced with the requirement that we must rethink identity from first principles, reify trust in the form of policies of arbitrary expressiveness, and permeate our

infrastructure with new enforcement mechanisms at every level, it is only natural to experience a moment of panic. And did we mention we need to do this by FY 2024?

The good news is that for Kubernetes users, at least, Kubernetes can make some aspects of adopting zero trust significantly easier. Kubernetes's gift to the world is a platform with an explicit scope, a well-defined security model, and clear mechanisms for extension, which makes Kubernetes particularly fruitful for zero trust.

And thankfully, a lot of projects in the Kubernetes ecosystem have sprung up to help users realize the benefits of zero trust in their own applications! In this book, we'll show you how four of these projects - cert-manager, Emissary-ingress, Linkerd, and Polaris - work together to secure a single application in line with the zero-trust model.

That's not to say that Kubernetes is a security panacea, of course. Kubernetes security is a complex topic that requires a very clear understanding of the potential threats and what Kubernetes can and cannot provide to help you protect against them. A "pure" zero trust environment may never be fully achievable in practice, and that's OK: the value of zero trust is that it provides a model to better measure and understand security decisions.

Summing it up

Zero trust is a powerful security model that's at the forefront of modern security practices. If you can cut through the marketing noise, adopting zero trust offers some profound and important benefits. And while zero trust requires some radical changes to core ideas such as identity, Kubernetes users at least have a big leg up: the ecosystem already provides the tools they need - like cert-manager, Emissary-ingress, Linkerd, and Polaris - to shift from a purely perimeter-based security model to "continual verification of each user, device, application, and transaction."

Zero Trust Reference Architecture

Edidiong Asikpo, Senior Developer Advocate, Ambassador Labs

As we saw in the previous chapter, there is a good reason why zero trust is such a hot topic. For Kubernetes companies that care about security (and who doesn't?), zero trust isn't a nice-to-have, it's a business critical must-have. So, how do you implement such an approach? In this book, we'll provide a step-by-step guide for zero trust with open source tools.

You've probably heard people talking about "zero trust and the service mesh," "zero trust and certificate management," and even "zero trust policy enforcement." That's because zero trust spans several distinct levels, and requires a carefully-assembled software stack to manage everything – from Pod-to-Pod and cross-cluster communications, to verifying identity and authentication, to policy enforcement. Throughout this book, we'll discuss the role of each piece and how to implement it. This reference architecture will be based on the following open source projects:

- [Emissary-ingress](#) runs at the edge of the cluster, providing secure access to the Kubernetes cluster with TLS and enforcing end-user authentication, thereby bridging the external and internal worlds.
- [Linkerd](#) manages secure communications inside the cluster, supporting mTLS everywhere and enforcing low-level security policies to guarantee that each workload has the minimal access needed.
- [cert-manager](#), as the name implies, manages the many TLS certificates needed to ensure identity within and outside of the Kubernetes cluster.
- [Polaris](#) monitors which policies are defined and ensures that no one is straying from the specified zero-trust best practices intentionally or otherwise.

Each of these projects plays a very specific role within our zero trust reference architecture. We'll start with a birds-eye view of our architecture, then dive into how each project works together to ensure that every access is checked, every time, at every level of the infrastructure and the application.



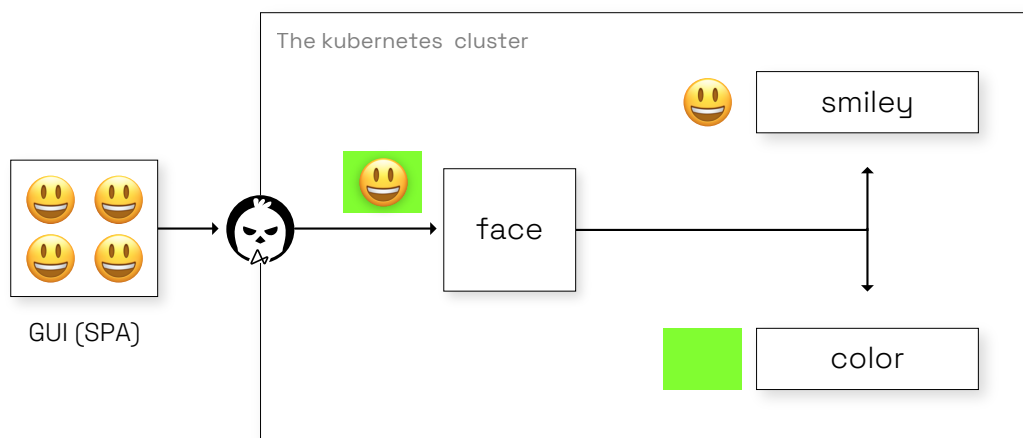
The architecture of the Faces application

We'll use the [Faces demo application](#) to demonstrate how zero trust works.

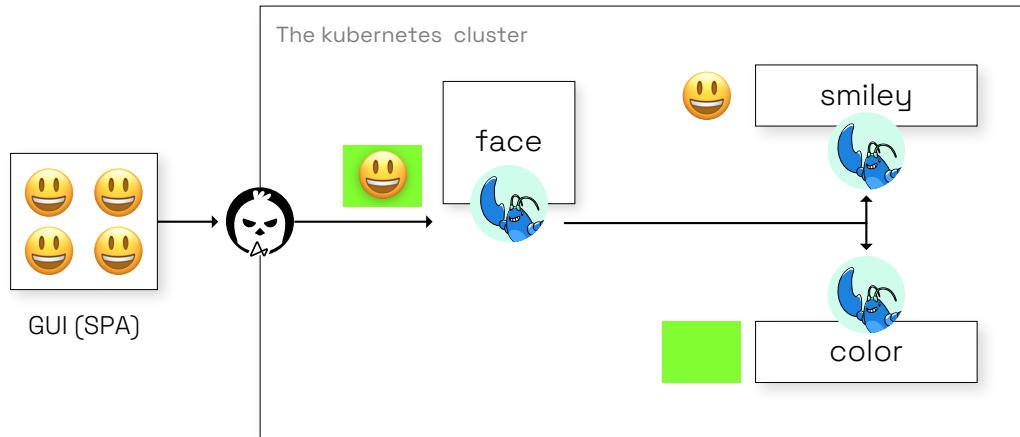
There are two parts to this application: (1) the Kubernetes cluster, where all the services that make up the application run, and (2) a single-page web app that serves as the GUI (graphical user interface) for the application.

Requests from the GUI are sent to the face service inside the Kubernetes cluster. The **face** service then communicates with the **smiley** and **color** services. The smiley service is expected to return smileys, while the color service is built to return green. The face service combines the returned values from the two services (**smiley** and **color**) and sends them back to the GUI, which displays them into multiple cells.

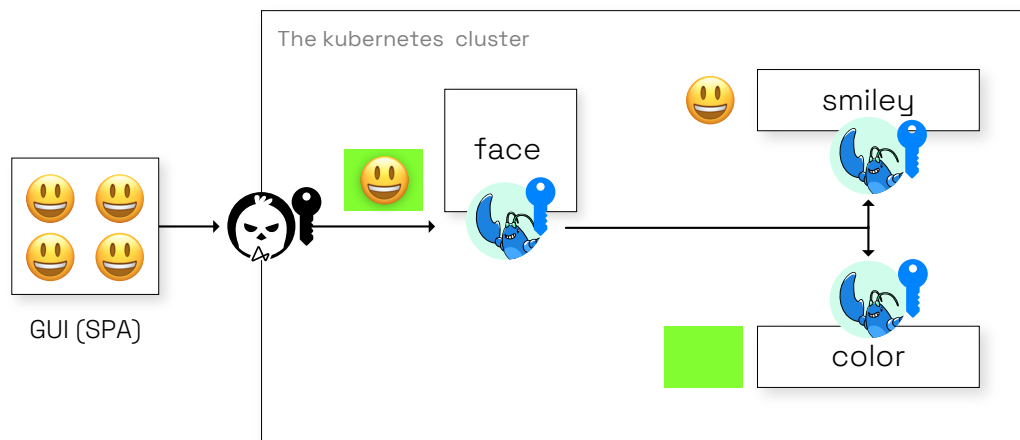
We use Emissary-ingress (represented by its blackbird logo) as the ingress controller to route user requests into the Kubernetes cluster and communicate with the **face** service.



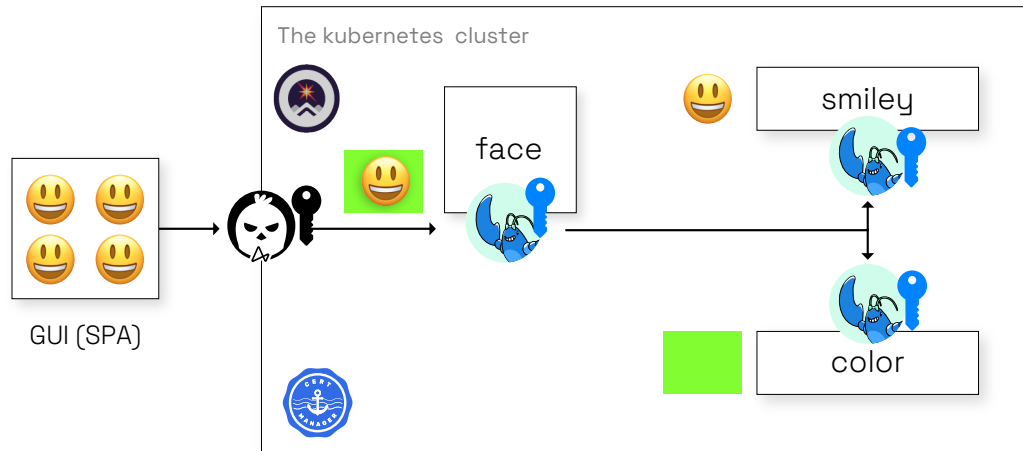
Linkerd (represented by Linky the lobster) mediates all the traffic flow inside the cluster. You see Linky at many points because there's a Linkerd proxy for every workload Pod in the cluster.



All the communications we show here use TLS, which relies on certificates. Emissary-ingress uses a certificate provided by cert-manager to identify itself to the browser (shown by the black key in the diagram below); Linkerd uses many more for mTLS between workloads (the blue keys), which are in turn secured by Linkerd's trust anchor and identity issuers. They're not shown in this diagram, but are also provided by cert-manager (represented by its logo).



Finally, Polaris (represented by the mountain logo) keeps an eye on everything going on in the Kubernetes cluster to ensure that all the set best practices are being followed, and when they are not, notify the necessary parties about it. In essence, it ensures that the only things happening in the cluster are things that we are okay with.



