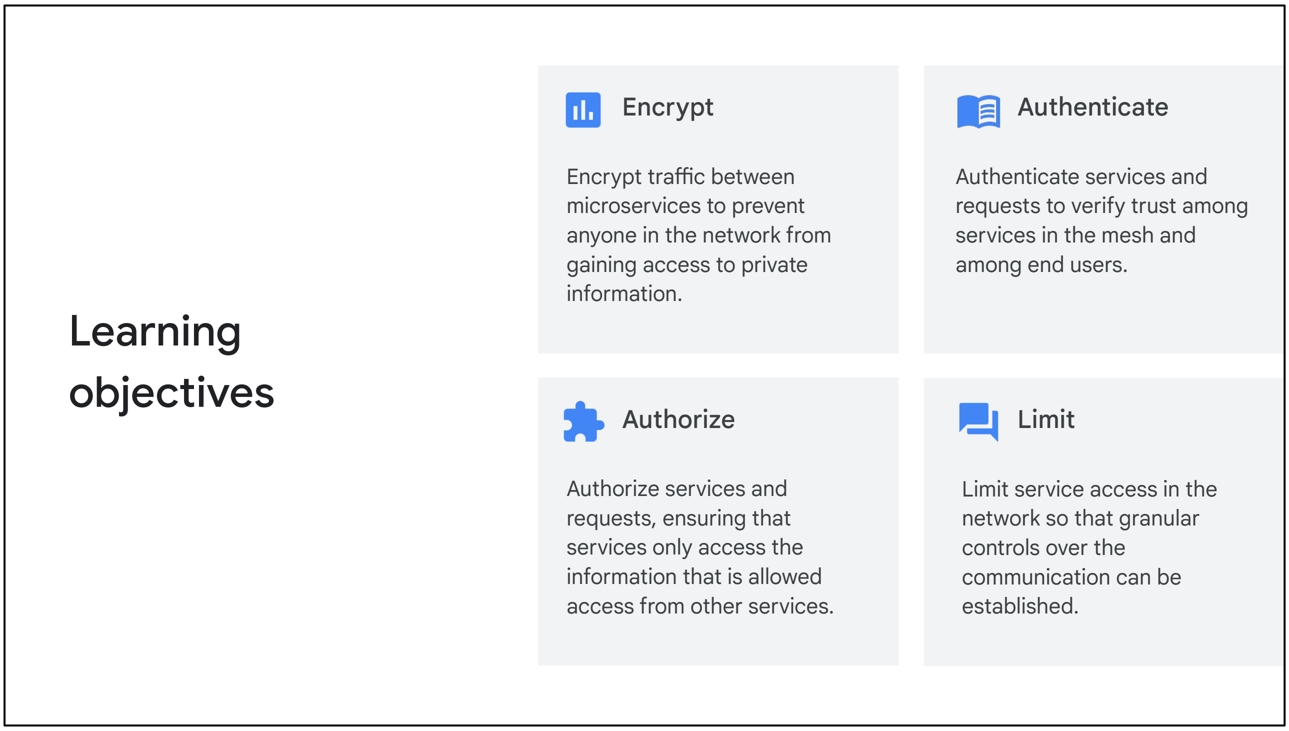
Google Cloud

Securing Network Traffic with Anthos Service Mesh

Welcome to the introduction of Anthos Service Mesh.



Learning objectives

Encrypt

Encrypt traffic between microservices to prevent anyone in the network from gaining access to private information.

Authenticate

Authenticate services and requests to verify trust among services in the mesh and among end users.

Authorize

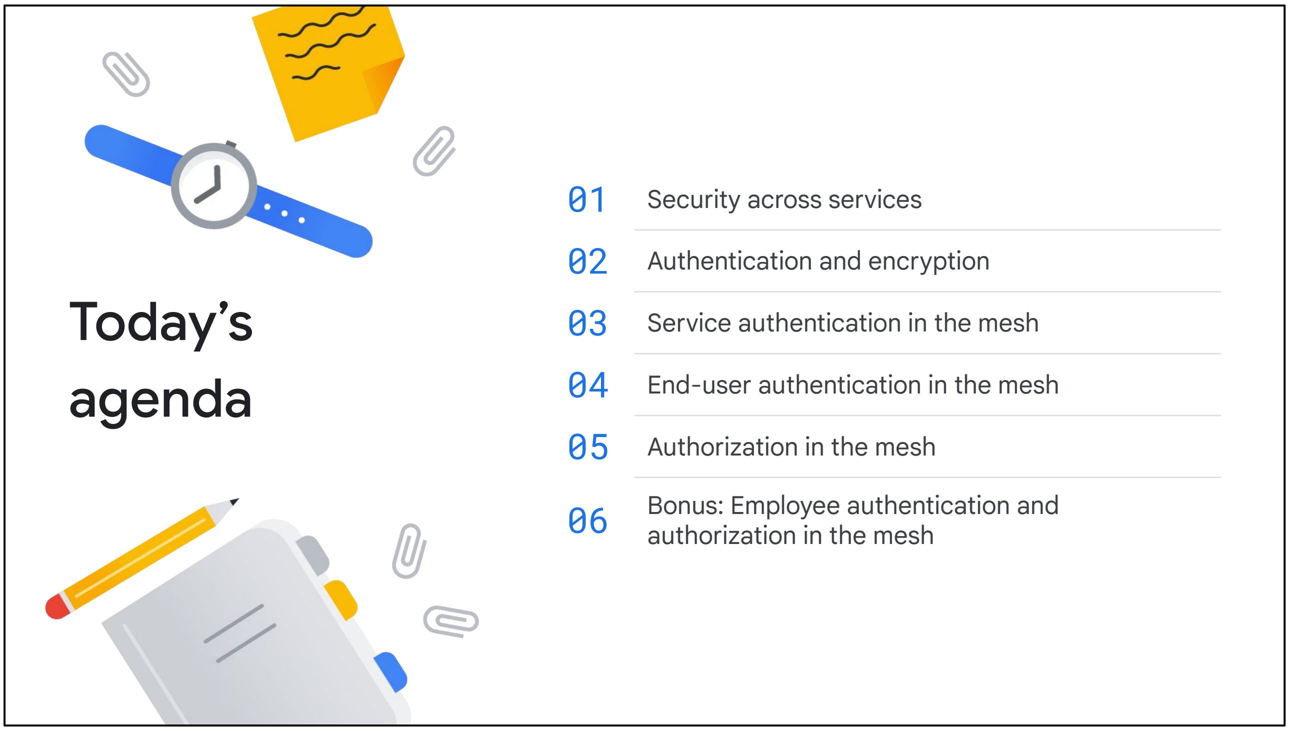
Authorize services and requests, ensuring that services only access the information that is allowed access from other services.

Limit

Limit service access in the network so that granular controls over the communication can be established.

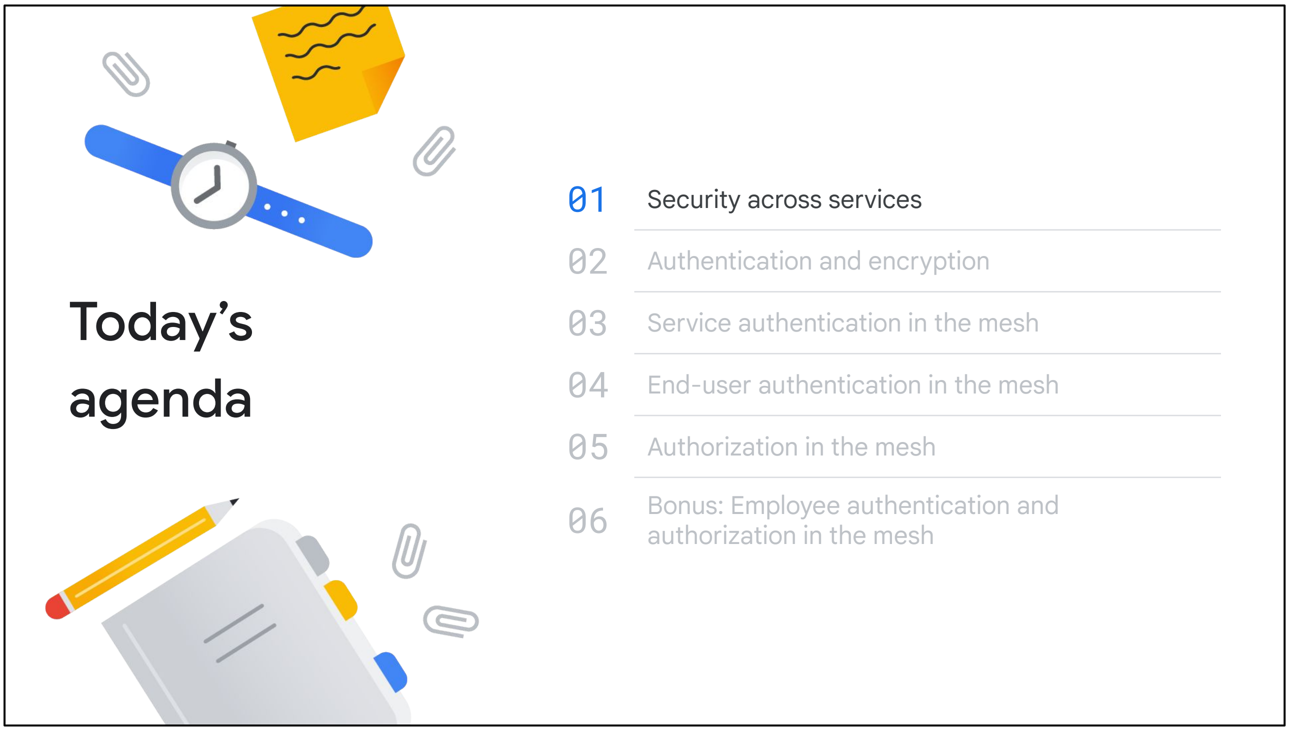
In this module, you learn to:

- Encrypt traffic between microservices to avoid anyone in the network gaining access to private information.
- Authenticate services and requests, verifying trust among services in the mesh as well as end users.
- Authorize services and requests, making sure services only access the information that is allowed access from other services.
- Limit service access in the network, so that granular controls over the communication can be established.



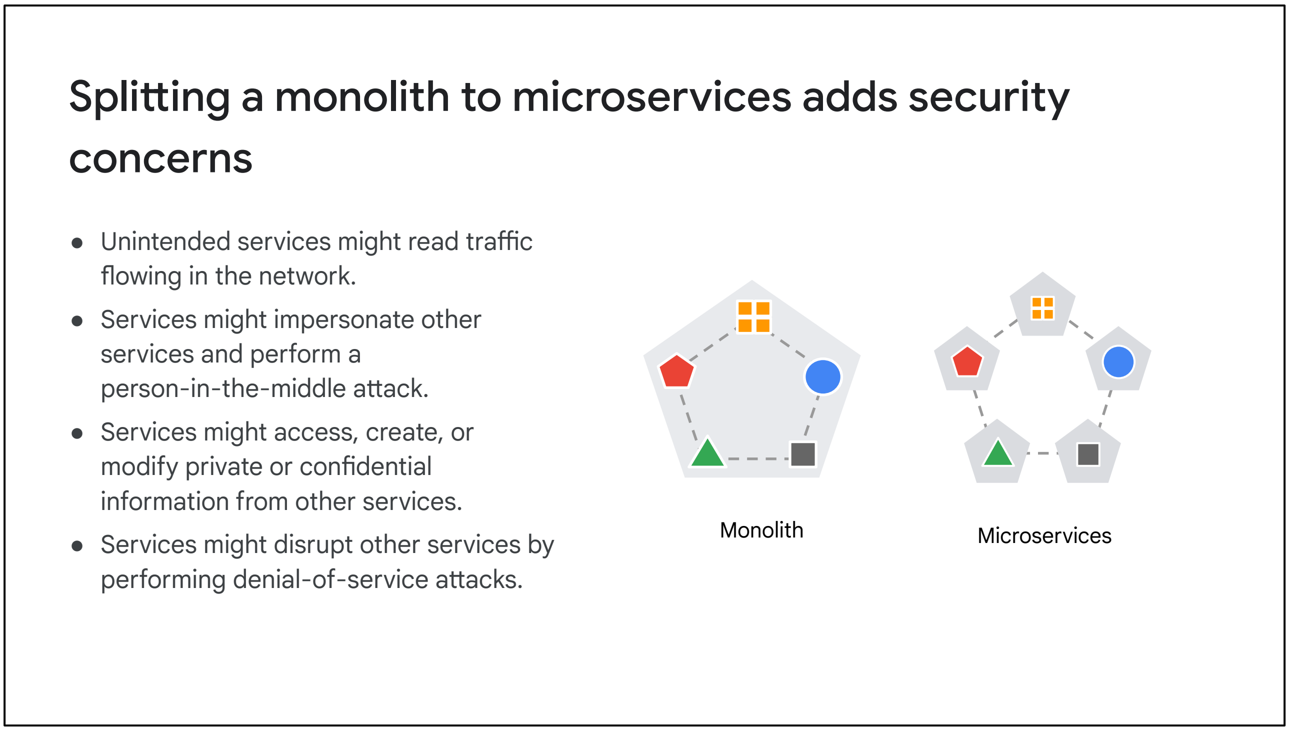
Today's agenda

This is our agenda for the module, shown on the slide. Let's get started.



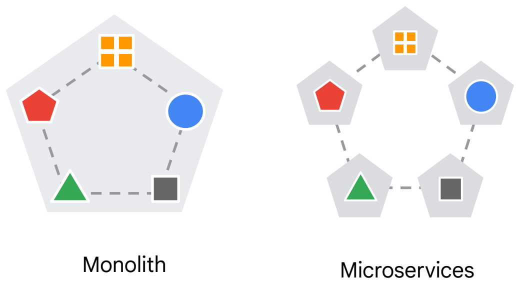
Today's agenda

We start by describing why a service mesh is useful.



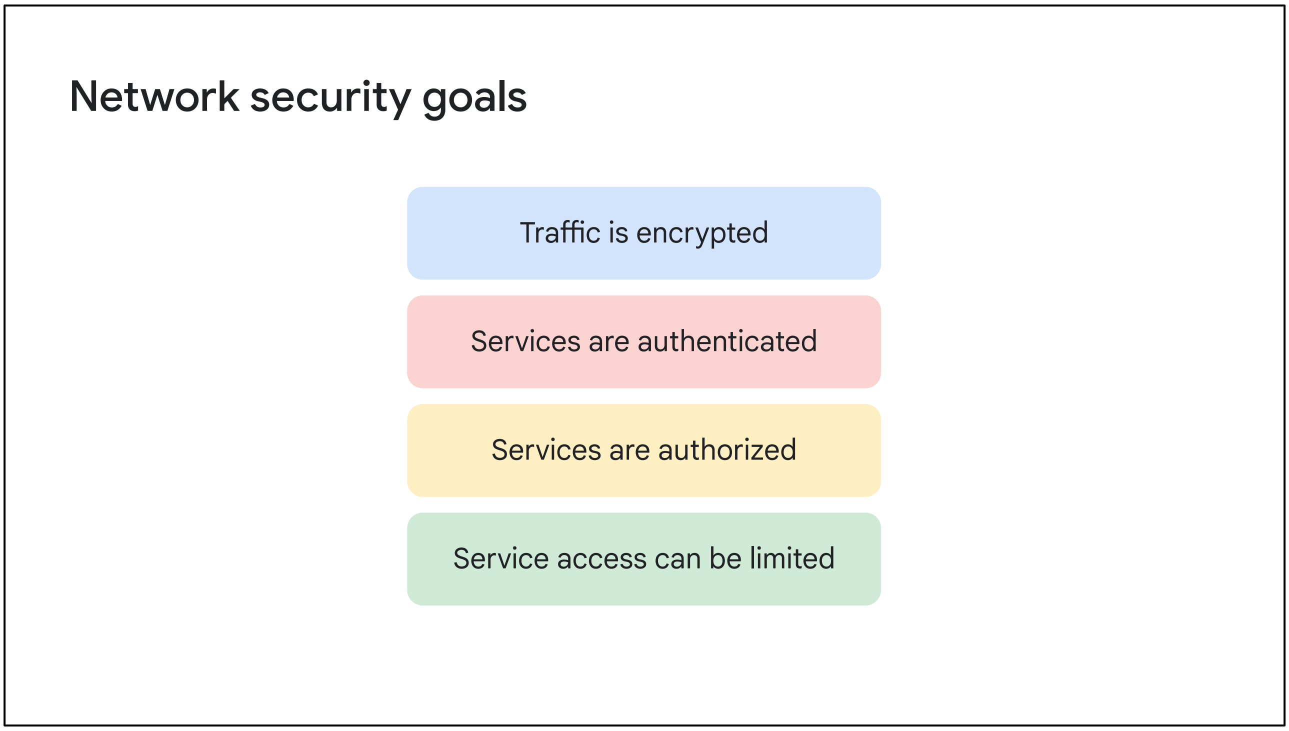
Splitting a monolith to microservices adds security concerns

- Unintended services might read traffic flowing in the network.
- Services might impersonate other services and perform a person-in-the-middle attack.
- Services might access, create, or modify private or confidential information from other services.
- Services might disrupt other services by performing denial-of-service attacks.



Breaking down a monolithic application into atomic services offers various benefits, including better agility, better scalability, and better ability to reuse services. However, microservices also introduce security concerns:

- Unintended services might read traffic flowing in the network.
- Services might impersonate other services and perform a person-in-the-middle attack.
- Services might access, create, or modify private or confidential information from other services.
- Services might disrupt other services by performing denial of service attacks.



Network security goals

Traffic is encrypted

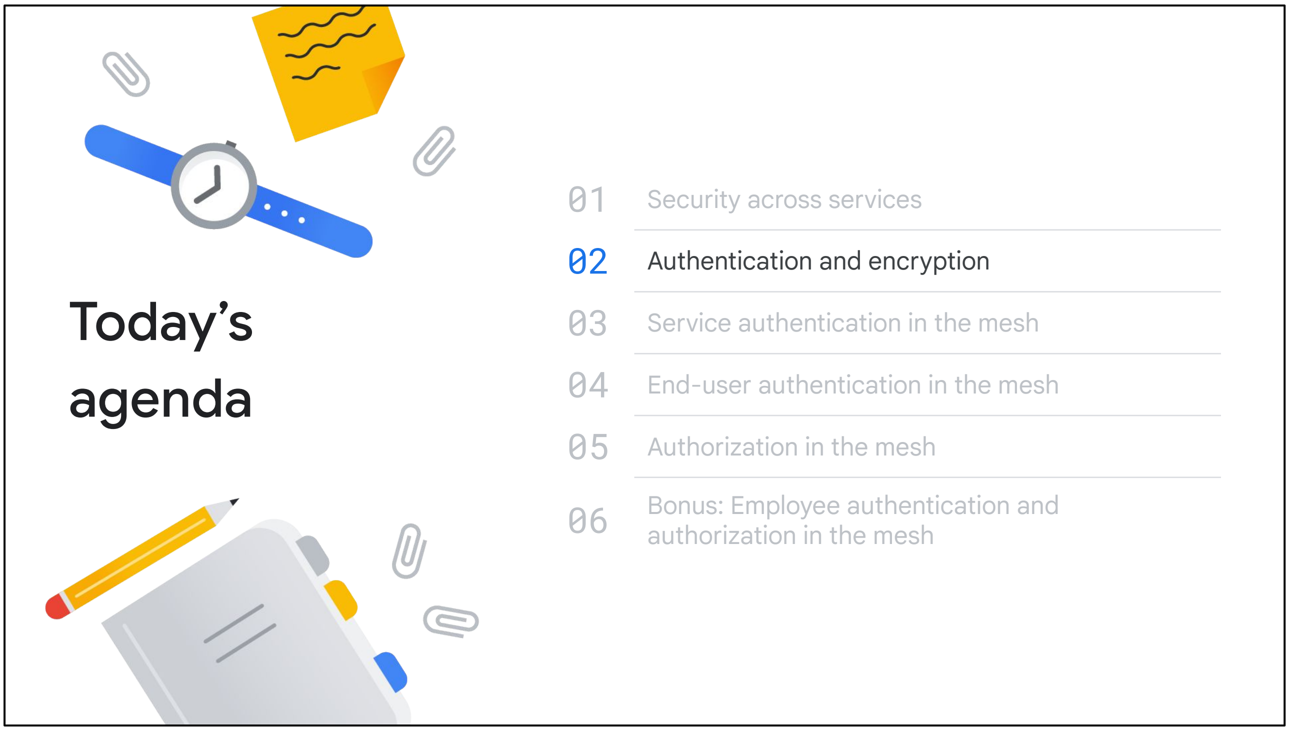
Services are authenticated

Services are authorized

Service access can be limited

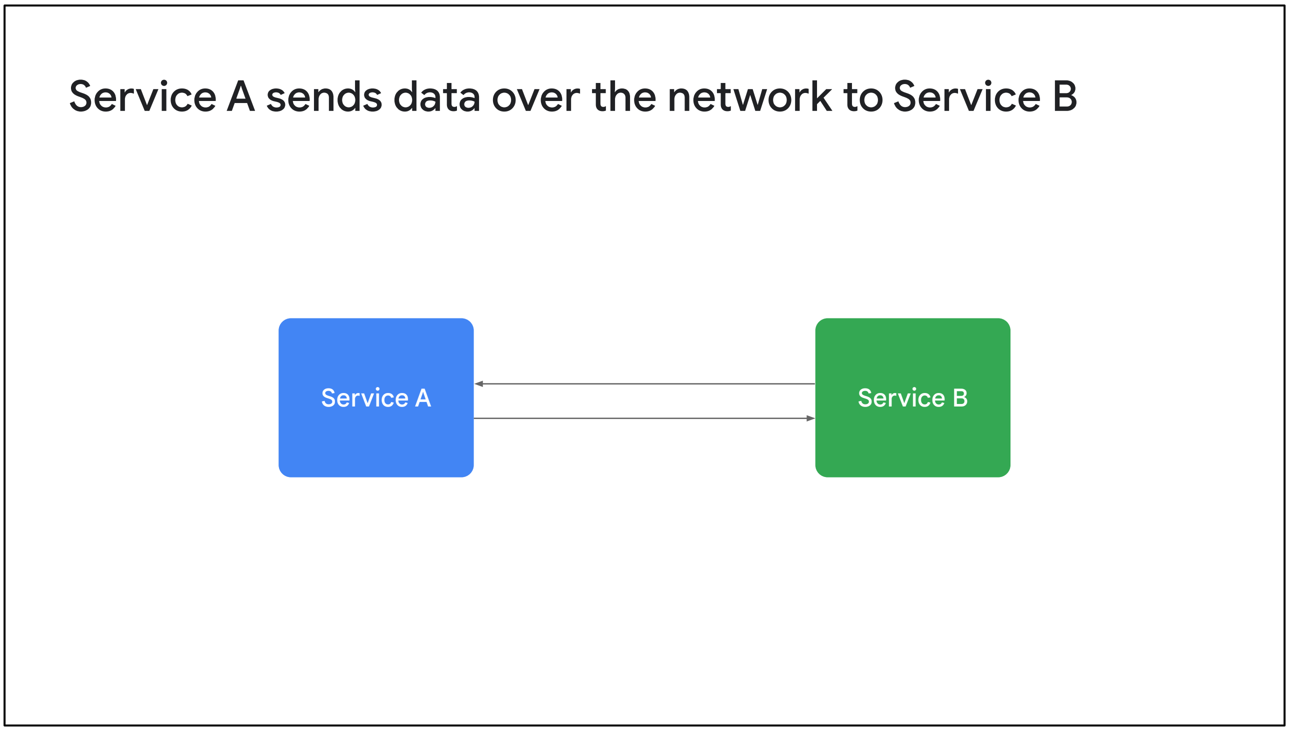
To solve those issues, we should have some basic network security goals:

- Traffic between microservices should be encrypted, that way avoiding anyone in the network gaining access to information.
- Services should be authenticated, being able to verify the service you are communicating with to establish trust.
- Services should be authorized, so that services only access the information that is allowed access from other services.
- Once our services are authenticated and authorized, we might choose to add other access limitations such as quotas to control the load.



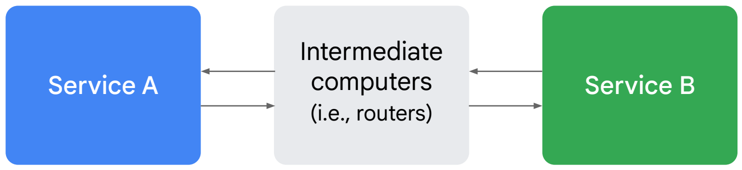
Today's agenda

We saw the importance of having security goals in a network. Let's see how authentication and encryption work.



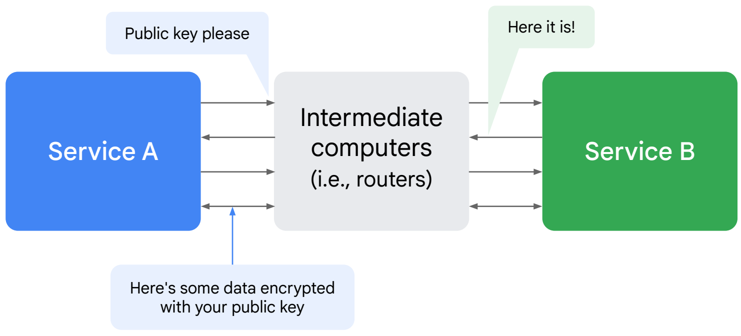
Let’s conceptualize how services communicate. Service A sends data over the network to Service B.