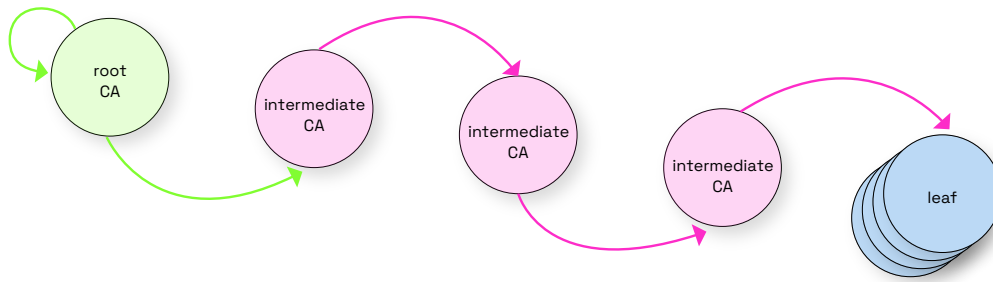
With this, all four projects have been incorporated into the application, giving us a solid foundation for zero trust across the entire cluster and application.

Trust, TLS, and Certificates

We mentioned above that TLS is fundamental to this zero trust architecture. TLS relies on X.509 certificates (normally just called “certificates” or “TLS certificates”) to convey identity: each certificate has a public key and a private key, with the private key actually representing identity, and the public key providing a way to cryptographically validate the private key.

The Trust Hierarchy

Certificates exist within a trust hierarchy or trust chain, where each certificate is signed by another certificate, with one exception: at the very top of the hierarchy is the root certificate, which always signs itself. Signing a certificate is commonly called issuing the certificate: the issuing certificate, or *issuing authority*, is effectively vouching for the issued certificate, as shown in the diagram in next page:



We talk about a workload sending “its certificate” to another workload, but really, it’s an entire trust chain being sent. The workload receiving the certificate can then walk up the trust chain, validating each signature until it finds a certificate it already trusts. If it reaches the top of the chain without finding any trusted certificate, the unknown certificate is rejected. (One important note in all this is that validating signatures only requires the public key of the issuer – the workloads don’t need, and must not have, the private key.)

An entity that uses a certificate to issue other certificates is called a *Certifying Authority*, or *CA*. These entities used to be companies, but now they’re more likely to be software operated for whatever group of certificates makes sense, often using just a single certificate as an issuer. Unlike validating certificates, issuing certificates does require access to the issuer’s private key, so a CA can’t function without that.

We’ll be using cert-manager to manage certificates. cert-manager is itself a CA that provides an automated way to issue certificates in a Kubernetes cluster.

Rotating Certificates

A vitally important note about certificate management is that certificates expire, and if they’re in use when they expire, it will cause downtime. To prevent this, certificates must be rotated, regenerating their keypairs before they expire. We’ll also be automating this with cert-manager: it’s possible to do it manually, but it’s far from ideal.

Deciding how frequently to rotate certificates is a bit of a balancing act. When a key is compromised, it will stay compromised until at least its next rotation, which means that rotating frequently helps to reduce risk. Rotating every few hours means that there’s only a small window during which the compromised certificate can be exploited. However, your rotation practices should be regularly tested to ensure their effectiveness. Longer rotation periods might lend more time to focus on work rather than the platform, but they can carry increased risk.

Installing Faces

This whole book starts with the simple step of installing Faces to your Kubernetes cluster. First, clone the [GitHub repository for this ebook](#):

```
$ gh repo clone kubecrash/spring2023 kubecrash-spring2023
$ cd kubecrash-spring2023
```

Then... install Faces!

```
$ kubectl create ns faces
$ kubectl apply -f k8s/02-faces-workloads
$ kubectl -n faces wait --for condition=available \
    --timeout=90s deploy --all
```

At this point Faces will be running, and you can test it with `kubectl port-forward`:

```
$ kubectl port-forward -n faces svc/faces-gui 8080:80 &
$ curl http://localhost:8080/
```

Once that `curl` succeeds (which might take a little time after all the Deployments are running), kill the `port-forward` and get on with the rest of the book!

Summing it up

The world is evolving. With applications mostly built in the cloud (AKA someone else's infrastructure), we are faced with new challenges that require a new security model: a zero trust approach. Zero trust enables us to embrace a hybrid workplace, protect people, devices, and data wherever they're located, and effectively adapt to the complexity of the modern environment.



CHAPTER 3

cert-manager Deep Dive

Tim Ramlot, cert-manager maintainer, Venafi

We've seen a birds-eye view of how these projects contribute to our zero trust reference architecture. In the next chapters, experts from each open source project describe the role of their project and how to implement them correctly, from providing secure access to clusters to securing communication within clusters to managing TLS certificates, and defining the necessary policies.

In the previous chapters, we learned what zero trust is, why it is so relevant today, and how it relates to the four open source projects we'll discuss in this book. This chapter will focus on [cert-manager](#), an open source X.509 certificate controller for Kubernetes.

X.509 certificates are found all over the Internet: you need them the moment you start using TLS or mTLS. In the pre-cloud days, getting a certificate was an expensive, manual, slow process. cert-manager changed all that by automating certificate issuance for Kubernetes workloads.

About cert-manager

Back in 2017, when cert-manager was first released as an open-source project, we suspected we were onto something big. Fast-forward a few years, and cert-manager has surpassed a billion downloads, has collected over 10,000 GitHub stars, and is an officially adopted project within the CNCF.

We couldn't be prouder of how cert-manager has become the go-to solution for issuing and renewing X.509 certificates from within Kubernetes clusters. When you consider the need to secure the overall speed and scope of Kubernetes and OpenShift taking place right now, the popularity of cert-manager has only just begun.

Cloud native technologies are now the de facto standard for new applications, and with Kubernetes the undisputed leader for where we run cloud-native applications, a staggering number of workloads need to be encrypted and verified as developer teams work on ever-faster release cycles.

Why cert-manager

cert-manager has received some accolades in recent years. At the start of 2021, cert-manager was considered an essential general solution for secrets management within the CNCF End User Technology Report. By the end of the year, it was included in the ThoughtWorks Technology Radar for the first time! But what are some of the more practical applications of cert-manager? Why is it being downloaded over a million times each day?

Let's take a closer look at how cert-manager is deployed to secure cloud native machine identities.

Securing ingress traffic

One of the most widespread uses of cert-manager is to secure incoming traffic to your Kubernetes clusters with TLS encryption. It just makes sense. You wouldn't give a stranger the keys to your house — so why would organizations running a highly distributed infrastructure give unfiltered access to public-facing workloads? They shouldn't.

By verifying the machine identities of incoming traffic and adopting one of the core principles of zero trust (never trust, always verify), organizations will ensure that their public-facing web applications are locked down and tamper-resistant.

mTLS protection

Developers often build internal workloads that aren't necessarily exposed to ingress traffic but could still be susceptible to an attack if a nefarious actor found their way into a related Kubernetes cluster. It's no longer enough to assume your

network perimeter is perfectly secure. We need to secure east-to-west traffic as well as north-to-south traffic within clusters.

An mTLS-type deployment for mutually authenticated communication would typically use cert-manager as the conduit to issue and renew private certificates. Whether through HashiCorp Vault, ACME, or Venafi Firefly certificates, there are several ways organizations can use cert-manager to extend the underlying principles of zero trust to include Kubernetes workloads.

Managing workloads in a service mesh

A service mesh is a networking technology that allows secure connections between the increasingly expanding number of endpoints within a cloud native architecture. Often seen as an extension of mTLS, cert-manager can be used to issue and renew certificates within service mesh zones.

A service mesh will only allow access to services within a microservice architecture with explicit authorization. This service-based identity allows an application to seamlessly scale resources to keep pace with demand. And how might a resource identify itself to a service? You guessed it, TLS certificates. In short, cert-manager acts as a control plane that can be used within service mesh environments to enforce security policies for mesh workload encryption and automated protection.

Blueprint for certificate management success

Cloud native technologies offer almost endless potential in terms of operational expenditure and speed-to-market gains, but the cloud also represents an opportunity to act on the lessons learned from the world of on-premises. As such, many organizations view the cloud as a means to avoid vendor lock-in and deploy a multi-cloud approach.

Fortunately, cert-manager is a cloud-agnostic, open-source solution that can be deployed across all your cloud environments. This allows developer teams to roll out a centralized and fully automated approach to certificate management across cloud native infrastructures.

cert-manager and zero trust

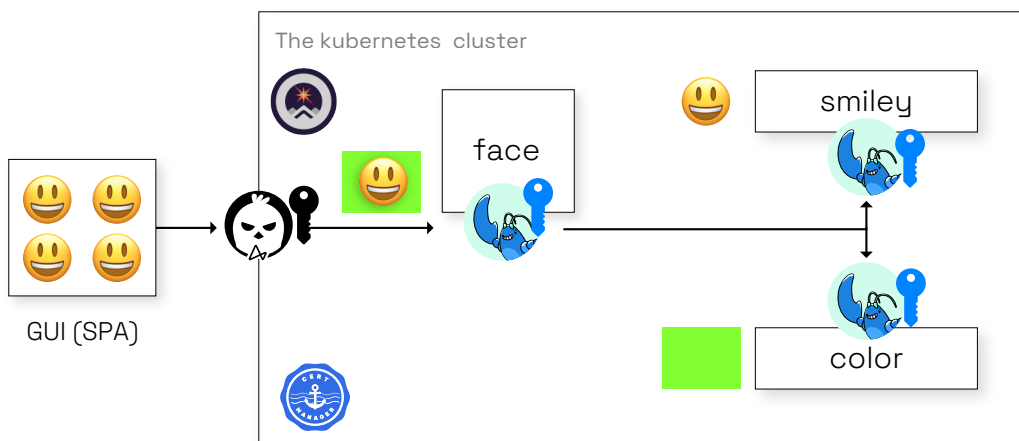
cert-manager is a certificate management tool that allows an organization to enact the founding principles of zero trust. Organizations can use cert-manager to secure the machine identities of east-to-west traffic as well as ingress.

With cert-manager, developers can ensure that every workload deployed to your Kubernetes platform is from a legitimate and verified source. This is widely accepted as a best practice container security, which developer teams can rely on to ensure they can move fast and securely. It is also important to note that the NIST guidelines recommend this approach for container security.

Certificates for cross-cluster and Pod-to-Pod communication

Within our zero trust architecture, certificates will be used to prove the identity of the Emissary API gateway and the identity of peers in an mTLS Linkerd service mesh. To do that, we need:

- A TLS termination certificate for Emissary, and
- A trust anchor certificate and an identity issuer certificate for Linkerd.



Thanks to cert-manager's issuer integrations, external CAs like Let's Encrypt, Vault, and many others can be used to sign these certificates.