There is no direct line between services

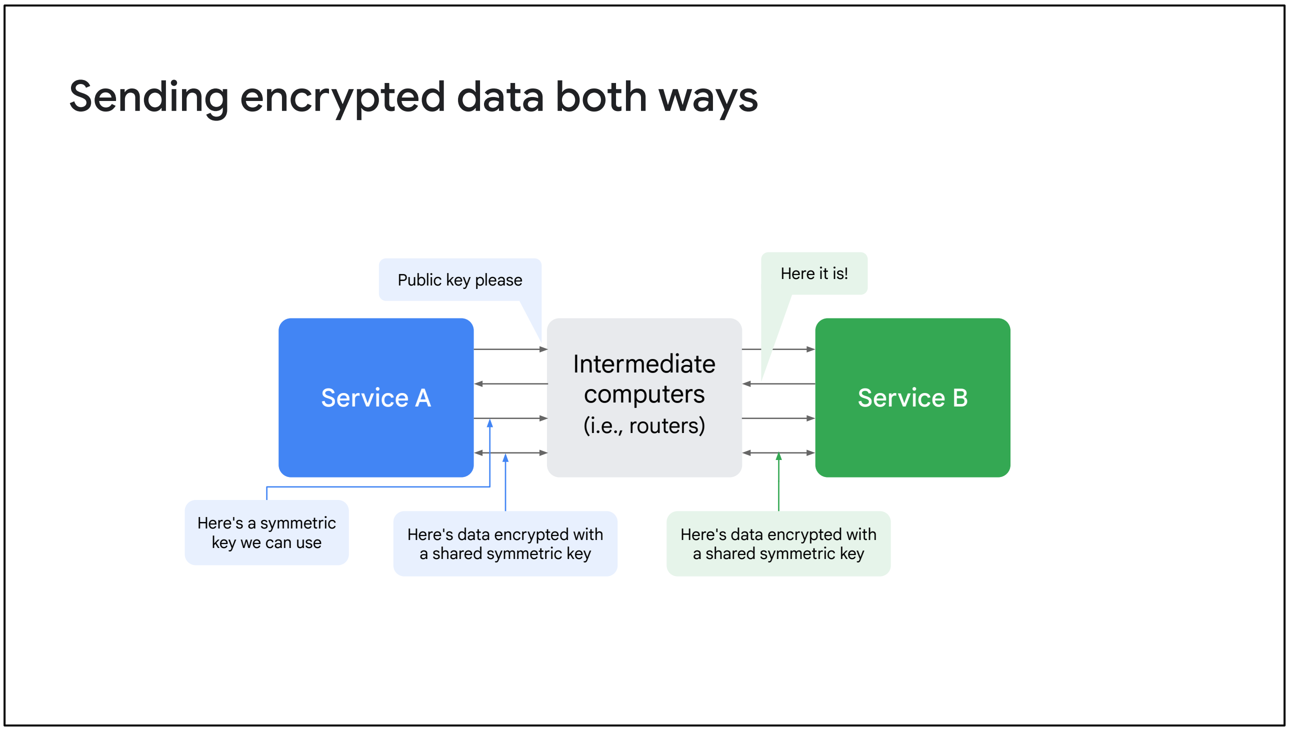


However, this communication does not happen directly, but rather, there might be a web of intermediaries such as routers, switches, or other servers. All these intermediaries were able to read the data and even modify it, and Service A cannot be sure that the original request actually arrived at Service B

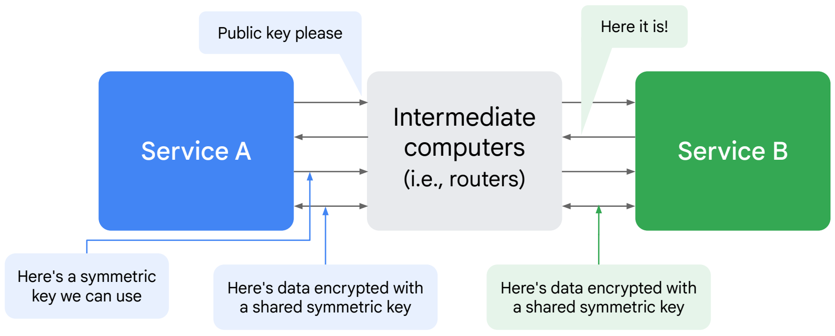
Encrypt the data with TLS so intermediaries cannot read it


Public key please
Here it is!
Service A
Intermediate computers (i.e., routers)
Service B
Here's some data encrypted with your public key

To ensure that data is not read by intermediaries, we can use the Transport Layer Security protocol, or TLS, to encrypt it. To perform this process, we use two keys. A public key is used to encrypt data, and a private key is used to decrypt data. Server A will use the public key from Server B to encrypt the message. Then Server B will use its own private key to decrypt the received message. This process is called asymmetric encryption since we are only encrypting the sent message.

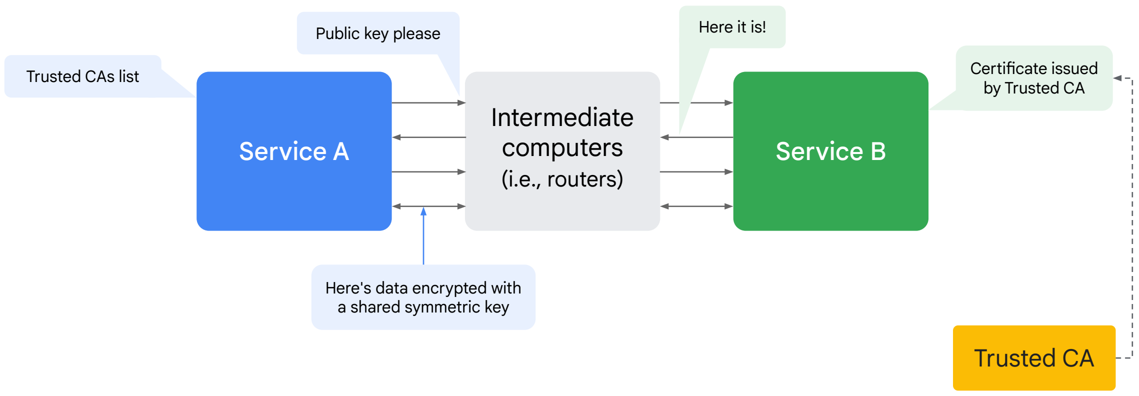


Sending encrypted data both ways


Public key please
Here it is!
Service A
Intermediate computers (i.e., routers)
Service B
Here's a symmetric key we can use
Here's data encrypted with a shared symmetric key
Here's data encrypted with a shared symmetric key

However, in a microservice architecture, we are going to have services communicating both ways, so traffic must also be encrypted from Service B to Service A. Therefore, when Service A establishes the connection with Service B, it will share a symmetric key, which both services can use to encrypt and decrypt messages. Since services A and B are the only who have access to the symmetric key, nobody else can decrypt the information.

Authenticating Service B

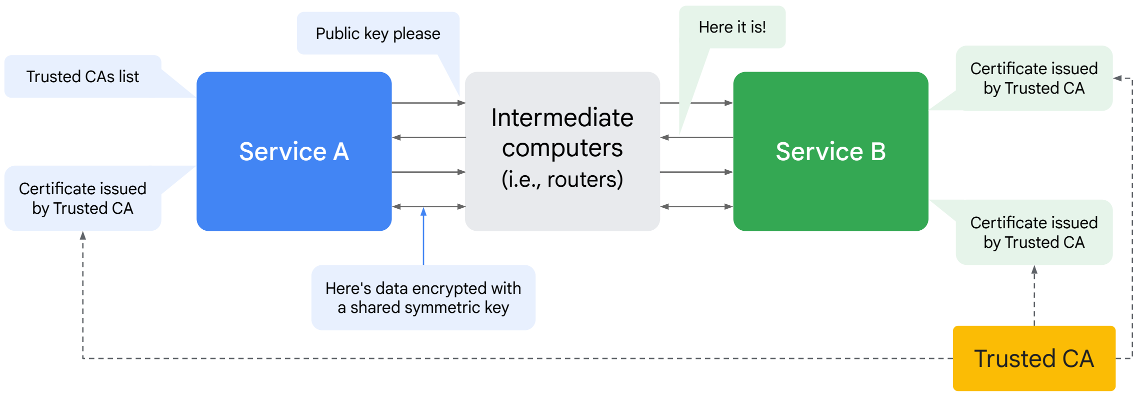

Trusted CAs list
Public key please
Service A
Intermediate computers (i.e., routers)
Here it is!
Service B
Certificate issued by Trusted CA
Here's data encrypted with a shared symmetric key
Trusted CA

There’s another problem though. What if an intermediate computer intercepts the traffic and claims to be Service B?

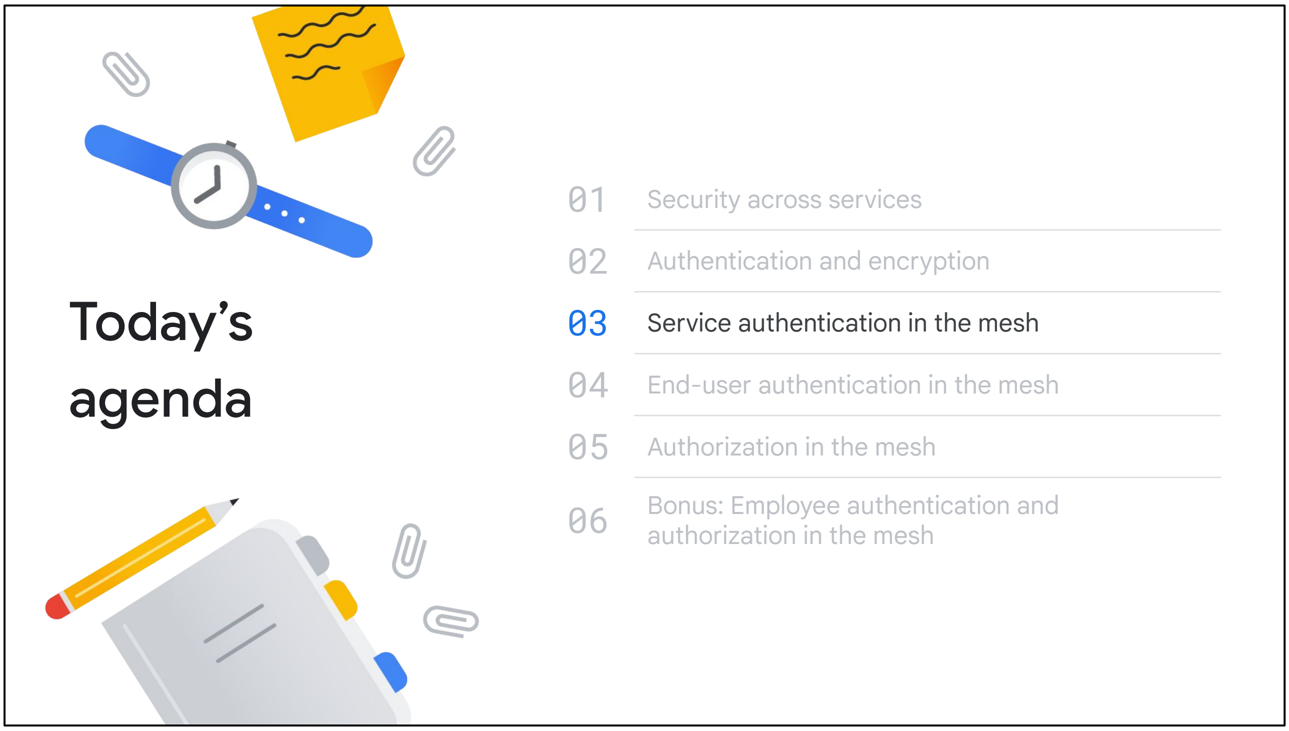
We need a way to ensure that Service B, which is providing the public key, is really Service B.

Part of the TLS standard is also designed to solve this problem. Specifically, when Service A first tries to establish an encrypted connection, it not only shares a symmetric key but also asks Service B for a certificate of its identity, in the form of an X.509 certificate. Service A then asks a trusted adviser called a Certificate Authority (or CA) that proves that the certificate is valid and proceeds based on what the CA says. This is how websites running over HTTPs work.