[trust-manager](#), another project led by the cert-manager team, distributes and manages the CA certificates to be trusted in our Kubernetes cluster. In this chapter, we will show how to ensure all Kubernetes services trust the CA used for Linkerd

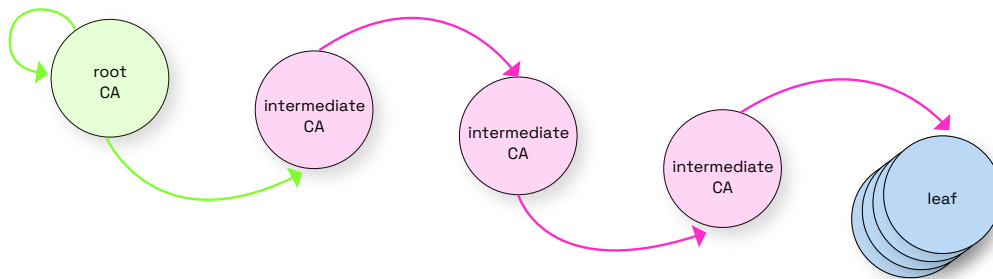
mTLS. This will allow us to verify the peer identity in an mTLS connection. Additionally, we can use trust-manager to quickly update what public CAs are trusted.

Why do we trust?

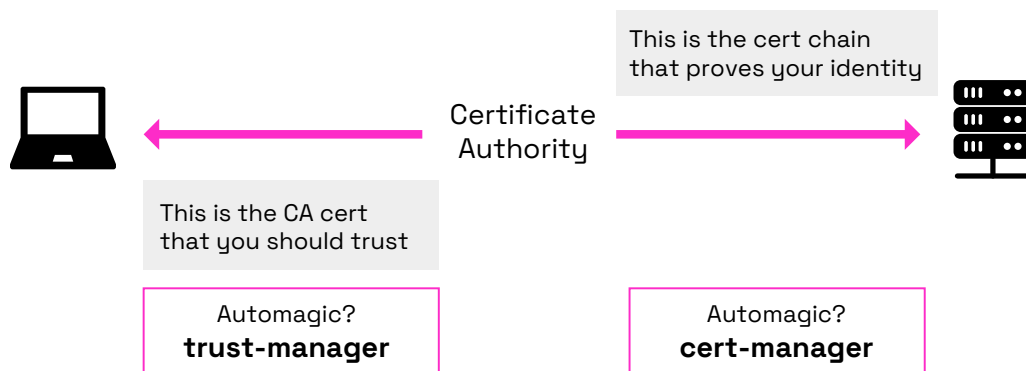
You only trust your bank with your banking login credentials, and your bank only trusts you with your bank account details after you provide your credentials, right? That is no different in zero trust. For server-to-server communication, we can prevent and limit attacks by specifying policies that restrict what services are allowed to communicate with each other.

How do we trust?

In Chapter 2, we discussed the importance of TLS for the zero trust architecture and described TLS certificates and the trust hierarchy:



cert-manager is a Certifying Authority that allows us to automate issuing certificates in a Kubernetes cluster. Its companion project, **trust-manager**, automates the provisioning of CA certificates in client trust stores to allow clients to use those CA certificates for validation

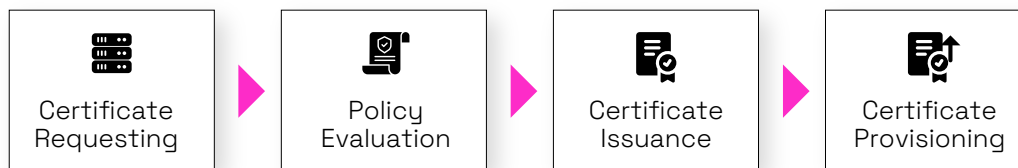


In addition to generating certificates in the first place, cert-manager also automates the vitally important function of rotating certificates before they expire to prevent downtime. It's possible to handle certificate rotation manually, but it's not a good idea: it's hard to manually rotate certificates frequently enough for good security, and it's easy to make mistakes that break things.

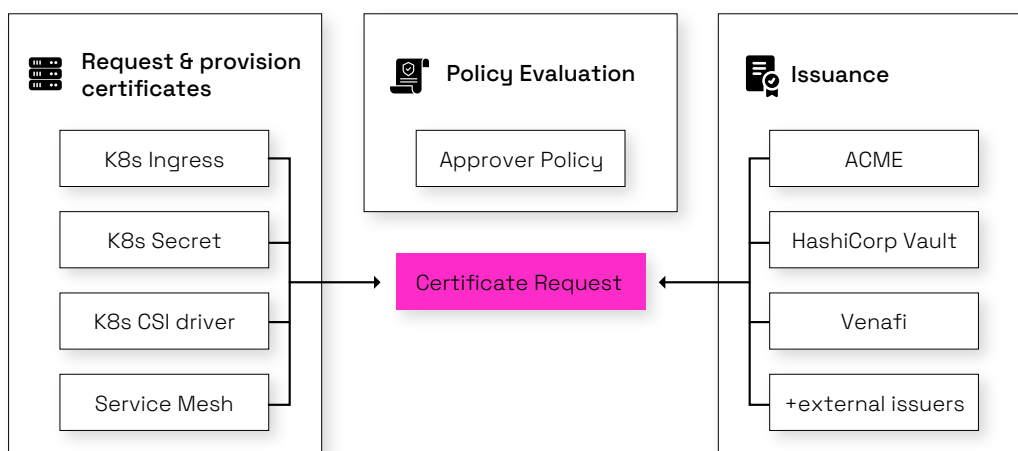
How do we automate trust using cert-manager?

cert-manager has a 4-step issuance flow:

1. **Certificate requesting:** a certificate is requested because an application needs it
2. **Policy evaluation:** a policy determines whether the request for a certificate should be sent to the CA or should be denied
3. **Certificate issuance:** the CA signs a certificate based on the request
4. **Certificate provisioning:** the signed certificate is returned to the application



These steps can be linked to the following three component types:



Now that we have established why trust is important, how we trust in a zero trust model, and how it can be automated by cert-manager, let's install cert-manager.

Installing cert-manager

First, add the Helm chart repository.

```
$ helm repo add jetstack https://charts.jetstack.io --force-update
```

Next, install the Custom Resource Definitions (CRDs).

```
# This is a single long line, don't break it!
$ kubectl apply -f https://github.com/cert-manager/cert-
manager/releases/download/v1.13.2/cert-manager.crd.yaml
```

Finally, we install the chart.

```
$ helm install \
  cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --version v1.13.2
```

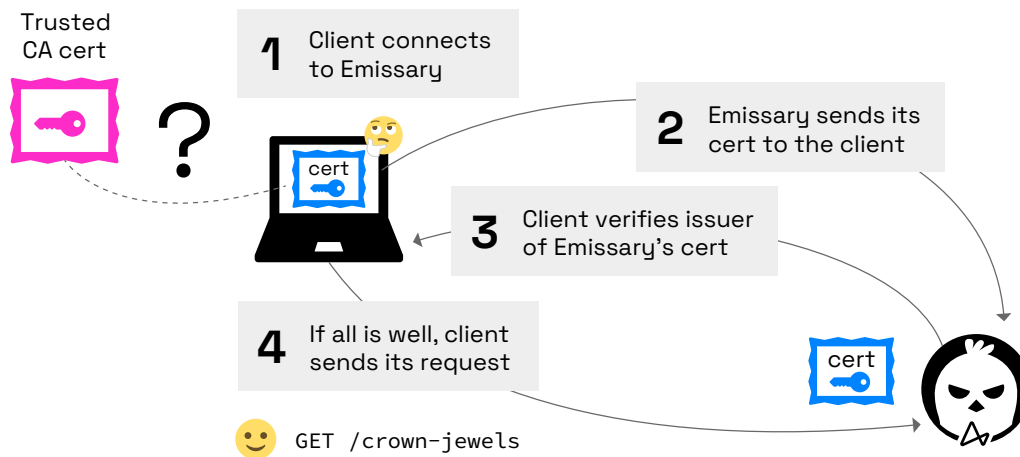
Configuring cert-manager

With cert-manager installed, we can start configuring Issuers and Certificates.

Such a Certificate resource will create a corresponding Secret resource that contains a certificate file and a private key. After mounting the Secret in a container filesystem or referencing the Secret from another application, we can use it to set up secure TLS connections between our applications in our PKI.

Emissary-ingress' Certificate:

Our zero trust architecture uses Emissary-ingress to secure traffic from users outside the cluster. Part of this security involves TLS, which means that Emissary-ingress must have a TLS certificate.



While Emissary-ingress only needs a single certificate, that certificate needs to be signed (ultimately) by something that can be trusted by a client out on the Internet. These clients are often web browsers, which are built with a hardcoded set of trusted root certificates, and the easiest way for a browser to trust Emissary-ingress' certificate is for it to come from a CA that uses one of those trusted root certificates.

Let's Encrypt is one such CA, and it's very popular because it's very easy to work with. We'll use cert-manager to get Emissary's certificate from Let's Encrypt, using ACME DNS-01 validation.

To make this work, we start by creating an Issuer in the **emissary-ingress** namespace. This Issuer needs to use Let's Encrypt for its ACME provider, but the DNS-01 validation also requires cert-manager to be able to edit the DNS. We'll demonstrate using Cloudflare as the DNS provider. To use Cloudflare, we need to create a Secret holding the Cloudflare API key, then create an Issuer referencing that Secret. Here's the Secret definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: cloudflare-api-key
  namespace: emissary-ingress
data:
  apiKey: <base64 encoded key>
```


After creating the Secret, we can create the Issuer that uses it:

```
---
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: letsencrypt-production
  namespace: emissary-ingress
spec:
  acme:
    email: user@example.com
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: production-issuer-account-key
    solvers:
      - dns01:
          cloudflare:
            apiTokenSecretRef:
              name: cloudflare-api-key
              key: apiKey
```

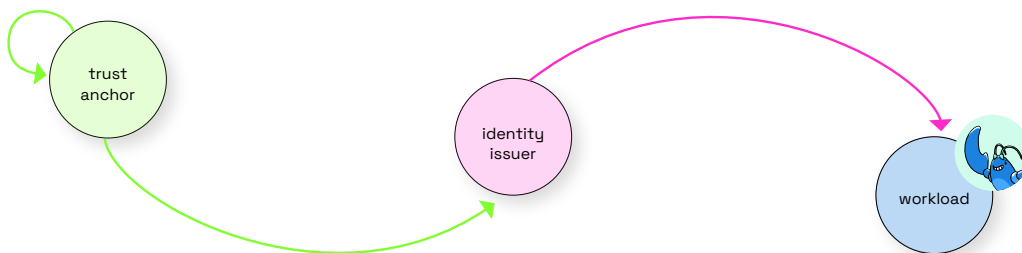
We can now use that Issuer to create a Certificate, which will store the certificate data in a Secret using our requested name. Here, we're requesting a certificate for **faces.company.com** – when setting up Emissary-ingress, it will be very important to make sure its configuration matches this hostname!

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: faces
  namespace: emissary-ingress
spec:
  secretName: faces-company-com-cert
  duration: '2160h'
  renewBefore: '360h'
  dnsNames:
    - faces.company.com
  issuerRef:
    name: letsencrypt-production
    kind: Issuer
```

Note that the Secret, the Issuer, and the Certificate above all live in the **emissary-ingress** namespace: this is necessary to have the final TLS certificate created in the **emissary-ingress** namespace, so that Emissary can find it.

Linkerd's Certificates:

Linkerd requires an *identity issuer certificate* that lives in a Secret in the cluster. Since Linkerd uses this certificate to sign and rotate the *workload certificates* used for mTLS, it needs access to both the public and private key of the identity issuer certificate. Finally, the identity issuer must be signed by the *trust anchor certificate*, so that the workloads can trust each other using the mTLS validation chain.



The best practice here is to create and manage this trust anchor in an external certificate management system (e.g., Vault, Venafi) and use one of cert-manager's issuer integrations to sign the identity issuer certificate. This way, the trust anchor will not be managed by the Kubernetes cluster, and its private key will not be stored in the cluster. In keeping with this practice, we will assume that you have already created a ClusterIssuer named **enterprise-clusterissuer** to integrate with some external certificate management system.

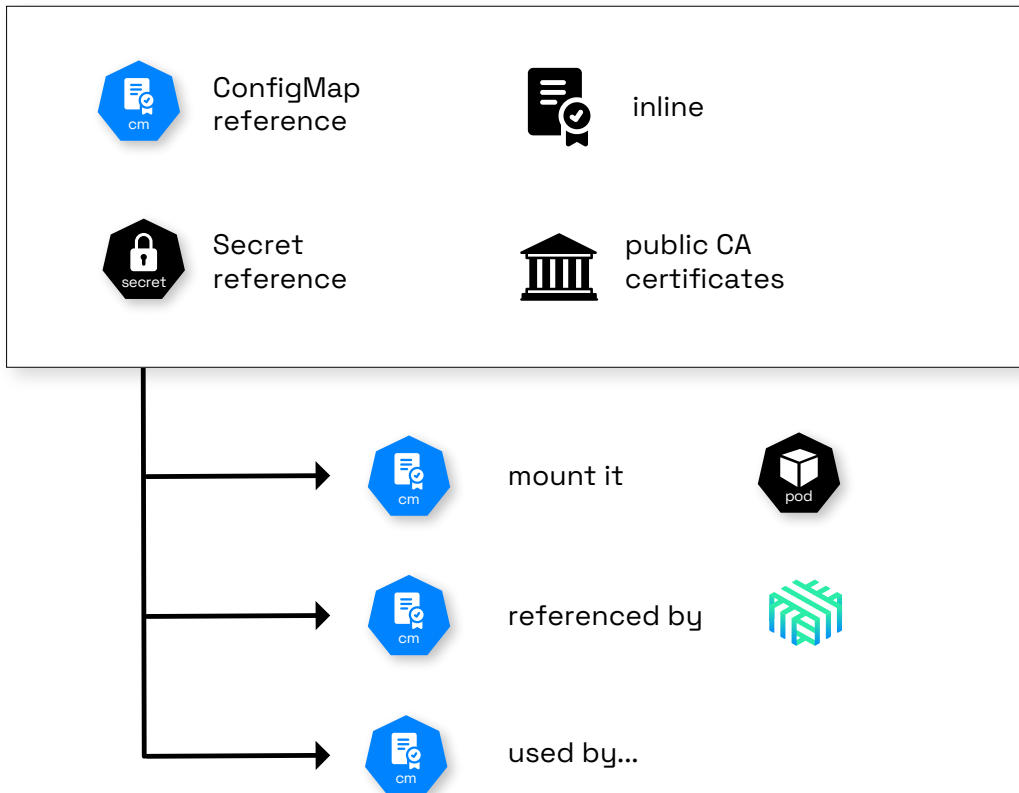
If you don't already have an external certificate management system set up, you can create a "self-signed" ClusterIssuer named **enterprise-clusterissuer** following the directions at <https://cert-manager.io/docs/configuration/selfsigned/>. **This is not production-ready**, but it will let you follow the rest of the examples in this book!.

We'll use a cert-manager Certificate resource to create Linkerd's identity issuer certificate, using the **enterprise-clusterissuer** described above. The new certificate will be written into a Secret named **linkerd-identity-issuer**, which is the name that Linkerd requires for the identity issuer certificate. Note also that the Certificate is in the **linkerd** namespace: again, Linkerd requires its identity issuer Secret to be in this namespace.

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: linkerd-identity-issuer
  namespace: linkerd
spec:
  secretName: linkerd-identity-issuer
  duration: 48h
  renewBefore: 25h
  issuerRef:
    name: enterprise-clusterissuer
    kind: ClusterIssuer
  commonName: identity.linkerd.cluster.local
  dnsNames:
  - identity.linkerd.cluster.local
  isCA: true
  privateKey:
    algorithm: ECDSA
  usages:
  - cert sign
  - crl sign
  - server auth
  - client auth
```

Automating trust using trust-manager

Next, we will automate trust with trust-manager. trust-manager reconciles bundle resources, which define a set of *sources* and a set of *targets*. During reconciliation, trust-manager combines the certificates in the sources and writes the resulting bundle to the targets in the target namespaces.



Installing trust-manager

First, let's install trust-manager by adding the Helm chart repository.

```
$ helm repo add jetstack https://charts.jetstack.io \
  --force-update
```

Next, we install the chart.

```
$ helm install \
  trust-manager jetstack/trust-manager \
  --namespace cert-manager \
  --create-namespace \
  --version v0.7.0
```

Configuring trust-manager

Making sure all servers have a certificate that they can use to host their TLS endpoints is only half of the story. We also need to ensure all our clients trust these certificates. Therefore, we must correctly populate the trust stores of our clients. For clients that live in Kubernetes, this can be done using trust-manager.

Linkerd's trust anchor certificate

Linkerd's workloads use the trust anchor certificate to validate mTLS connections. This certificate is stored in a ConfigMap resource named **linkerd-identity-trust-roots** in the Linkerd namespace. We can use trust-manager to provision this ConfigMap. We manually copy the public trust anchor certificate information from our external CA (e.g., Vault, Venafi) to the bundle. When the trust anchor must be rotated, we can add a new entry in the trust-manager Bundle and update and wait for all workloads to trust the new trust anchor before issuing a new identity CA certificate. After all identities are re-issued under the new trust anchor, we can remove the old trust anchor.

```
apiVersion: trust.cert-manager.io/v1alpha1
kind: Bundle
metadata:
  name: linkerd-identity-trust-roots
spec:
  sources:
    - inLine: |
        -----BEGIN CERTIFICATE-----
...
        -----END CERTIFICATE-----
  target:
    configMap:
      key: "ca-bundle.crt"
    namespaceSelector:
      matchLabels:
        kubernetes.io/metadata.name: "linkerd"
```

Distributing the public CAs to our applications:

We'll use trust-manager's public CA certificates option to keep our application's trust stores up-to-date. Decoupling the trust-store contents from the image version is necessary to support rolling back images to older versions.