Authenticating and encrypting service-to-service communication with mTLS


Public key please
Here it is!
Trusted CAs list
Service A
Intermediate computers (i.e., routers)
Service B
Certificate issued by Trusted CA
Certificate issued by Trusted CA
Certificate issued by Trusted CA
Here's data encrypted with a shared symmetric key
Trusted CA

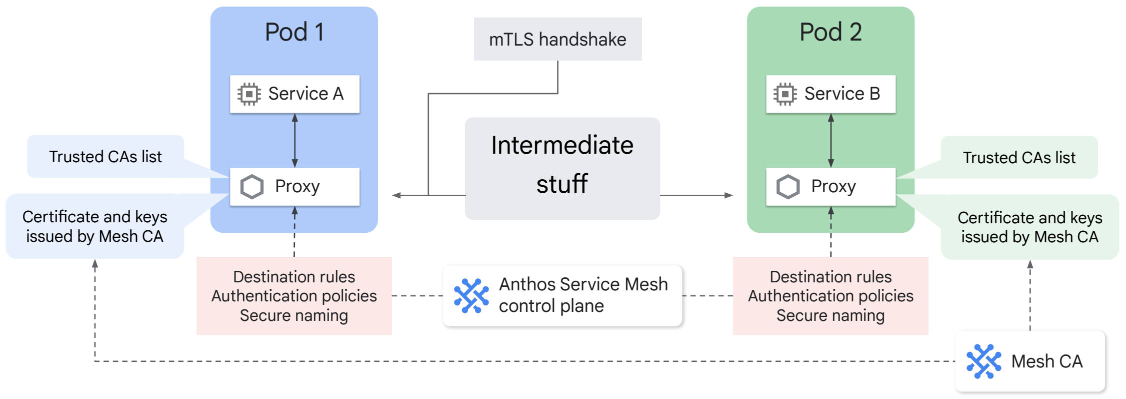
Since we need authentication and encryption both ways, when Service A wants to establish the connection with Service B, both certificates are exchanged and checked against the Trusted CA and the symmetric key is used to encrypt the messages. The process of authenticating services and encrypting messages both ways is called mutual TLS or mTLs.



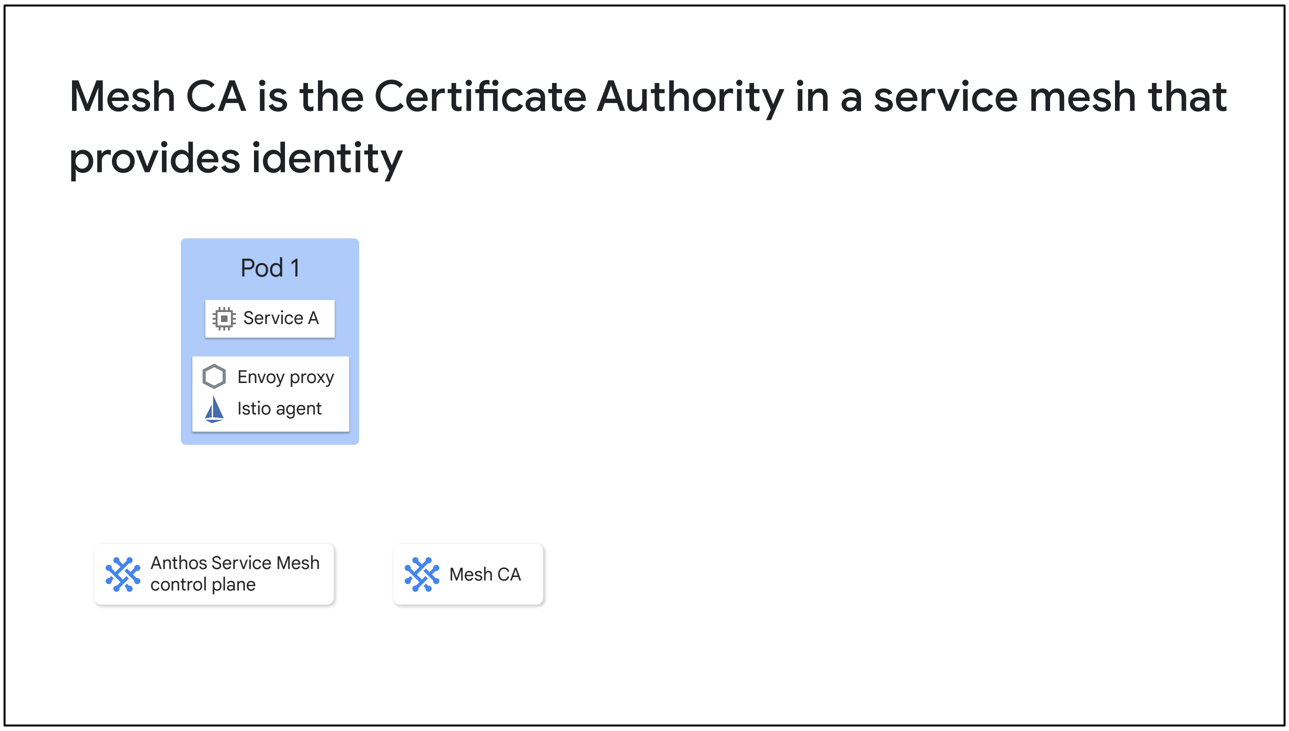
Today's agenda

Now that we have seen how authentication and encryption work in general, let's see specifically how it's implemented in Anthos Service Mesh.

Anthos Service Mesh offers a managed CA and tools to control the communication protocol

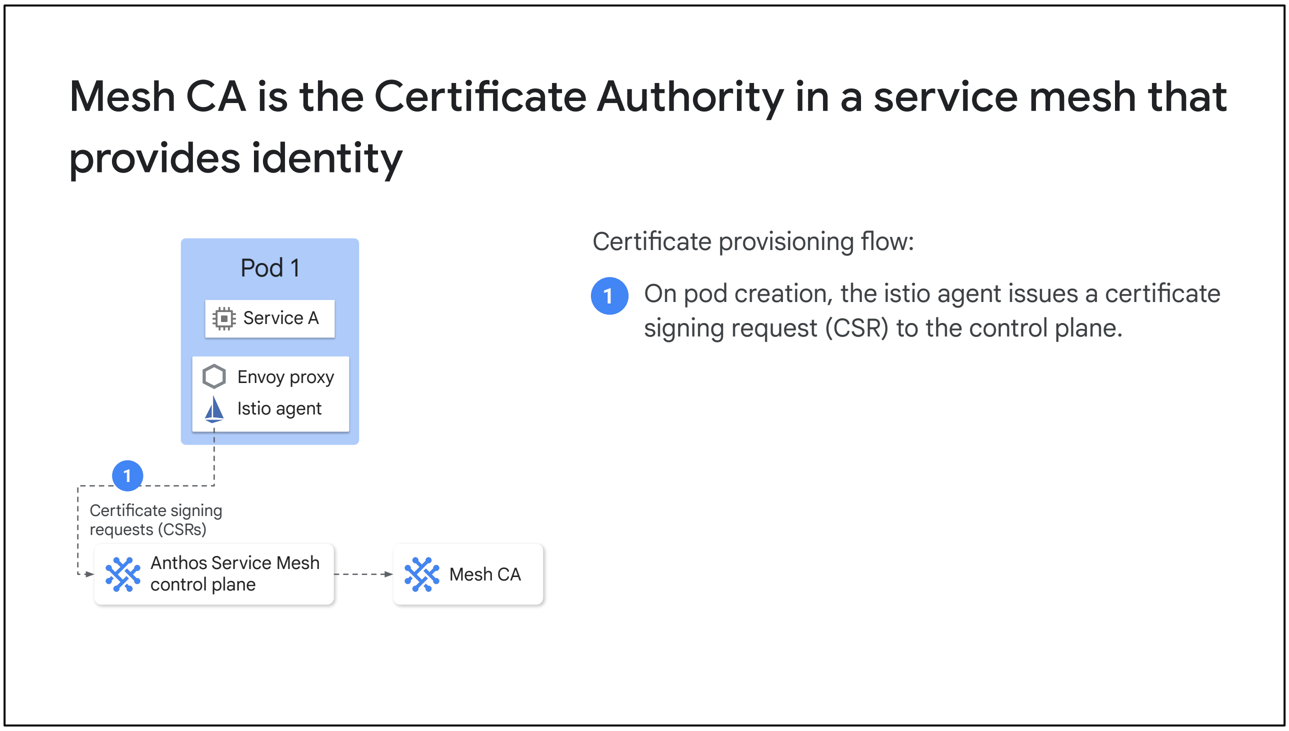


Anthos Service Mesh offers a managed CA and tools to control the communication protocol. When pods are created, Mesh CA shares the certificates and keys that are later used to verify identity in the network. Admins control the communication protocol using destination rules and authentication policies. Those policies are applied to the sidecar proxies, which work as Policy Enforcement Endpoints, or PEPs, and secure the communication between client and server. That way, the traffic can be secured with mTLS between services in the mesh.

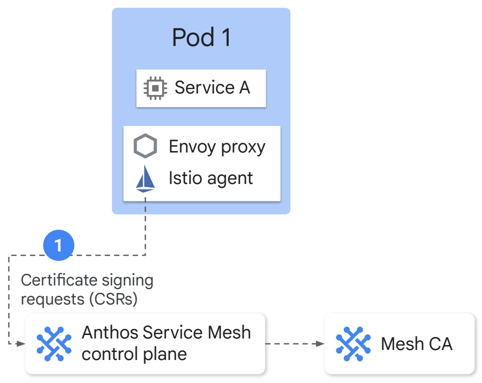


Mesh CA is the Certificate Authority in a service mesh that provides identity

Mesh CA is the Certificate Authority in a service mesh and it is responsible for distributing, rotating, and managing certificates to provide identity. The provisioning flow looks as follows:



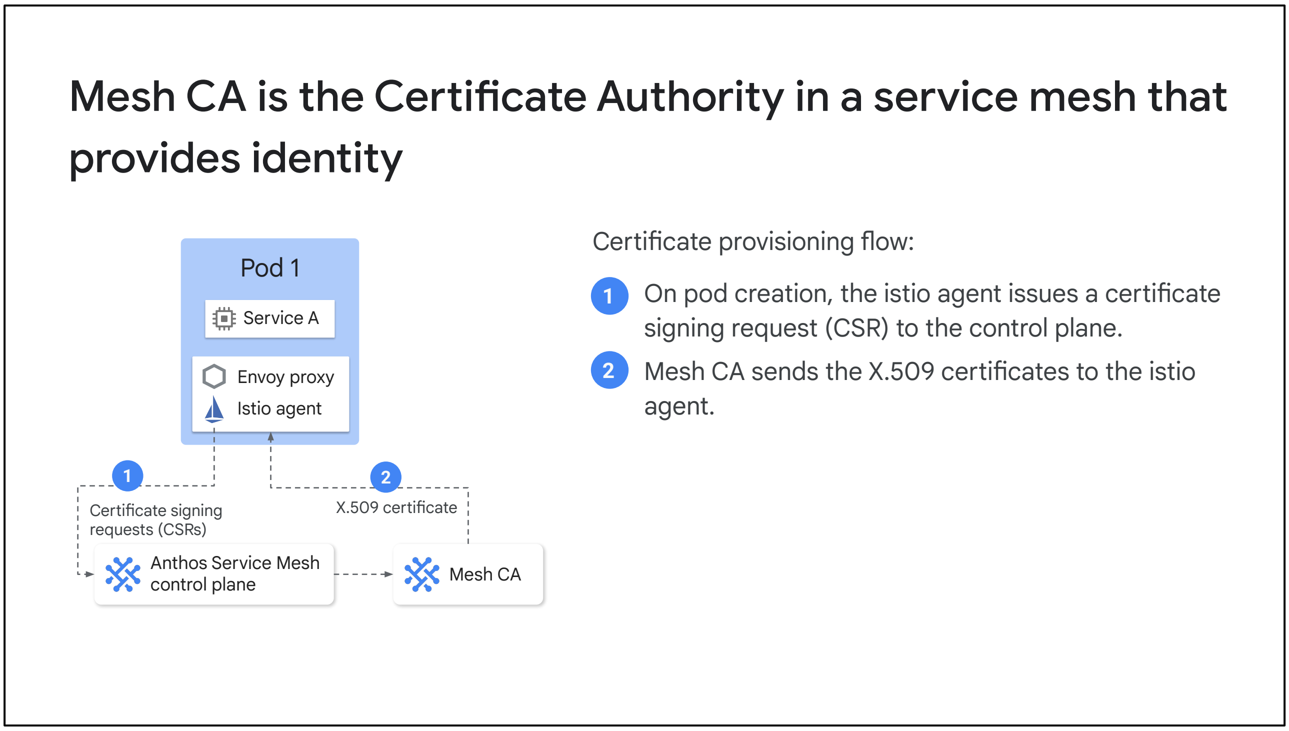
Mesh CA is the Certificate Authority in a service mesh that provides identity


Pod 1
Service A
Envoy proxy
Istio agent
1
Certificate signing requests (CSRs)
Anthos Service Mesh control plane
Mesh CA

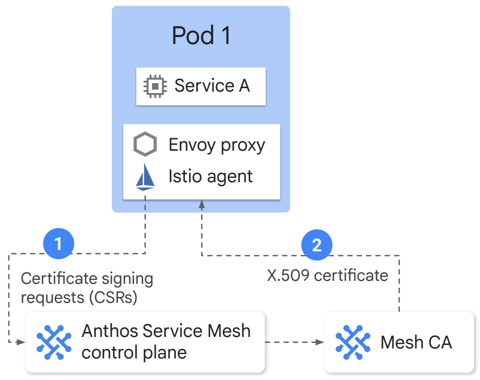
Certificate provisioning flow:

1. On pod creation, the istio agent issues a certificate signing request (CSR) to the control plane.

1. The pod gets created and the sidecar proxy is injected. The sidecar contains the Envoy proxy and the istio agent, which is responsible for communicating with the control plane over gRPC. The istio agent issues a certificate signing request, or CSR, to the control plane, which communicates with Mesh CA.