```
apiVersion: trust.cert-manager.io/v1alpha1
kind: Bundle
metadata:
  name: example-bundle
spec:
  sources:
    - useDefaultCAs: true
  target:
    configMap:
      key: "bundle.pem"
```

Summing it up

Zero trust is about adding a mechanism to protect against malicious actors within your network, behind your firewall, and on your server. An important part of the solution is to move the “trust barrier” as close to the application as possible. This can be done using TLS. cert-manager is the best solution to set up TLS securely and safely in Kubernetes. It automatically signs and provisions certificates, ensuring new certificates are issued to protect you against outages. Managing the trust stores for your Kubernetes workloads can be automated, too, using trust-manager. With these fundamentals set up, we are ready to deploy the remaining parts of our demo faces application.



Need access to expertise and dedicated support to help you operationalise and scale cert-manager for your Kubernetes environments?

Venafi is the company that actively maintains the open source cert-manager project on behalf of the CNCF. If you need access to expertise and dedicated support to help you operationalize and scale cert-manager for your Kubernetes environments, you can [reach out to us here](#).

Emissary-ingress Deep Dive

By Dave Sudia, Director of Developer Relations at Ambassador Labs

Now that we have the TLS certificates to secure communications in our application, we'll work on traffic coming into the cluster. This might look like the old-school perimeter security model, but it's really just the start of our zero trust implementation: zero trust demands defense in depth, and "in depth" definitely includes the perimeter. You need a secured transport layer, from user to ingress to services, and that's where Emissary-ingress comes in.

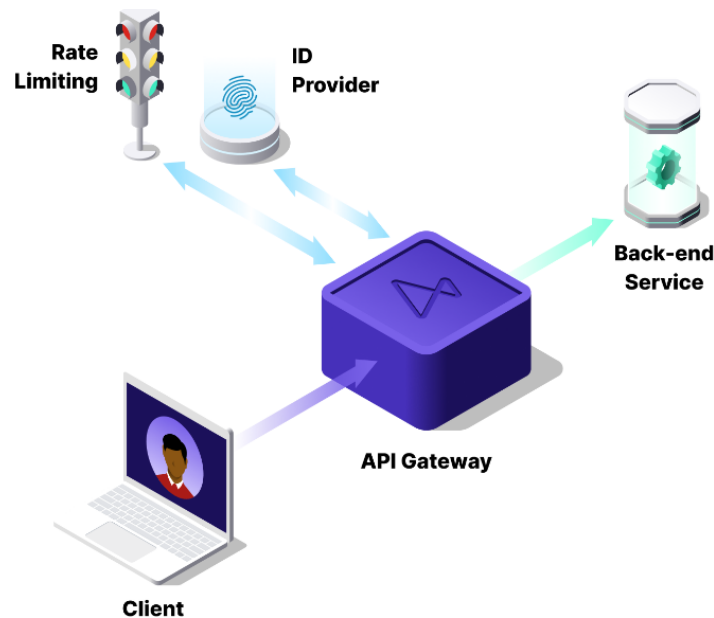
About Emissary-ingress

Emissary-ingress was first released in early 2018 with the goals of:

- Providing a better interface for an API gateway than the original IngressController and Ingress resources did, and
- Using Envoy Proxy to provide a high-performing and richly featured gateway controlled by that interface.

Emissary-ingress, which now influences projects like the Kubernetes Gateway API, established the pattern of having role-based resources. DevOps or security teams can control the deployment of Emissary-ingress, and resources like TLS certificates and hosts in a namespace that only they or their tooling can change. Developer teams can create routing rules, called Mappings, in their own namespaces, separate from those that allow ingress to traffic.

In contrast, Ingress rulesets had to contain host and TLS information alongside the routing information. They had to be colocated with the applications, i.e., where developers might have permission to make changes, making it a less secure model. Emissary-ingress also has extension points for security features like rate limiting and authentication. For an organization with the right resources, adding more extension points is possible because it is open source.



Why Emissary-ingress

Emissary-ingress has several benefits from a security perspective.

One ingress point to secure

Because it uses Envoy Proxy under the hood, Emissary-ingress scales well vertically and horizontally. With a single externally facing service (usually a load balancer type), you can ingest all traffic to your system, making it easier to secure. You don't have to track hundreds or thousands of external IPs. All security configuration happens in one place, and the permissions to manage that configuration can be locked to an approved set of users via Kubernetes RBAC.

Integration with cert-manager

Emissary-ingress can use certificates created with cert-manager to do TLS handshakes. Emissary-ingress can also route traffic to cert-manager pods that handle HTTP ACME validation to get new certificates.

Extensions

Emissary-ingress is extensible and comes with authentication and rate-limiting extension APIs built-in. By writing an authentication service or rate-limiting service, or a translation layer between Emissary-ingress's API and an existing service, these features can be added to Emissary-ingress.

Emissary-ingress and zero trust

At the perimeter, we need to enforce several aspects of zero trust:

- Is this request encrypted?
- Should we accept this request from this client?
- Should we accept this request at all?

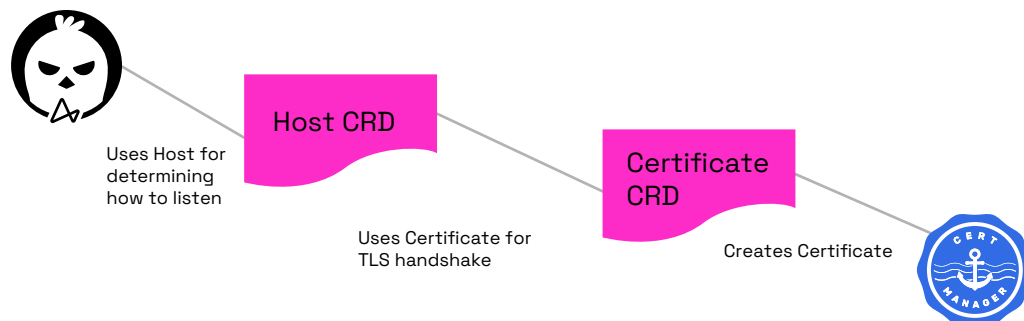
Emissary-Ingress enables organizations to answer all three questions. If a request comes in unencrypted, it can be redirected to an encrypted endpoint. If a request comes in on a hostname, port, or path we don't accept, it can be automatically rejected. And, for all or certain request paths, we can enforce authentication and rate limiting to determine if we should allow the request into the cluster based on who made it.

Encryption and authentication

We'll set up Emissary-ingress to serve secure traffic to the example Faces app used throughout this book using TLS between the user and the gateway, and to authenticate all requests. We're not worried about encrypting traffic between Emissary-ingress and our application, as Linkerd manages that.

Installing Emissary-ingress

The easiest way to install Emissary-ingress is with Helm, which we'll cover in this chapter. Full instructions (including other methods) can be found in the [Emissary-ingress docs](#).



First, add the Helm chart repository.

```
$ helm repo add datawire https://app.getambassador.io
$ helm repo update
```

Now, we need to install the CRDs (Custom Resource Definitions). This also installs a controller called **emissary-apiext** that converts between different Emissary-ingress API versions to assist in upgrade processes.

```
# This is a single long line, don't break it!
$ kubectl apply -f https://app.getambassador.io/yaml/
emissary/3.7.1/emissary-crd.yaml
$ kubectl wait --timeout=90s --for=condition=available \
  deployment emissary-apiext -n emissary-system
```

Finally, we install the chart. We set the value **createDefaultListeners** to true as this installs two Listener CRDs that tell Emissary-ingress to listen on ports 80 and 443 on the service created by the chart (by default, a load balancer).

```
$ helm install -n emissary \
  --create-namespace \
  --set createDefaultListeners=true \
  emissary-ingress datawire/emissary-ingress && \
  kubectl rollout status -n emissary \
  deployment/emissary-ingress -w
```


Defining our hostname and securing it

Next, we need to create a Host resource. This resource defines one way that Emissary-ingress is visible to the outside world (multiple Host resources can exist together). It collects:

- A hostname by which Emissary-ingress will be reachable
- How Emissary-ingress should handle TLS certificates for that hostname
- How Emissary-ingress should handle secure and insecure requests
- Optionally, which Mappings should be associated with the Host using labels

A quick note: Hosts are not required to be in the same namespace as Emissary. If a platform team manages Emissary-ingress's deployment but a security team manages hostnames, that's fine! They can be in separate namespaces with separate permissions. The Host does need to be in the same namespace as the Certificate, and - critically - the hostname needs to be the same as the one in the certificate we've gotten from cert-manager!

Our Host is pretty simple. It tells Emissary-ingress to listen to requests on the hostname **faces.company.com**, to use the certificate data we created in the previous step for TLS, and to redirect any request that uses HTTP to HTTPS.

```
---
apiVersion: getambassador.io/v3alpha1
kind: Host
metadata:
  name: faces-company-com
  namespace: emissary-ingress
spec:
  hostname: faces.company.com
  tlsSecret:
    name: faces-company-com-cert
  requestPolicy:
    insecure:
      action: Redirect
```

Routing traffic to our app

Mappings tell Emissary-ingress how to route traffic. These resources are generally owned by development teams and are deployed alongside the app they provide routing information for. Mappings can route traffic by path, parameter, or hostname, and control many network functions like timeouts, retries, advanced load balancing, WebSocket upgrades, and more. To show their structure, we'll deploy two simple mappings for the example Faces app.

```
---
apiVersion: getambassador.io/v3alpha1
kind: Mapping
metadata:
  name: faces-gui
  namespace: faces
spec:
  host: faces.company.com
  prefix: /
  service: http://faces-gui
---
apiVersion: getambassador.io/v3alpha1
kind: Mapping
metadata:
  name: face
  namespace: faces
spec:
  host: faces.company.com
  prefix: /face
  service: http://face
```

Authenticating incoming requests

At this point, traffic can reach our Faces app, and it must be encrypted. But we also want to take advantage of the extension system in Emissary-ingress to authenticate requests. Authentication can be complex, and the required methods of authentication and specific services that require those methods, are unique to a given organization. For the purposes of this guide, we'll use an application that requires Basic Auth to authenticate every request coming into the system.

We need to deploy that application along with an internal Service that Emissary-ingress can reach and then tell Emissary-ingress to use it via an AuthService resource. Here's the Service definition for Emissary's external authentication application:

```
---
apiVersion: v1
kind: Service
metadata:
  name: example-auth
  namespace: ambassador
spec:
  type: ClusterIP
  selector:
    app: example-auth
  ports:
    - port: 3000
      name: http-example-auth
      targetPort: http-api
```

Here's the Deployment for Emissary's external authentication application:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-auth
  namespace: ambassador
spec:
  replicas: 1
  strategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app: example-auth
  template:
    metadata:
      labels:
        app: example-auth
    spec:
      containers:
        - name: example-auth
          image: thedevelopnik/ambassador-auth-service:1.1.1
          imagePullPolicy: Always
          ports:
            - name: http-api
              containerPort: 3000
          resources:
            limits:
              cpu: "0.1"
              memory: 100Mi
```

Finally, we need to tell Emissary to use our **example-auth** workload to authenticate requests, by defining an AuthService resource:

```
---
apiVersion: getambassador.io/v3alpha1
kind: AuthService
metadata:
  name: authentication
  namespace: ambassador
spec:
  auth_service: "example-auth:3000"
  path_prefix: "/extauth/"
  allowed_request_headers:
    - "x-faces-session"
  allowed_authorization_headers:
    - "x-faces-session"
```

With those resources created, all requests to **faces.company.com** now require basic authentication!

You should be able to test all this simply by pointing a Web browser at <https://faces.company.com/faces/>. Authenticate once, and then you should see the Faces application in all its glory.

Summing it up

Part of defense in depth is defending the perimeter. To do so, we want to know that all traffic coming into our system is encrypted, only on validated hostnames, and (most of the time) that we know who is making the request. Inside our system we want to make sure that only approved people can make changes to these configurations. Emissary-ingress fulfills all these requirements. Next, let's explore how Linkerd secures pod-to-pod communication.



Looking to simplify your Kubernetes development experience? The team at Ambassador Labs, creators of Emissary-ingress, can help. [Get started for free](#) or [schedule a demo](#) today.

Linkerd Deep Dive

By Flynn, Tech Evangelist at Buoyant

In earlier chapters, we've seen how our zero-trust architecture uses cert-manager to establish certificates for identity and Emissary-ingress to provide a safe way to bring data from outside the cluster to workloads inside the cluster (also known as north/south traffic). The next critical step is knowing that communications from Emissary to the workloads, and from workloads to other workloads (east/west traffic), are properly secured and authenticated. That's where Linkerd can help.

About Linkerd

Linkerd is an ultralight, security-first service mesh designed to provide enterprise-grade security, reliability, and observability to Kubernetes without the complexity. Linkerd is fully open source, was one of the first projects to join the CNCF, and was the first service mesh to achieve graduated status in recognition of its maturity and strong community.

A service mesh provides critical security, reliability, and observability features without requiring application changes. Linkerd provides this functionality by transparently inserting ultralight, Rust-based “micro-proxies” into each pod. This gives Linkerd the ability to provide powerful capabilities such as mutual TLS, workload identity, fine-grained security policies, cross-cluster communication, and more, in a way that is fully transparent to the application.

Why Linkerd

The service mesh landscape is messy and complex. In this field, Linkerd stands out for its focus on simplicity and security.

Linkerd's security posture

Every aspect of Linkerd's design is built for security, from the use of Rust in the critical data plane layer to the use of Kubernetes security primitives such as ServiceAccounts and the zero-config philosophy of providing critical security features such as mutual TLS as "on by default" without requiring configuration.

Linkerd's simplicity

Linkerd installs in minutes and requires zero configuration to get started, even for advanced features such as mutual TLS. Linkerd automates as much as possible, minimizes configuration and human involvement, and focuses on reducing long-term ownership costs.

Linkerd's blazing speed and minimal resource consumption

Linkerd is the only service mesh to use a specialized Rust micro-proxy designed specifically for the service mesh use case. The choice of Rust allows Linkerd to avoid a pernicious class of security vulnerabilities and CVEs that are common to C and C++ projects such as Envoy, while retaining native code performance and a minimalist runtime footprint.

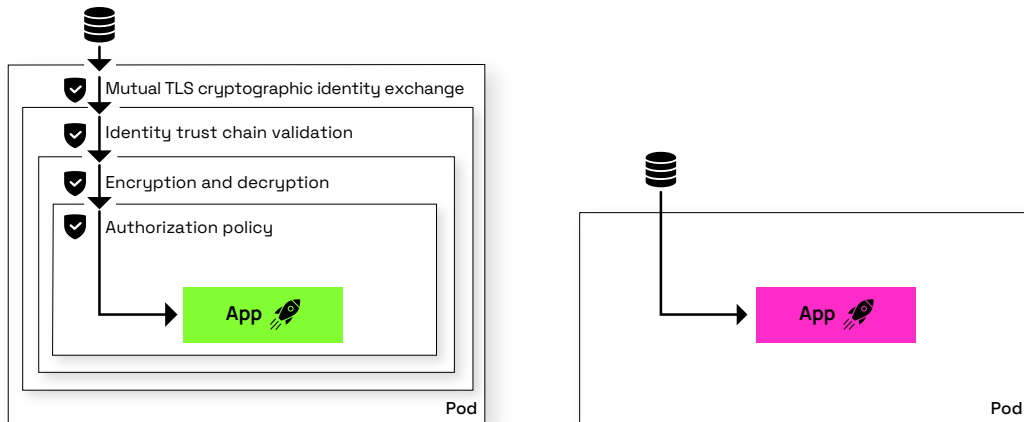
Linkerd and Zero Trust

Zero trust principles require checking every access, every time. To do this, we need knowledge about identity, encryption, integrity, and policy: both workloads involved in any communication need to have clear identification, and the request itself must remain uncompromised and confidential. Moreover, we need a clear policy to define what's acceptable and what's not.

Linkerd is the only component of the system as a whole that has visibility into all communications within the cluster. As such, it is ideally positioned to make sure that everything we need to check actually is checked every time, making Linkerd a critical piece of Kubernetes zero trust.

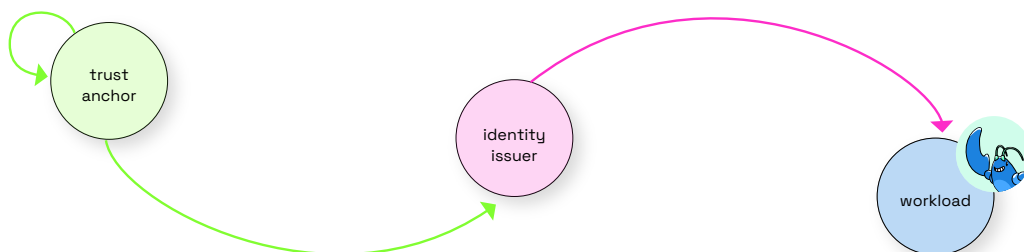
Linkerd Certificates

To ensure encryption, integrity, and identity, Linkerd adopts industry-standard mutual TLS (mTLS). This approach is automatic, requiring no configuration.



Linkerd's approach to identity hinges on certificates rather than anything tied to the network. These certificates are derived from Kubernetes ServiceAccount tokens, but ultimately, they're standard X.509 TLS certificates, which have been around for quite some time but are still often tricky to manage.

As discussed in Chapter 3 on cert-manager, certificates exist in a hierarchy of trust. From Linkerd's perspective, the root of the Linkerd trust hierarchy is the trust anchor, the ultimate source of trust. In turn, the trust anchor signs the identity issuer, which is used to sign workload certificates, which identify individual workloads.



Since Linkerd uses the identity issuer to issue workload certificates, Linkerd must have access to the identity issuer's private key. However, the trust anchor is used only for validating certificates, so Linkerd does not need the trust anchor's private key. In fact, we recommend not storing the trust anchor's private key in the cluster at all.

In Chapter 3 on cert-manager, we installed cert-manager and set it up to manage Linkerd's certificates for us, so we can trust that they're working here.

Zero-Trust Faces

As discussed earlier, the Faces app starts with a GUI which calls the **face** workload. In turn, the **face** workload calls the **smiley** and **color** workloads. (The GUI itself is served by a workload called **faces-gui**.)

We installed Faces such that all communications with the world outside the cluster are brokered by Emissary-ingress, which requires the client to be using TLS. This is a good start; the next step is to protect communications within the cluster.

Installing Linkerd

We'll install Linkerd with Helm, so we need to start by adding the Helm chart repository.

```
$ helm repo add linkerd https://helm.linkerd.io/stable
$ helm repo update
```

Next, we can use Helm to install the Linkerd CRDs (this also creates the linkerd namespace):

```
$ helm install linkerd-crds linkerd/linkerd-crds \
  -n linkerd --create-namespace
```

Once that's done, we can install the Linkerd control plane.

```
$ helm install \
  linkerd-control-plane linkerd/linkerd-control-plane \
  -n linkerd \
  --set identity.externalCA=true \
  --helm install linkerd-control-plane \
  linkerd/linkerdcontrol-plane \
```

At this point, Linkerd should be running – however, nothing will be in the mesh until the next step.

Meshing Emissary and Faces

Once Linkerd is running, it will automatically enforce mTLS between all meshed workloads, so the first step is simply to ensure that all the workloads are meshed. This includes Emissary-ingress: to Linkerd, the ingress is just another workload. (This also implies that it's important to configure Emissary to speak cleartext to its upstream workloads: we want Linkerd mTLS to do the heavy lifting of securing communications between Emissary and the upstreams.)

Note that part of Linkerd's mTLS is that communications are identified and that identity maps back to Kubernetes ServiceAccount tokens. This makes it important for every workload to have its own distinct ServiceAccount.

Meshing Linkerd and Emissary-ingress is fairly simple:

```
$ kubectl get deployment -n emissary -o yaml \
  | linkerd inject - \
  | kubectl apply -f -
$ kubectl get deployment -n faces -o yaml \
  | linkerd inject - \
  | kubectl apply -f -
$ kubectl rollout status -n emissary deploy
$ kubectl rollout status -n faces deploy
```

Default Deny

As for policy implementation, Linkerd uses its custom resource definitions (CRDs), including Server, AuthorizationPolicy, and HTTPRoute. Policies can be defined per workload or per route.

The next step is to switch Linkerd's default security posture to "deny", so that any requests not explicitly authorized are refused. This is a very important step for proper zero trust: inevitably, we'll make mistakes in configuring authorization policy, and the default-deny posture helps to prevent those mistakes from being critical.

The simplest way to switch Linkerd to default-deny is to apply the **config.linkerd.io/default-inbound-policy=deny** annotation on every namespace that should have this policy. For the Faces demo, we need this annotation on the **faces** namespace at minimum. Once this is done, you must restart the meshed workloads in the namespace since the proxies only check the default policy at startup.

Annotating the namespace and restarting the workloads is simple:

```
$ kubectl annotate ns faces \
    config.linkerd.io/default-inbound-policy=deny
$ kubectl rollout restart -n faces deploy
$ kubectl rollout status -n faces deploy
```

(The **emissary** namespace needs to be switched as well, but we're going to start with **faces**.)

Adding Permissions

After switching the default policy, the application will be entirely broken — it will show all grimacing faces on gray backgrounds, meaning that the GUI can't talk to anything at all. This is expected! To allow things to work again, we need to explicitly allow all the requests that the Faces demo needs.

Note that we only want to add necessary permissions. This is in accordance with the principle of least privilege, which is a requirement for zero trust: a given actor should not have permission to do more than they need to.

For the Faces application, it's easy to state exactly what's required:

1. Emissary-ingress needs to be able to talk to its external auth service.
2. The browser needs to be able to talk to the **faces-gui** workload (it's the one that serves the HTML and JavaScript for the GUI itself).
3. The Faces GUI needs to be able to talk to the **face** workload.
4. The **face** workload needs to be able to talk to the **smiley** and **color** workloads.

That's it. No other communications are required for the Faces application. Note that #1 is entirely within the emissary-ingress namespace, both #2 and #3 cross from the emissary-ingress namespace into the faces namespace, and #4 is entirely within the faces namespace.

For more complex applications, this list will be longer. For extremely complex applications, the list might not even be fully known when you begin. Linkerd's Viz extension can help here by monitoring traffic that actually happens and producing a set of rules you can use as a starting point. This is covered in the Linkerd [ServiceProfile documentation](#).

Adding Permission for Emissary to Reach Faces

To allow the browser to reach the **faces-gui** and **face** workload, we must permit Emissary-ingress to reach those workloads.

One critical point here is the distinction between workload auth and user auth. Part of Emissary-ingress's job is to understand who the user is: that is, the identity of the human being sitting at a browser trying to interact with Faces. Emissary-ingress is good at that; if Faces required end-user auth, Emissary would be quite capable of figuring out who the end user is and providing that information to the rest of the application (probably via a header).

Linkerd's job is to make sure that the application workloads can trust that the workload claiming to be Emissary really is Emissary. This relies on workload auth, which is a layer underneath user auth — after all, if you're not sure it's really Emissary talking to you, it's a bad idea to trust what it says about who the user is.

Given the requirements above, Emissary should only be allowed to talk to the **face** and **faces-gui** workloads.

Given the requirements above, Emissary should only be allowed to talk to the **face** and **faces-gui** workloads. To configure this, we'll add **Server**, **AuthorizationPolicy**, and **MeshTLSAuthentication** resources:

- **Server** resources describe specific ports of workloads, allowing us to specify exactly what traffic we want to authorize;
- **AuthorizationPolicy** resources describe exactly what kind of authentication is required to access a particular Server; and
- **MeshTLSAuthentication** describes specific mesh identities that make up an authenticated group.

Together, the three resources let us require specific identities for specific kinds of traffic.

We'll start by defining a Linkerd Server that selects both the **face** and **faces-gui** workloads. We'll call this **faces-front-end**:

```
---
apiVersion: policy.linkerd.io/v1beta1
kind: Server
metadata:
  name: faces-front-end
  namespace: faces
  labels:
    app: faces-front-end
    app.kubernetes.io/part-of: faces
    project: faces
spec:
  podSelector:
    matchExpressions:
      - key: service
        operator: In
        values:
          - face
          - face-gui
  port: 80
  proxyProtocol: HTTP/1
```

Once we apply this resource, we can refer to the **face** and **faces-gui** workloads on port using the name **faces-front-end**.