Mesh CA is the Certificate Authority in a service mesh that provides identity

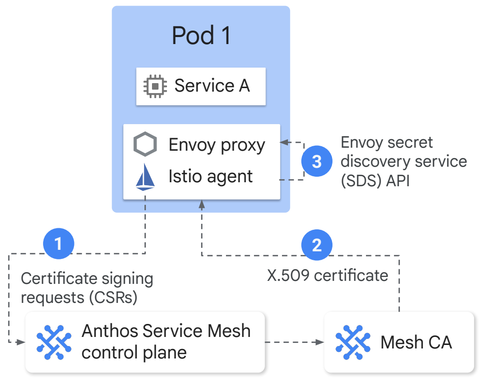

Pod 1
Service A
Envoy proxy
Istio agent
1
Certificate signing requests (CSRs)
2
X.509 certificate
Anthos Service Mesh control plane
Mesh CA

Certificate provisioning flow:

1. On pod creation, the istio agent issues a certificate signing request (CSR) to the control plane.
2. Mesh CA sends the X.509 certificates to the istio agent.

2. Mesh CA sends the X.509 certificates to the istio agent.

Mesh CA is the Certificate Authority in a service mesh that provides identity

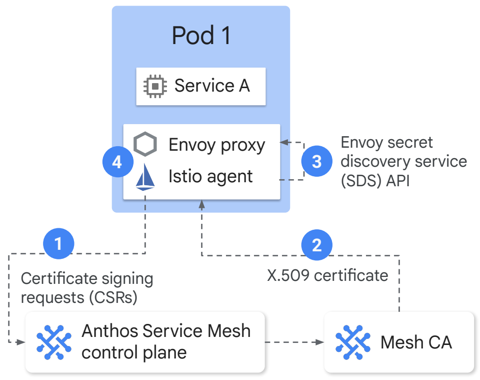

Pod 1
Service A
Envoy proxy
Istio agent
3
Envoy secret discovery service (SDS) API
1
Certificate signing requests (CSRs)
2
X.509 certificate
Anthos Service Mesh control plane
Mesh CA

Certificate provisioning flow:

1. On pod creation, the istio agent issues a certificate signing request (CSR) to the control plane.
2. Mesh CA sends the X.509 certificates to the istio agent.
3. The istio agent sends the certificates and the private key to Envoy via the Envoy SDS API.

3. The Istio agent sends the certificates received from Mesh CA and the private key to Envoy via the Envoy SDS API.

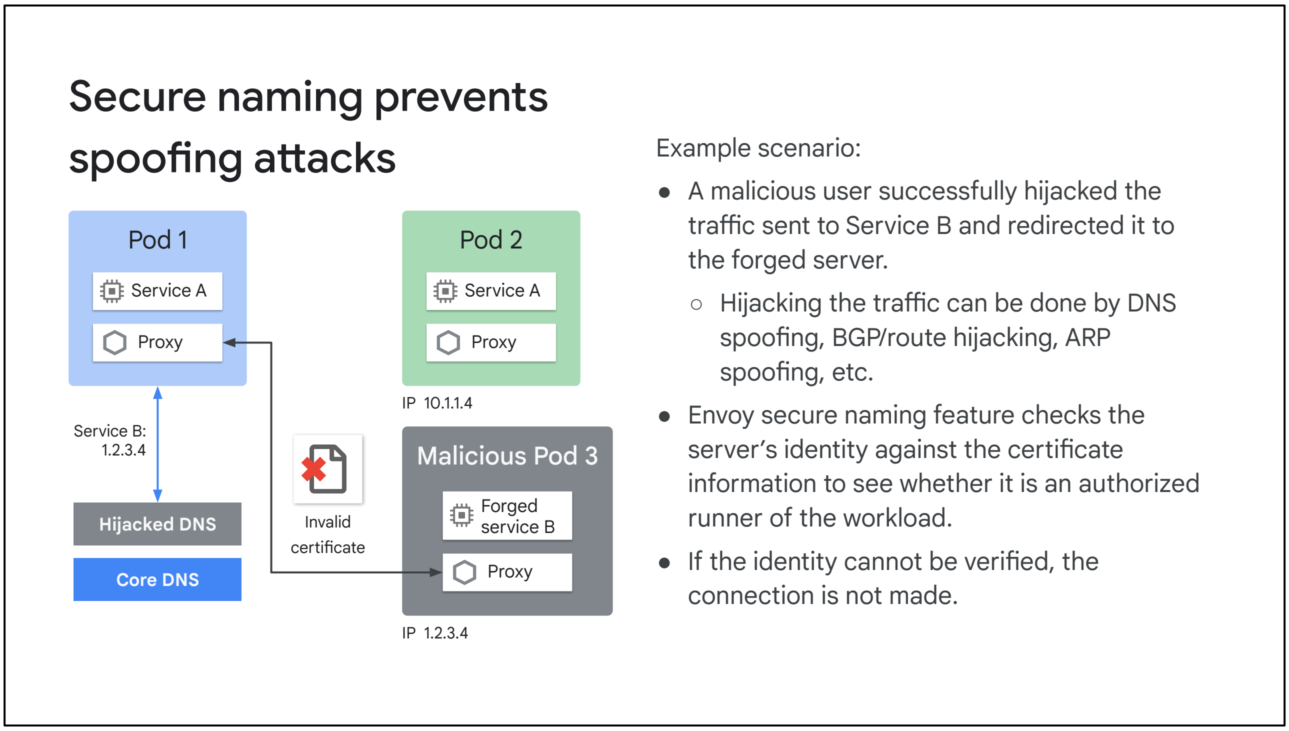
Mesh CA is the Certificate Authority in a service mesh that provides identity


Pod 1
Service A
Envoy proxy
Istio agent
4
3
Envoy secret discovery service (SDS) API
1
Certificate signing requests (CSRs)
2
X.509 certificate
Anthos Service Mesh control plane
Mesh CA

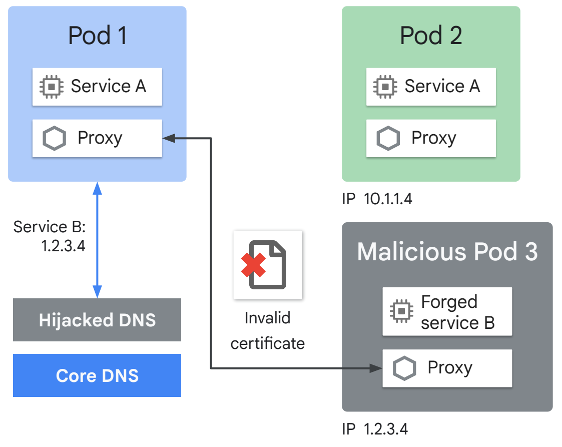
Certificate provisioning flow:

1. On pod creation, the istio agent issues a certificate signing request (CSR) to the control plane.
2. Mesh CA sends the X.509 certificates to the istio agent.
3. The istio agent sends the certificates and the private key to Envoy via the Envoy SDS API.
4. The istio agent monitors the expiration of the certificate, and the process repeats periodically for certificate and key rotation.

4. The istio agent monitors the expiration of the certificate and the process repeats periodically for certificate and key rotation.



Secure naming prevents spoofing attacks



Example scenario:

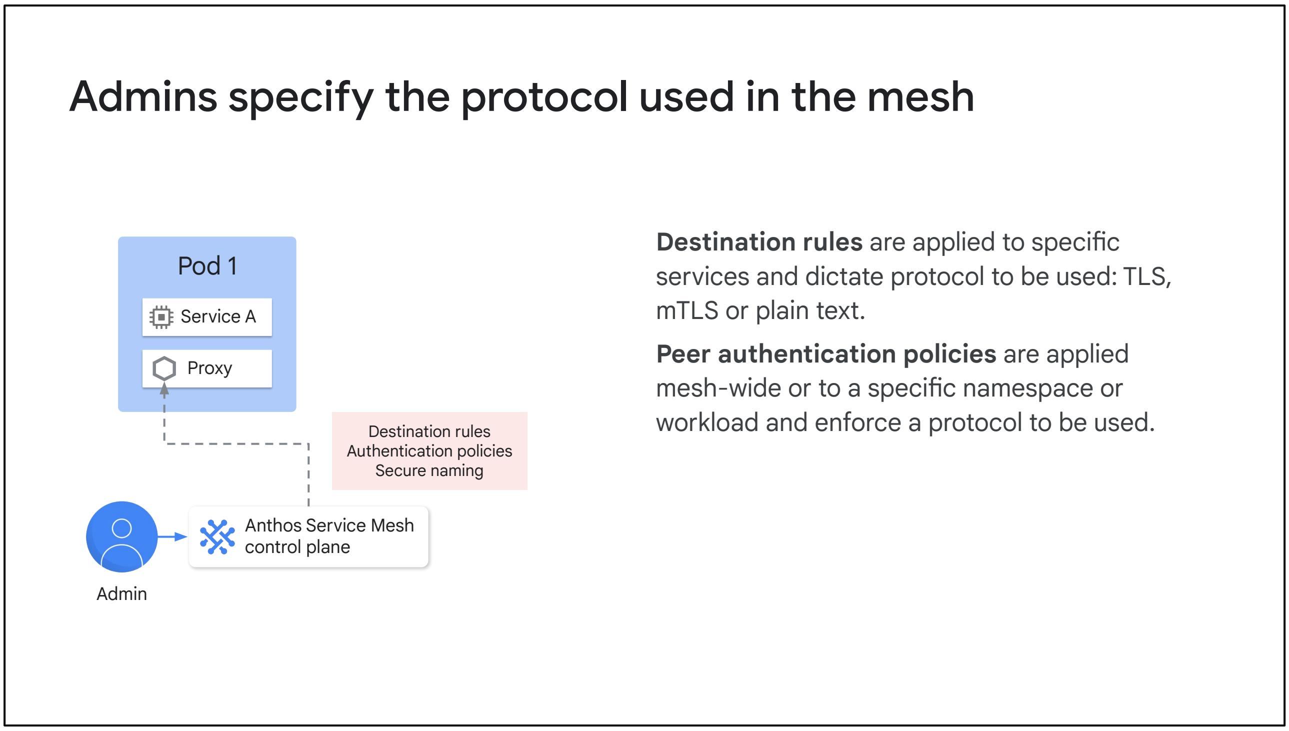
- A malicious user successfully hijacked the traffic sent to Service B and redirected it to the forged server.
  - Hijacking the traffic can be done by DNS spoofing, BGP/route hijacking, ARP spoofing, etc.
- Envoy secure naming feature checks the server's identity against the certificate information to see whether it is an authorized runner of the workload.
- If the identity cannot be verified, the connection is not made.

Secure naming is a mesh feature that enables envoys to check the server's identity against the certificate information to see if it is an authorized runner of the workload.

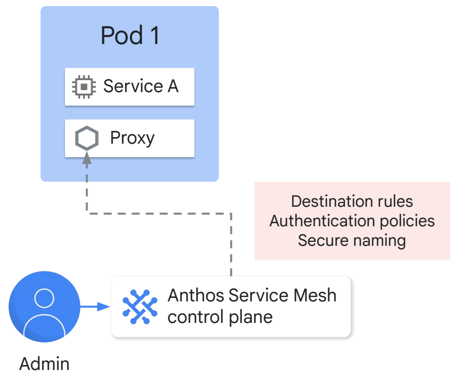
Suppose that a malicious user successfully hijacked the traffic sent to Service B and redirected it to the forged server.

In this example, we see a hijacked DNS service, but an attacker could have also hijacked the BGP protocol or spoofed the ARP protocol.

When the services try to establish the connection, the proxy secure naming feature will not be able to verify the identity of the certificate provided by pod 3 and the connection will not be performed.