The next step is to associate a Linkerd **AuthorizationPolicy** resource with the server. Given this Server, we can associate a Linkerd AuthorizationPolicy with the Server, to allow requests only from Emissary's identity (which is expressed with a Linkerd MeshTLSAuthentication resource):

```
---
apiVersion: policy.linkerd.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-emissary
  namespace: faces
spec:
  targetRef:
    group: policy.linkerd.io
    kind: Server
    name: faces-front-end
  requiredAuthenticationRefs:
    - group: policy.linkerd.io
      kind: MeshTLSAuthentication
      name: emissary-ingress
---
apiVersion: policy.linkerd.io/v1beta1
kind: MeshTLSAuthentication
metadata:
  name: emissary-ingress
  namespace: faces
spec:
  identities:
    # This is a single long line, don't break it!
    - "emissary-ingress.emissary.serviceaccount.identity.
      linkerd.cluster.local"
```

Applying both of those resources should magically permit the browser to talk to the GUI and the **face** workload. It won't make the application work! But it should shift us from grimacing faces to cursing faces, showing us that we have some connectivity, at least.

Adding Permission Within Faces

The reason for the cursing faces and the grey backgrounds is that we've allowed Emissary to talk to our front-end workloads, but we haven't allowed the **face** workload to talk to **smiley** or **color**, so that's our next step. We'll repeat the combination of Server, AuthorizationPolicy, and MeshTLSAuthentication, but for the Faces backend workloads. Here's the **faces-back-end** Server resource that selects HTTP traffic to the **smiley** and **color** workloads on port 80:

```
---
apiVersion: policy.linkerd.io/v1beta1
kind: Server
metadata:
  name: faces-back-end
  namespace: faces
  labels:
    app: faces-back-end
    app.kubernetes.io/part-of: faces
    project: faces
spec:
  podSelector:
    matchExpressions:
      - key: service
        operator: In
        values:
          - smiley
          - color
  port: 80
  proxyProtocol: HTTP/1
```

Here are the AuthorizationPolicy and MeshTLSAuthentication resources that allow access from the **face** workload's identity to the **faces-back-end** Server:

```
---
apiVersion: policy.linkerd.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-face-to-back-end
  namespace: faces
spec:
  targetRef:
    group: policy.linkerd.io
    kind: Server
    name: faces-back-end
  requiredAuthenticationRefs:
    - group: policy.linkerd.io
      kind: MeshTLSAuthentication
      name: face-workload
--apiVersion: policy.linkerd.io/v1beta1
kind: MeshTLSAuthentication
metadata:
  name: face-workload
  namespace: faces
spec:
  identities:
    # This is a single long line, don't break it!
    - "face.faces.serviceaccount.identity.linkerd.
cluster.local"
```

Of these three, the AuthorizationPolicy is perhaps the strangest. A helpful way to read it is that it's defining what authentication we need to allow communications to the **targetRef** – in this case, the **faces-back-end** Server.

When we apply these three resources, our application should start working again!

Locking Down Emissary

Let's go ahead and lock down Emissary as well. Switching its namespace to default-deny is the first step:

```
$ kubectl annotate ns emissary \
    config.linkerd.io/default-inbound-policy=deny
$ kubectl rollout restart -n emissary deploy
$ kubectl rollout status -n emissary deploy
```

When we do this, we'll see that we're suddenly back to grimacing faces on grey backgrounds, meaning that the Faces GUI can't talk to anything at all.

What's happening is that all traffic in the namespace is getting blocked, because we've told Linkerd to deny everything without creating any exception for ingress traffic! This creates a bit of a quandary: by definition, ingress traffic is arriving from outside the mesh, so how do we authorize it?

This is what Linkerd Skip ports are for. We can simply tell Linkerd that the traffic arriving at the Emissary Deployment on its ingress ports is not to be processed by the proxies. To do this, we annotate the Emissary Pods with

config.linkerd.io/skip-inbound-ports: 8080,8443,8877

Note that this annotation must go on the Pods, not the Deployment! The simplest way to do this is to edit the Pod template within the Deployment with **kubectl edit**. If you just try to use **kubectl annotate** on the Deployment, it (sadly) won't work.

Once we annotate the Emissary pods, we need to restart them:

```
$ kubectl rollout restart -n emissary deploy
$ kubectl rollout status -n emissary deploy
```


As soon as the restarts finish, the Faces app will start working again!

One last note: It's safe to tell Linkerd not to process traffic coming into the ingress controller from outside because, by definition, ingress controllers have to be designed to handle that. Skipping the inbound ports won't cause a security problem on its own.

Going Further

We've locked things down quite a bit, but there are a few more things we can look at.

First, we created Emissary Mappings that provide direct access to the **smiley** and **color** workloads, even though the Faces GUI only needs access to the **face** workload. We can verify that we can't actually access them now by running

```
$ curl -s -o /dev/null -w "%{http_code}" \
    -u username:password \
    https://${DEMO_HOST}/face/
$ curl -s -o /dev/null -w "%{http_code}" \
    -u username:password \
    https://${DEMO_HOST}/smiley/
$ curl -s -o /dev/null -w "%{http_code}" \
    -u username:password \
    https://${DEMO_HOST}/color/
```

You'll see that the first request will succeed, but the others won't, because Linkerd will not allow Emissary access to them.

One other point: the **face** workload responds to two different paths: the GUI uses the `/cell/...` path to fetch cells to display, but we can also use the `/rl` path to see how many RPS the **face** workload thinks it's seeing:

```
$ curl -s -u username:password \
  https://${DEMO_HOST}/face/rl
| jq
```

This is just for debugging, so we really should not allow access to it from outside. We can use an HTTPRoute to close that down:

```
---
apiVersion: policy.linkerd.io/v1beta2
kind: HTTPRoute
metadata:
  name: face-only-root
  namespace: faces
spec:
  parentRefs:
  - name: face
    kind: Service
    group: core
    port: 80
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /cell/
    backendRefs:
    - name: face
      port: 80
      weight: 80
```

Once we apply that, the Faces application will still be working - good! - but if we try the `/rl` path again, we'll get a 404:

```
$ curl -s -u username:password \
  https://${DEMO_HOST}/face/rl
```

(This is actually probably too broadly restrictive since it won't let any workload use the `/rl` path. We could couple this with an `AuthorizationPolicy` to open it up just a bit, but for now, we'll just be Draconian about it.)

Finally, if you try to look at the Viz Dashboard (which you can find at `https://${DEMO_HOST}/`), you'll find that locking down the namespace actually broke stats, too. This is obviously not ideal.

To reenable stats, we need a `Server` and an `AuthorizationPolicy` with `MeshTLSAuthentication` to allow viz traffic:

```
---
apiVersion: policy.linkerd.io/v1beta1
kind: Server
metadata:
  name: linkerd-admin
  namespace: faces
spec:
  podSelector:
    matchLabels: {}
  port: linkerd-admin
  proxyProtocol: HTTP/2
```

The `Server` itself might look a little odd, too: the null set of labels to match means that the port is the only relevant thing, so this `Server` matches the `linkerd-admin` port on any workload.

```

---
apiVersion: policy.linkerd.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-viz
  namespace: faces
spec:
  targetRef:
    kind: Namespace
    name: faces
  requiredAuthenticationRefs:
  - group: policy.linkerd.io
    kind: MeshTLSAuthentication
    name: viz-apps
---
apiVersion: policy.linkerd.io/v1beta1
kind: MeshTLSAuthentication
metadata:
  name: viz-apps
  namespace: faces
spec:
  identities:
    # Two long lines here, don't break them!
    - "prometheus.linkerd-viz.serviceaccount.identity.
linkerd.cluster.local"
    - "tap.linkerd-viz.serviceaccount.identity.linkerd.
cluster.local"

```

This AuthorizationPolicy doesn't reference the Server because it doesn't need to: if the **targetRef** is a Namespace, any traffic matching any Server in that namespace will be affected.

Once applying that, the Viz dashboard will start working again. (Note that it might take a few seconds to start updating.)

The Importance of Certificates

Everything we've shown above is completely reliant on Linkerd's workload identities, which are in turn utterly dependent on mTLS certificates. In particular, if a certificate is compromised, your workload identity is compromised. This is obviously

a dangerous situation: it's critical that you keep a close eye on your certificates. Specifically, keep the trust anchor's secret key off the cluster entirely, and make sure you rotate certificates often. Additionally, if a certificate expires, you will take downtime, so it's critical to rotate certificates well before their expiry dates. We're using cert-manager to provide these certificates for exactly these reasons: it's a straightforward way to address these requirements with minimal pain, much like Emissary is a straightforward way to safely manage traffic from the public Internet.

The Importance of Audit

We also described how to set up Linkerd policy for zero trust. Obviously, once we set this up, we don't want anyone weakening our security by changing our policies! To mitigate this situation, we can use Polaris to keep an eye on the cluster as a whole, to make sure that things aren't changed when we don't want them to be.

Summing it up: the well-tempered mesh

We opened by talking about zero trust and how it requires checking every access, every time. Linkerd is ideally suited to address this for all cluster communications, making it a critical tool for zero trust in Kubernetes — but it can't do it all by itself. Linkerd's superpower here is its ability to collaborate with other CNCF projects to provide zero trust security while maintaining operational simplicity.



BUOYANT

Interested in Buoyant Enterprise for Linkerd or need enterprise support?

The team at Buoyant, creators of Linkerd, can help.

Contact us at sales@buoyant.io or [book a meeting](#) today.

Polaris Deep Dive

By Stevie Caldwell, SRE Tech Lead at Fairwinds

A key component of any security story is maintaining a strong security posture once it has been set up to meet an organization's needs. In the previous chapters, we've seen how Emissary-ingress secures access to the cluster; how Linkerd enables secure communication between workloads; and how cert-manager provides the glue that helps these projects achieve their goals. However, security is not a "set it and forget it" practice, however. How do we ensure these security best practices persist through upgrades, new deployments, and changing teams? That's where Polaris comes in.

About Polaris

Polaris is an open source policy engine developed by Fairwinds, which includes a library of built-in configuration policies based on industry-standard best practices (including recommendations from the [NSA Hardening Guide](#)). Polaris enables users to define the configuration rules for any resource or workload in a cluster to either alert or prevent configurations that break configured security policies, whether intentionally or unintentionally. That means once the previous tools have been configured to secure a cluster, policies can be created within Polaris to catch configurations that violate those settings, helping to maintain cluster security.

Why Polaris

Teams using Kubernetes need to implement guardrails to ensure workloads do not put the organization at risk for security breaches, cloud overspend, or

performance issues. The benefits of Polaris over other open source policy engines include its out-of-the-box policies and ability to build custom policies with JSON scheme vs. learning another language. Polaris goes beyond policy enforcement to automatically remediate issues based on policy criteria when run on a command line or as a mutating webhook.

Polaris and Zero Trust

Zero trust requires policy enforcement. Polaris helps organizations implement policy around many security requirements, for example, privileged access or read-only filesystems. It will continuously check to ensure containers are not configured incorrectly or out of zero trust policies. Further, it will enforce actions if there are misconfigurations by not allowing code to reach production. It helps ensure that there are no loose hanging misconfigurations in a zero trust environment.

Defining the policies

The focus here will be on the Linkerd deployment as a target for policy enforcement, but these steps can be applied to any resource in the cluster.

The first step is to identify the policies we want to implement. As mentioned, a default installation of Polaris will use a pre-loaded [configuration file](#). These policy checks are written as YAML with an embedded JSON schema that describes the object to check against and the desired state.

Polaris also allows the ability to BYOC: Bring Your Own Checks. Policies that are specific to an organization's needs can be added to a custom config.yml file and passed in at runtime. It is recommended to use one of the pre-existing config files for reference. Documentation on creating custom checks can be found on the Polaris [website](#). In addition to checks, the config file can be used to exempt namespaces, controllers, and containers from auditing.

Linkerd has been installed to ensure secure communications between workloads via mTLS, therefore, a first reasonable policy to implement would be to check that the expected workloads are indeed part of the Linkerd service mesh. Here is an example of a check that can be used for that purpose:

```

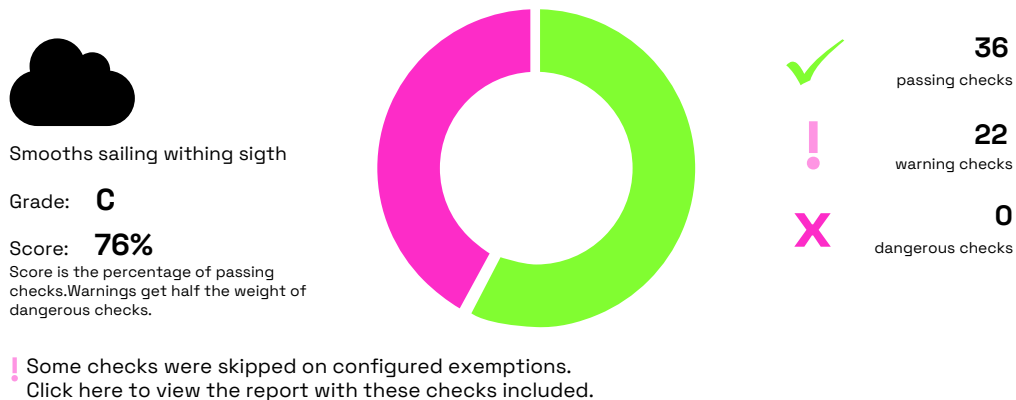
checks:
  linkerdSidecarInjected: warning # or danger/ignore
customchecks:
  linkerdSidecarInjected:
    successMessage: Linkerd sidecar has been injected
    # This is a single long line, don't break it!
    failureMessage: Linkerd sidecar should be injected to
enable mTLS
category: Security
target: PodTemplate
schema:
  '$schema': http://json-schema.org/draft-07/schema
  type: object
  properties:
    metadata:
      type: object
      required:
        - annotations
      properties:
        annotations:
          type: object
          properties:
            'linkerd.io/inject':
              type: string
              const: "enabled"
          required:
            - linkerd.io/inject
  exemptions:
    - namespace: kube-system
    - namespace: local-path-storage
    - namespace: default
    - namespace: kube-public
    - namespace: kube-node-lease

```

When a workload is added to the service mesh using either **linkerd inject** or via namespace annotation, the meshed pods will have the annotation of **linkerd.io/inject: enabled** added. This check inspects the pod template specs for every workload in the cluster for this annotation, except for workloads in the namespaces under the exempted list.

Now that there is a defined policy, the next step is to verify that current workloads adhere to that policy. Let's install Polaris! The binary can be downloaded from the [releases page](#) in the Polaris Github repo. Make sure there is a valid **kubeconfig** and a connection to the cluster that is to be audited. We can then run **polaris --config \$PATH dashboard --port 8080**, where **\$PATH** is the location of the custom config file. The dashboard can be viewed in a browser by going to **localhost:8080**.

Cluster Overview: <https://212.2.246.184:6443>



The dashboard provides an overview of the status of the cluster as it pertains to existing workloads and how they stack up to the defined Polaris configuration policies. A pod that lacks the Linkerd annotation (and that is not in an exempted namespace) would be considered in violation and flagged here. For example, in the **cert-manager** namespace, where the namespace has not been annotated and the Pods have not been manually injected, the **cert-manager** workload fails the defined check:

cert-manager workload fails the defined check:

Deployment: cert-manager

Spec: no checks applied

Pod Spec:

! Linkerd sidecar should be injected to enable mTLS (?)

Running the dashboard in this way allows users to see where violations occur, provides information that can be used to edit policies for necessary exemptions, and identifies workloads to remediate.

Once the cluster has been cleared of all existing violations, the next step would likely be to prevent violating workloads from entering the cluster to begin with. This is where running Polaris as an Admission Controller comes into play. An Admission Controller is a plugin that can intercept requests to the Kubernetes apiserver and run checks against the object before it gets persisted to etcd. The very same check defined earlier for the dashboard can be passed to Polaris in its role as an Admission Controller.

To achieve this, Polaris must be installed inside the target cluster. Helm is the preferred means of doing this. The Polaris Helm chart enables the embedding of custom checks as a Helm value. We'll create a values.yaml file that contains the same check that we passed in to the dashboard. We'll start by adding the Helm repo we need:

```
$ helm repo add fairwinds-stable\  
    https://charts.fairwinds.com/stable  
$ helm repo update
```

Once that's done, we can install Polaris itself.

```
$ helm install polaris fairwinds-stable/polaris \  
    -n polaris --create-namespace -f values.yaml \  
    --wait
```

So what happens now if a new workload is submitted to the cluster that lacks the required Linkerd annotation? Polaris checks can have a severity of danger, warning, or ignore and that controls how its validating webhook handles policy violations. Only checks that are set to danger or warning get validated, and only those set to danger will be rejected. Violations of checks with a severity of warning will get logged to the cluster events, which can be picked up using a preferred logging solution.

Now that Polaris is running in the cluster as an Admission Controller, let's try to submit a workload that violates the policy we created.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1
  namespace: demo
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
```

Notice that this Deployment spec is missing the pod template annotation that would add this pod to the service mesh. If we try to apply this broken Deployment, we will get an error:

```
$ kubectl apply -f deployment.yaml Error from server (
Polaris prevented this deployment due to configuration
problems:
- Pod: Linkerd sidecar should be injected to enable mTLS
```

Polaris prevented this deployment due to the missing Linkerd annotation. This generates an exit code of 1 as well, so an automated deployment pipeline can catch this exit code and take whatever action has been configured for this situation. To shift further left, Polaris can be run against Infrastructure as Code files, so errors can be caught before they get merged into the main branch and the deployment pipeline kicks in.

Emissary-ingress, cert-manager, and Linkerd are trusted tools for establishing a zero-trust environment. Polaris is an excellent tool for maintaining that zero-trust environment. Clusters, workloads, teams, users — all of these are subject to change, and change can be a big challenge when it comes to maintaining security. The role of Polaris in a zero-trust environment is to enforce an organization's security policies by implementing guardrails that protect against misconfigured workloads.

Fairwinds

Want Polaris across multiple clusters? Fairwinds Insights allows you to automate Polaris across clusters. Use [Insights for free](#) or [book a demo](#).



Bringing it all together

Security in the cloud-native world is complex. We use hardware we don't own and networks we don't control to manage business-critical tasks day in and day out, and we need to trust that it's safe to continue to do so. This is a tall order. The zero trust model evolved to meet it.

In this book, we've taken a tour of four very different open source projects - cert-manager, Emissary-ingress, Linkerd, and Polaris - and had experts from each project walk through not just what each project offers, but also how they can work together, using the principles of zero trust, to safeguard a single Kubernetes application.

This architecture isn't limited to our specific application, of course: it will work as a reference architecture for your applications, too. The code published in the GitHub repository is there to help guide you on your own zero trust journey.



Zero trust for the enterprise

Of course, as every enterprise architect knows, it's one thing to prototype with an open source project, and often quite a different experience to incorporate it into the enterprise environment while meeting stringent standards, policies, and controls around security, risk mitigation, and compliance.

If your organization is facing this challenge, the Ambassador Labs, Buoyant, Fairwinds, and Venafi teams are here for you. Each of these companies has years of experience not just shipping world-class open source code, but helping enterprises around the world adopt these projects successfully in their environments. If you want help adopting or even evaluating any of these projects, please reach out! We'd be happy to help.

Contact us at:



Emissary-ingress creators

[BOOK A DEMO](#)



Linkerd creators

[BOOK A DEMO](#)



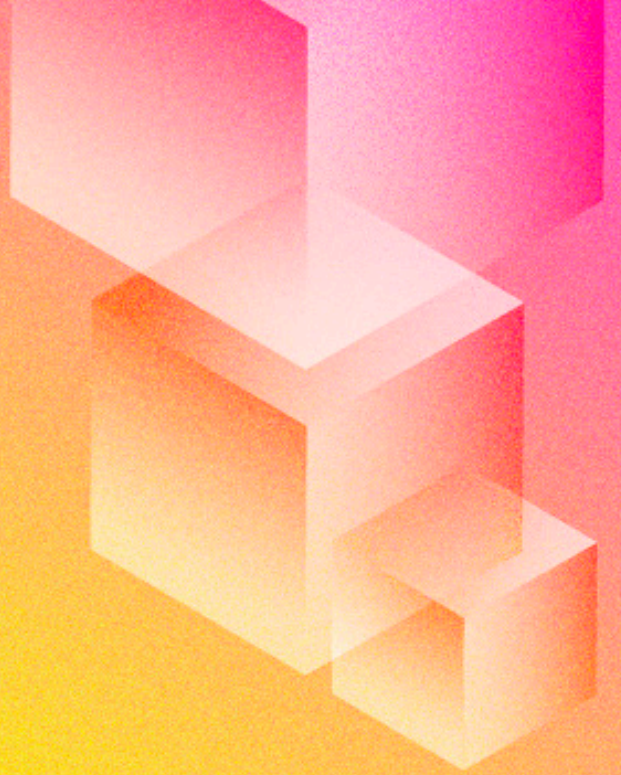
Polaris creators

[BOOK A DEMO](#)



cert-manager maintainers

[BOOK A DEMO](#)



**KUBE
CRASH**

All rights reserved KubeCrash 2023