Admins specify the protocol used in the mesh



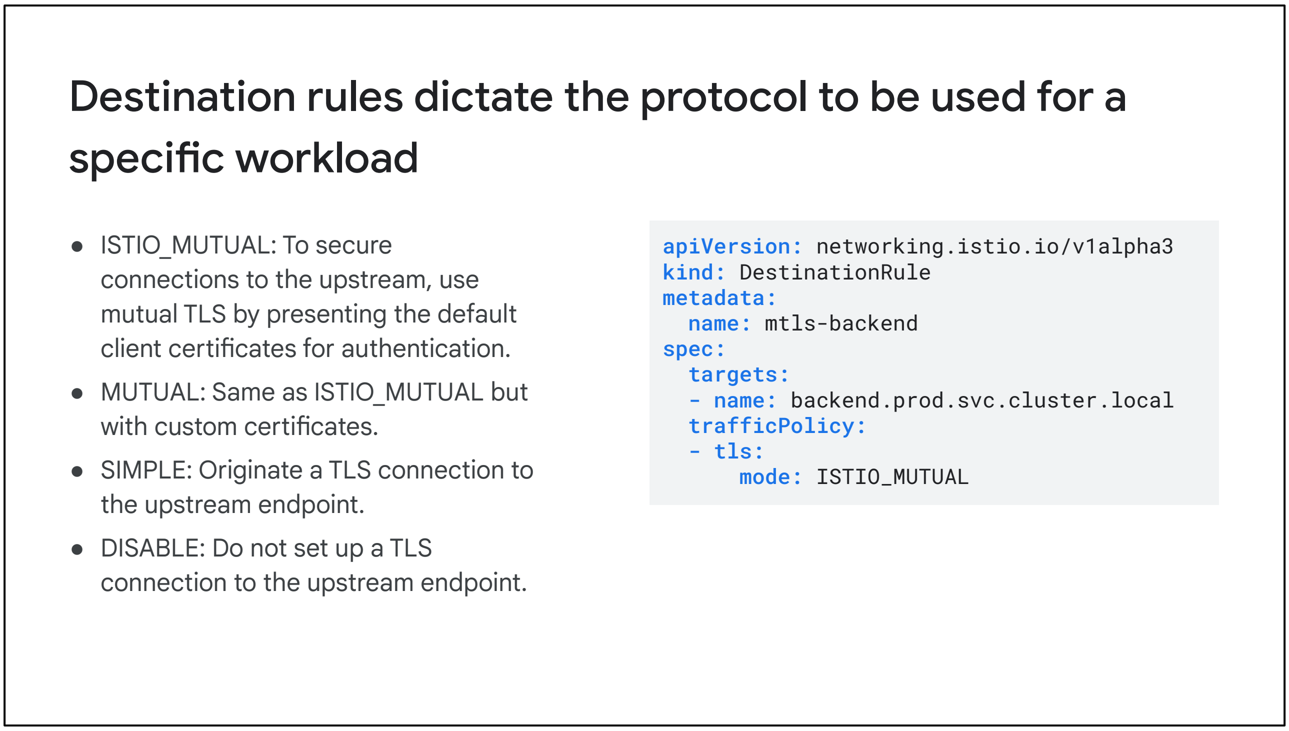
Destination rules are applied to specific services and dictate protocol to be used: TLS, mTLS or plain text.

Peer authentication policies are applied mesh-wide or to a specific namespace or workload and enforce a protocol to be used.

Administrators specify the protocol used in the mesh by setting up destination rules and peer authentication policies.

Destination rules are applied to specific services and dictate protocol to be used, that is TLS, mTLS, or plain text.

Peer authentication policies are applied mesh-wide or to a specific namespace or workload and enforce a protocol to be used.



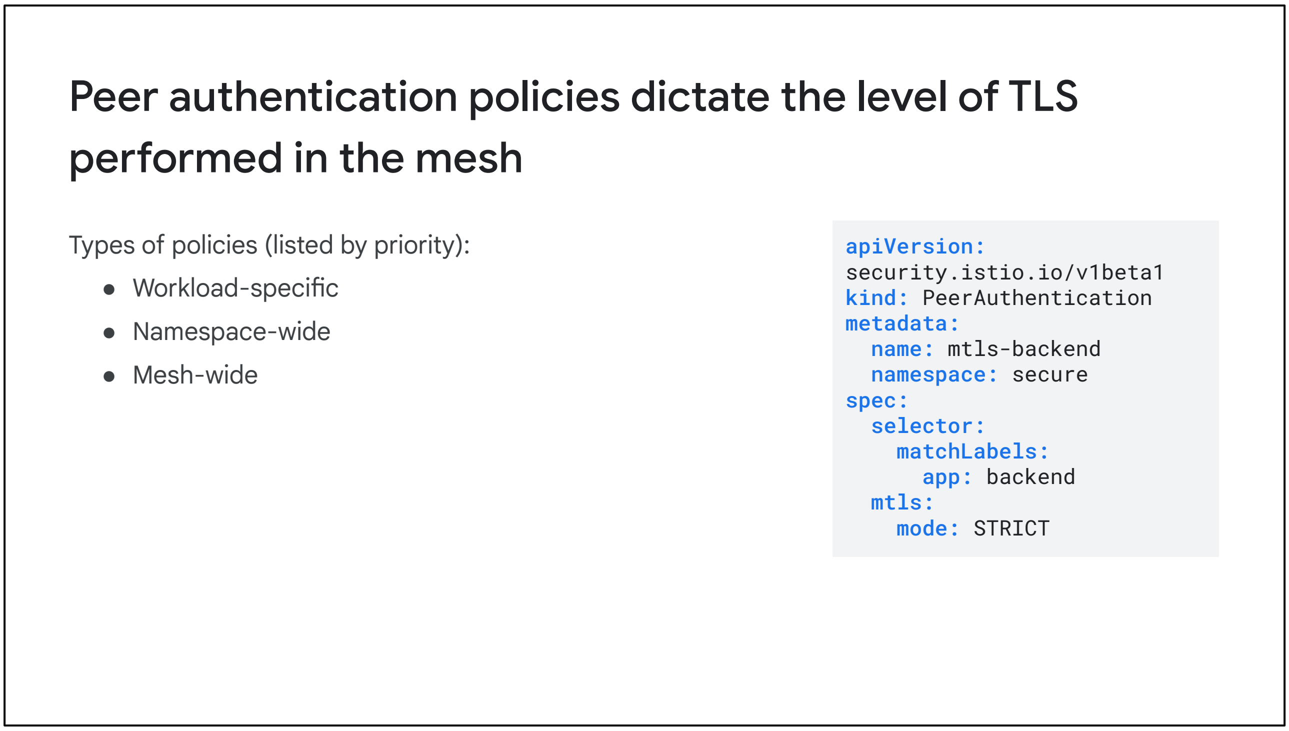
Destination rules dictate the protocol to be used for a specific workload

- ISTIO_MUTUAL: To secure connections to the upstream, use mutual TLS by presenting the default client certificates for authentication.
- MUTUAL: Same as ISTIO_MUTUAL but with custom certificates.
- SIMPLE: Originate a TLS connection to the upstream endpoint.
- DISABLE: Do not set up a TLS connection to the upstream endpoint.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: mtls-backend
spec:
  targets:
  - name: backend.prod.svc.cluster.local
  trafficPolicy:
  - tls:
      mode: ISTIO_MUTUAL
```

Destination rules dictate the protocol to be used for a specific workload and you can specify one of the following options:

- ISTIO_MUTUAL: offers a secure connections to the upstream using mutual TLS by presenting the default client certificates for authentication.
- MUTUAL: is the same as ISTIO_MUTUAL but with custom certificates.
- SIMPLE: originates a TLS connection to the upstream endpoint.
- DISABLE: does not set up a TLS connection to the upstream endpoint.



Peer authentication policies dictate the level of TLS performed in the mesh

Types of policies (listed by priority):

- Workload-specific
- Namespace-wide
- Mesh-wide

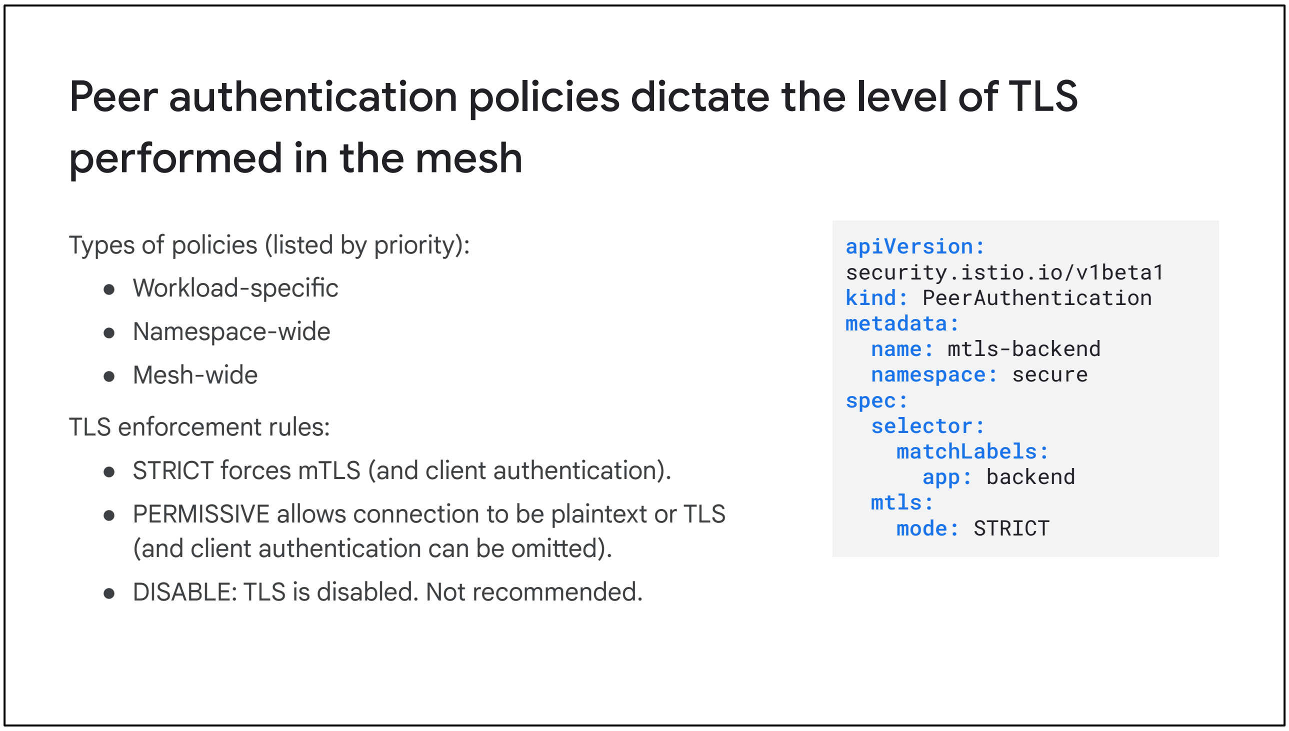
```
apiVersion:
security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: mtls-backend
  namespace: secure
spec:
  selector:
    matchLabels:
      app: backend
  mtls:
    mode: STRICT
```

Peer authentication policies dictate the level of TLS performed in the mesh:

- Workload-specific policies are specified with a namespace and selector.
- Namespace-wide have a namespace, but no selector.
- And mesh-wide don’t have a namespace, and can have a selector.

If there are conflicts across policies, the workload-specific will take priority, next the namespace-specific, and finally the mesh-wide.

If there are multiple policies in one of the levels, let’s say there are two mesh-wide policies, the oldest one will take effect.



Peer authentication policies dictate the level of TLS performed in the mesh

Types of policies (listed by priority):

- Workload-specific
- Namespace-wide
- Mesh-wide

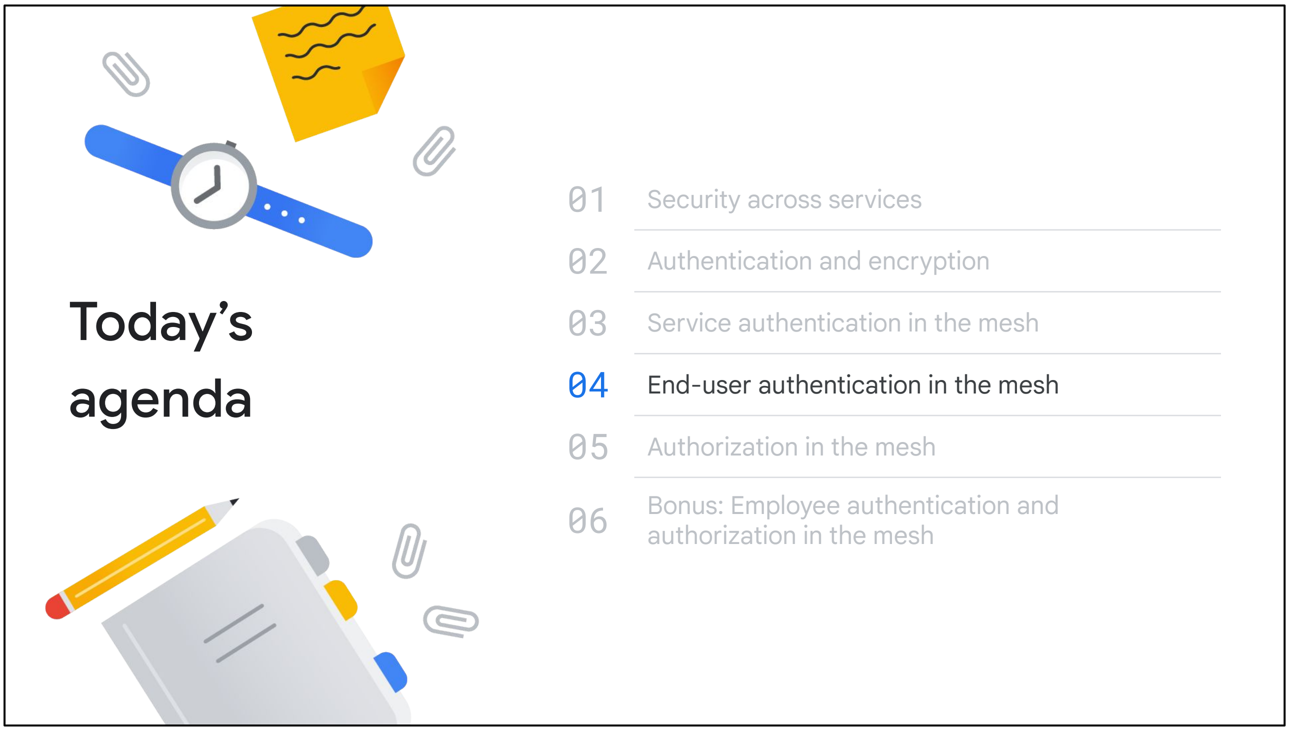
TLS enforcement rules:

- STRICT forces mTLS (and client authentication).
- PERMISSIVE allows connection to be plaintext or TLS (and client authentication can be omitted).
- DISABLE: TLS is disabled. Not recommended.

```yaml
apiVersion:
security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: mtls-backend
  namespace: secure
spec:
  selector:
    matchLabels:
      app: backend
  mtls:
    mode: STRICT
```

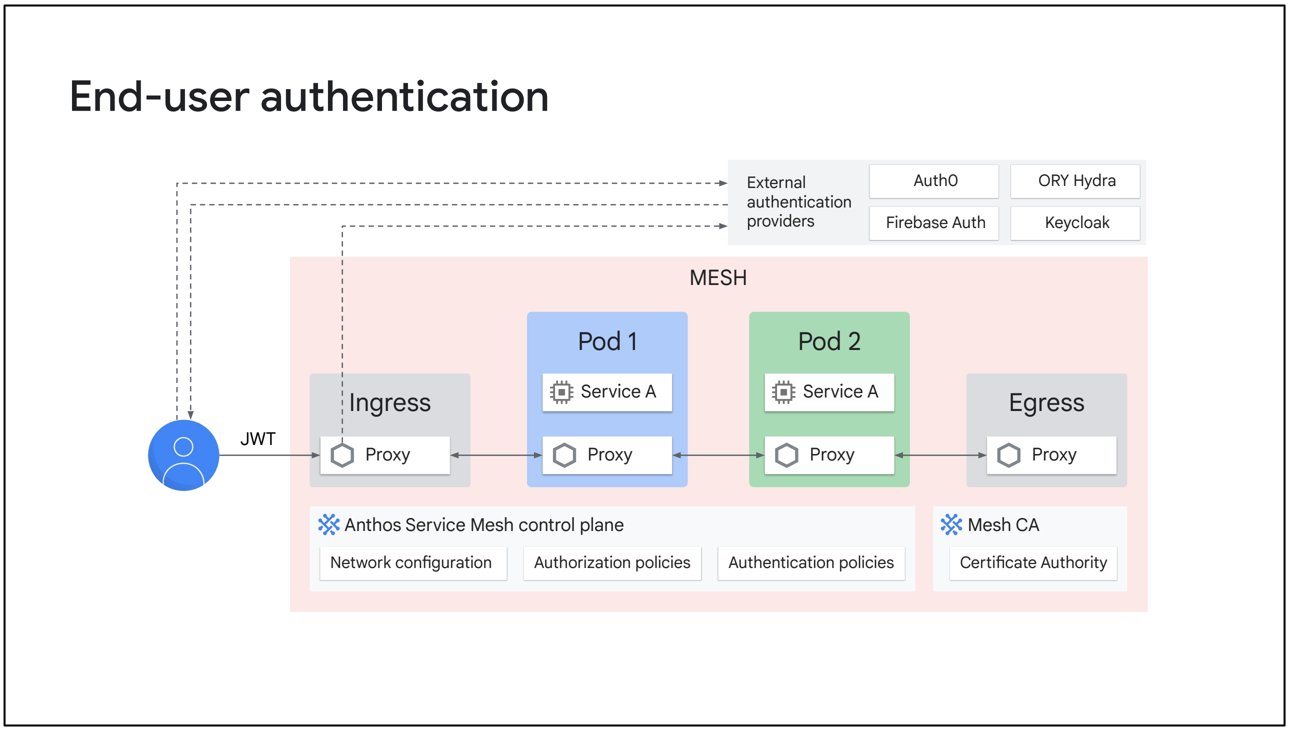
You can also specify the level of TLS enforcement:

- STRICT enforces mTLS and, thus, client authentication. If a workload cannot perform mTLS, the communication won't be performed.
- PERMISSIVE: defaults to the maximum amount of TLS by all clients, allowing to have plaintext connections. This is used when you are first rolling out a mesh, to allow teams to enable TLS on their workloads.
- DISABLE: TLS is disabled. This option is not recommended.



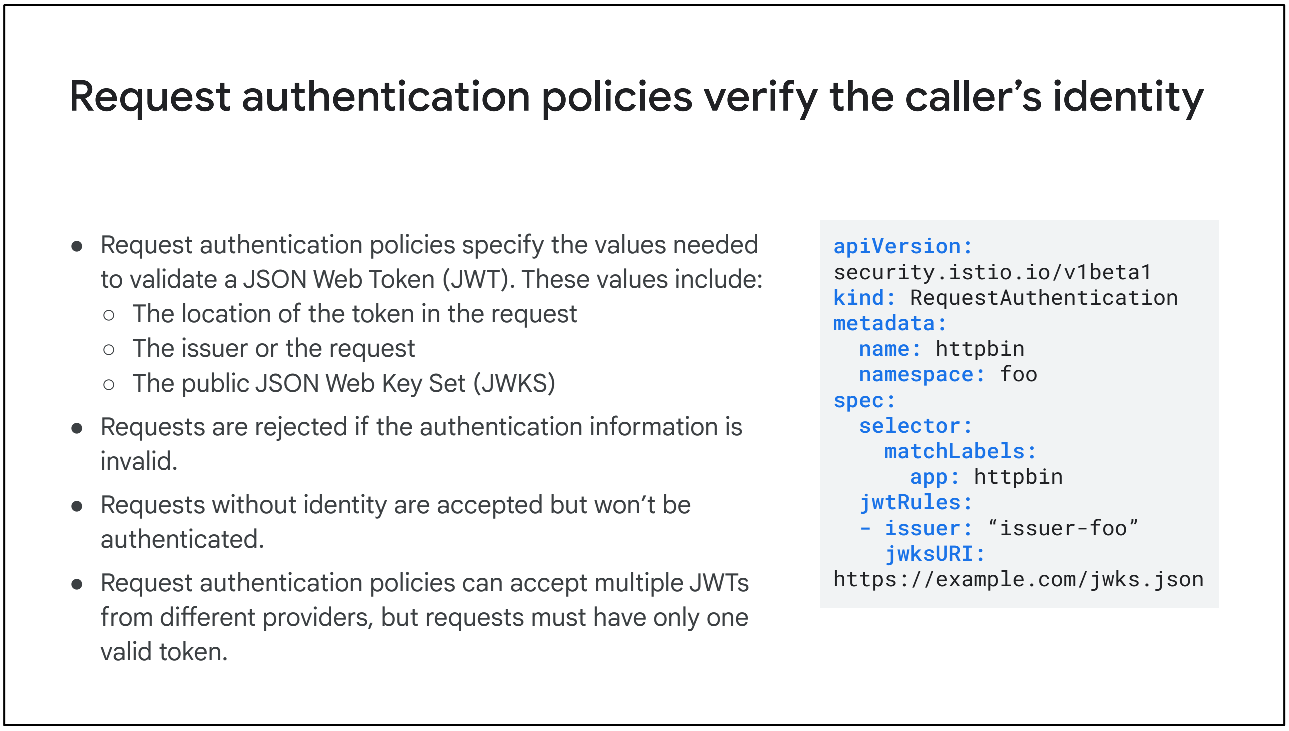
Today's agenda

In addition to service authentication, we also have request authentication, which is used to authenticate end users.



End-user authentication

Request authentication works using JSON Web Tokens, or JWT. The end user performs authentication using a custom authentication provider or any OpenID Connect providers, for example, Auth0, ORY Hydra, Firebase Auth, or Keycloak. This provider returns the JWT token which is included in subsequent requests to services in the mesh. The gateway’s sidecar proxy checks and validates the token before authenticating the request.



Request authentication policies verify the caller’s identity

- Request authentication policies specify the values needed to validate a JSON Web Token (JWT). These values include:
  - The location of the token in the request
  - The issuer or the request
  - The public JSON Web Key Set (JWKS)
- Requests are rejected if the authentication information is invalid.
- Requests without identity are accepted but won’t be authenticated.
- Request authentication policies can accept multiple JWTs from different providers, but requests must have only one valid token.

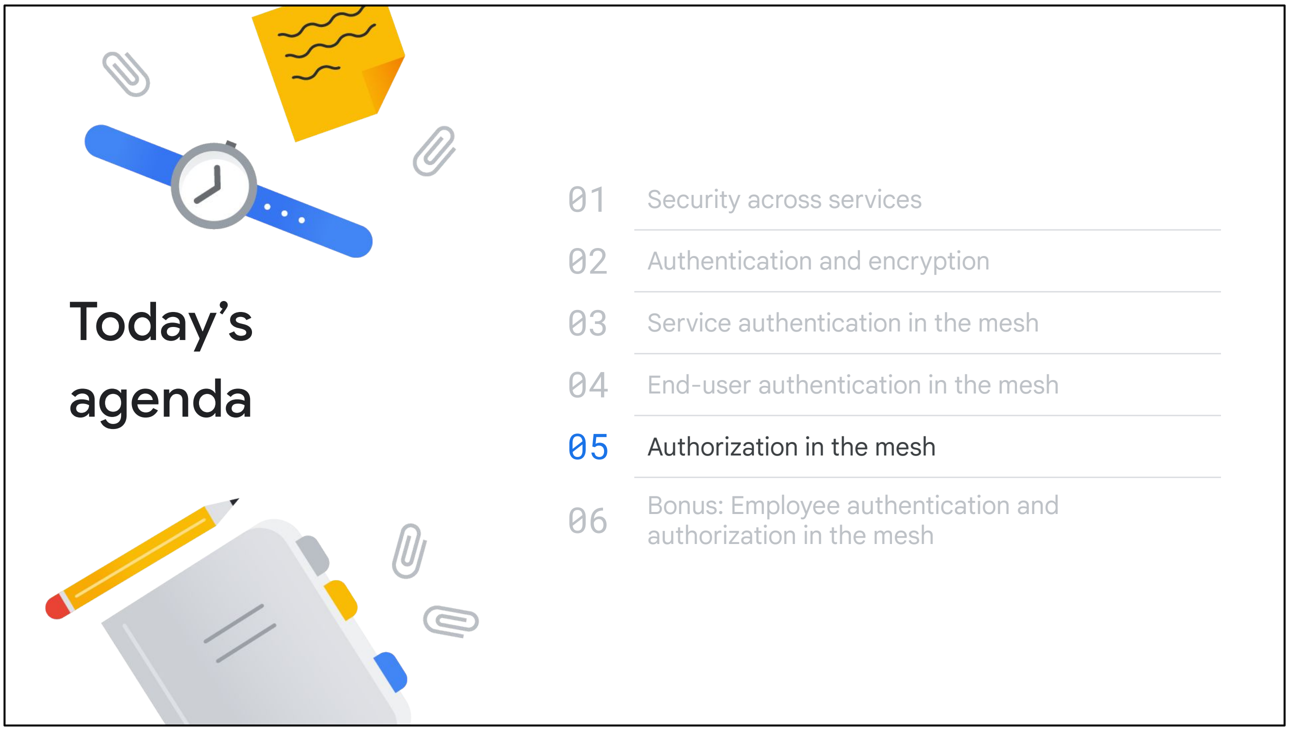
```
apiVersion:
security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  jwtRules:
  - issuer: “issuer-foo”
    jwksURI:
https://example.com/jwks.json
```

Request authentication policies verify the caller’s identity using JSON Web Token, or JWT. The values in these policies include:

- The location of the token in the request.
- The issuer or the request.
- The public JSON Web Key Set, or JWKS.

Requests are rejected if the authentication information is invalid. Requests without identity are accepted but won’t be authenticated.

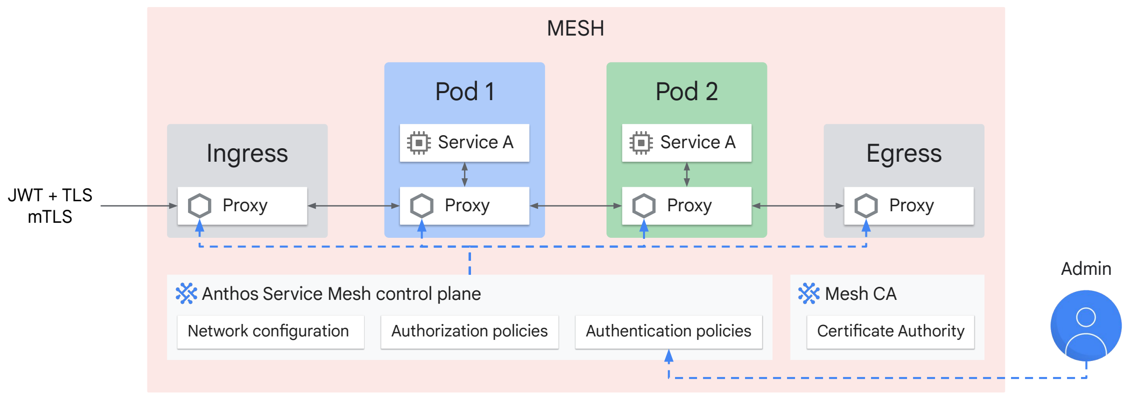
Request authentication policies can accept multiple JWTs from different providers, but requests must only have one valid token, otherwise, the output principle of the request is undefined.



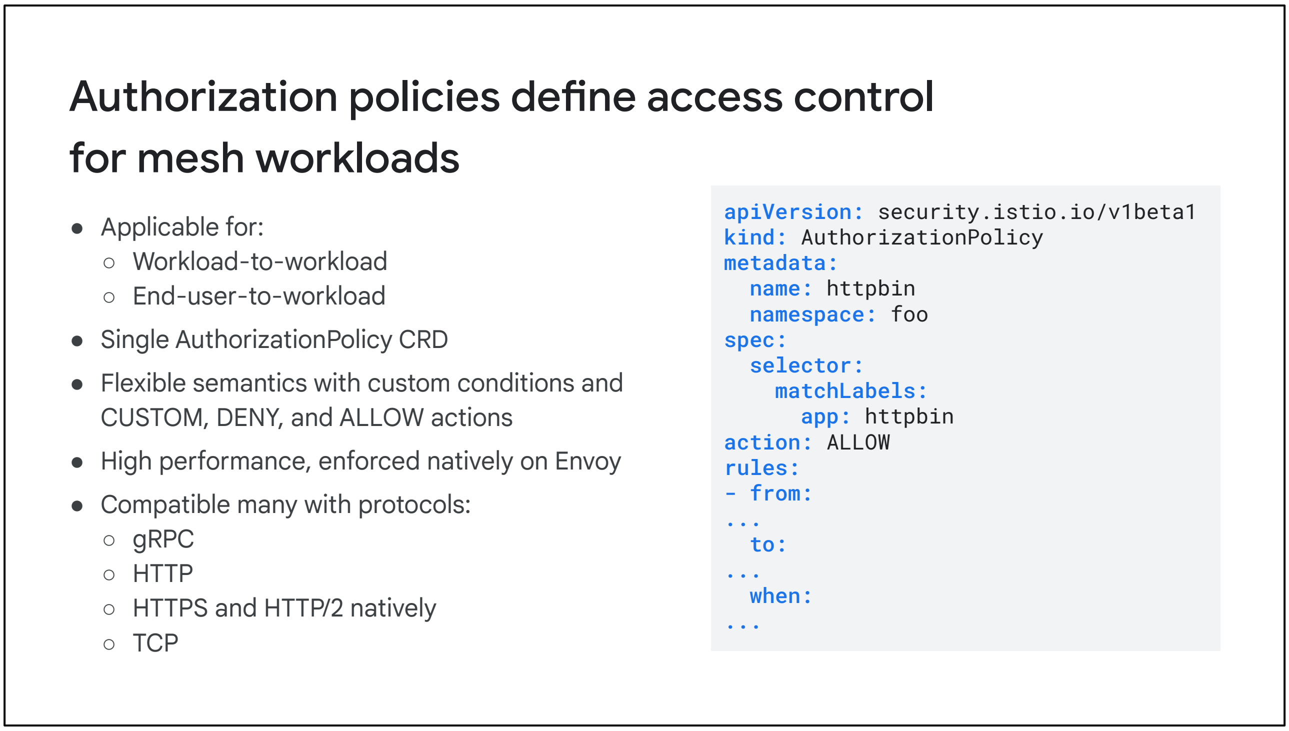
Today's agenda

Once the request has been authenticated, we can use this information to authorize the request.

Sidecar and perimeter proxies work as Policy Enforcement Points (PEPs) to authorize requests



Sidecar and perimeter proxies work as Policy Enforcement Points, or PEPs, to authorize requests. When a request comes to the proxy, the authorization engine evaluates the request context against the current authorization policies, and returns the authorization result, either ALLOW or DENY. Operators specify the mesh authorization policies using .yaml files.

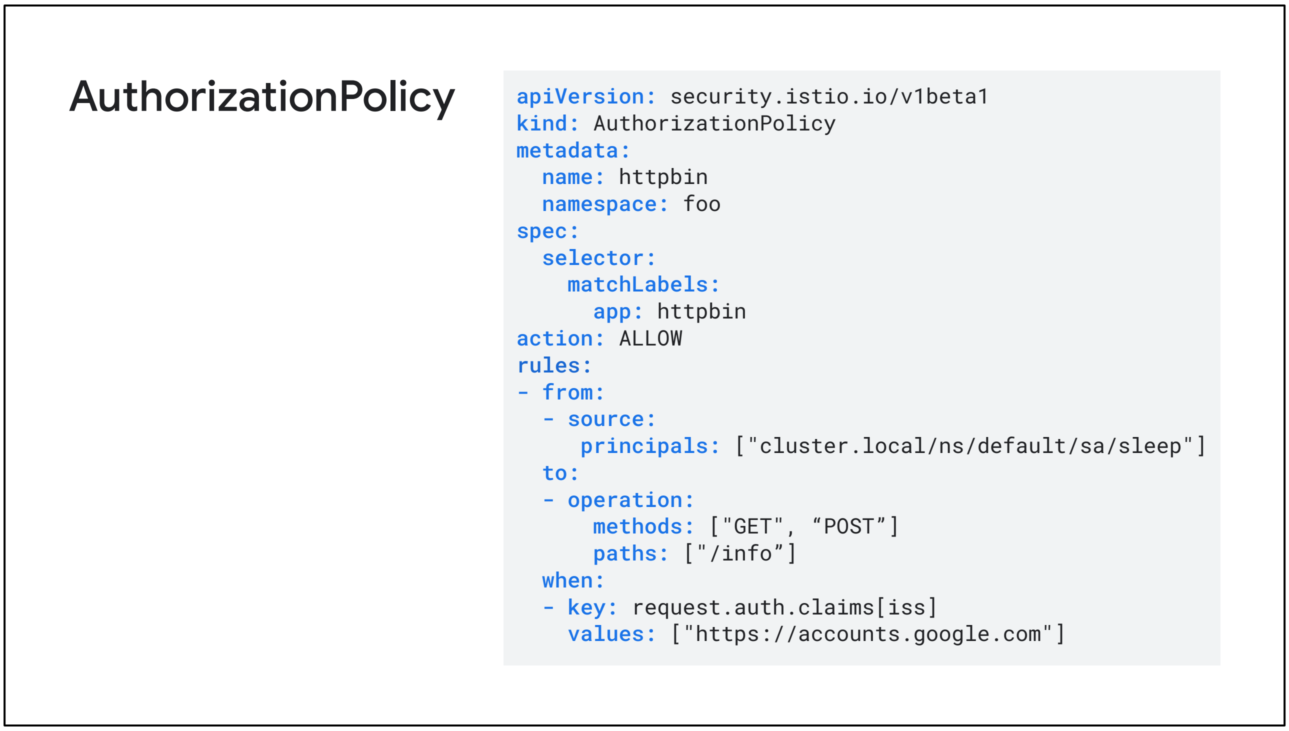


Authorization policies define access control for mesh workloads

- Applicable for:
  - Workload-to-workload
  - End-user-to-workload
- Single AuthorizationPolicy CRD
- Flexible semantics with custom conditions and CUSTOM, DENY, and ALLOW actions
- High performance, enforced natively on Envoy
- Compatible many with protocols:
  - gRPC
  - HTTP
  - HTTPS and HTTP/2 natively
  - TCP

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
action: ALLOW
rules:
- from:
...
  to:
...
  when:
...
```

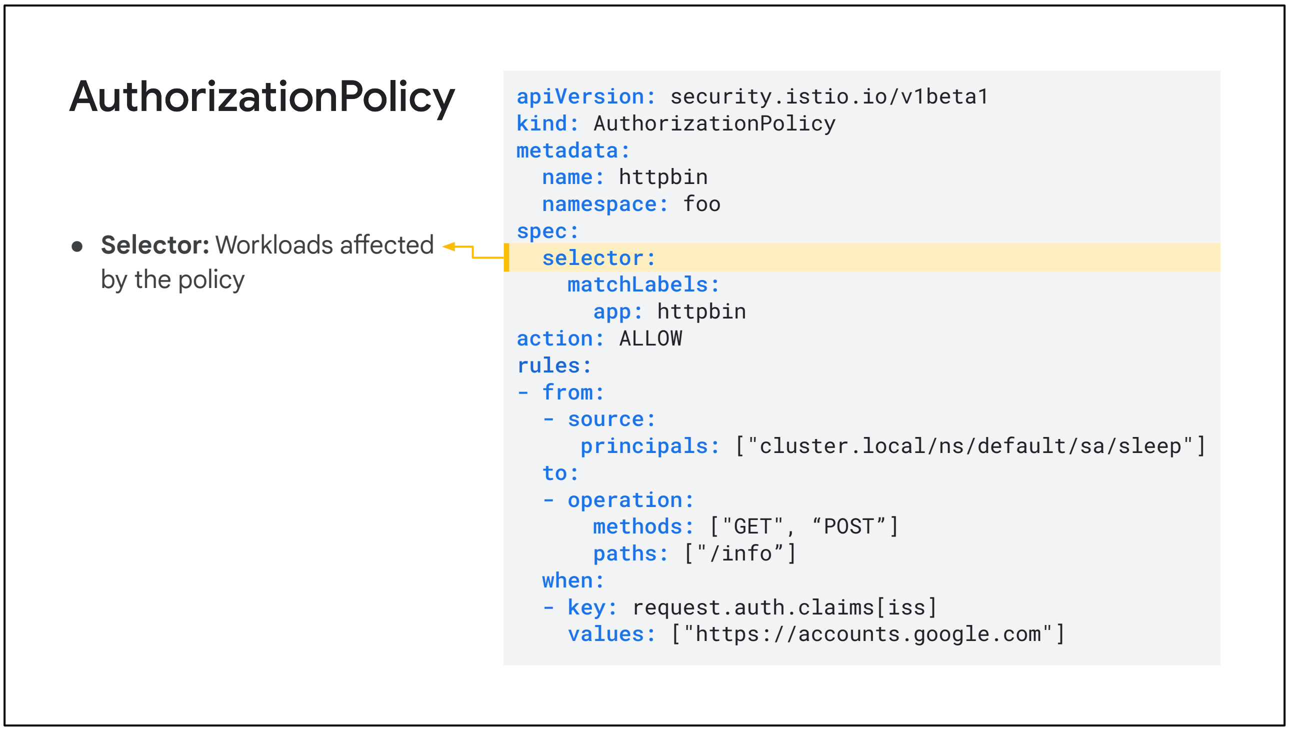
Authorization policies define access control for mesh workloads and are applicable both workload-to-workload and end-user-to-workload. They are defined using a single AuthorizationPolicy CRD, which provides flexible semantics with custom conditions and CUSTOM, DENY, and ALLOW actions. Those conditions are enforced natively on Envoy, achieving high performance. Authorization policies support a range of protocols such as gRPC, HTTP, HTTPS, and HTTP/2 natively, and any TCP protocol.



AuthorizationPolicy

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
action: ALLOW
rules:
- from:
  - source:
     principals: ["cluster.local/ns/default/sa/sleep"]
  to:
  - operation:
      methods: ["GET", “POST”]
      paths: ["/info”]
  when:
  - key: request.auth.claims[iss]
    values: ["https://accounts.google.com"]
```

Let’s look at the structure of this CRD. An Authorization Policy includes a selector, an action, and a list of rules:

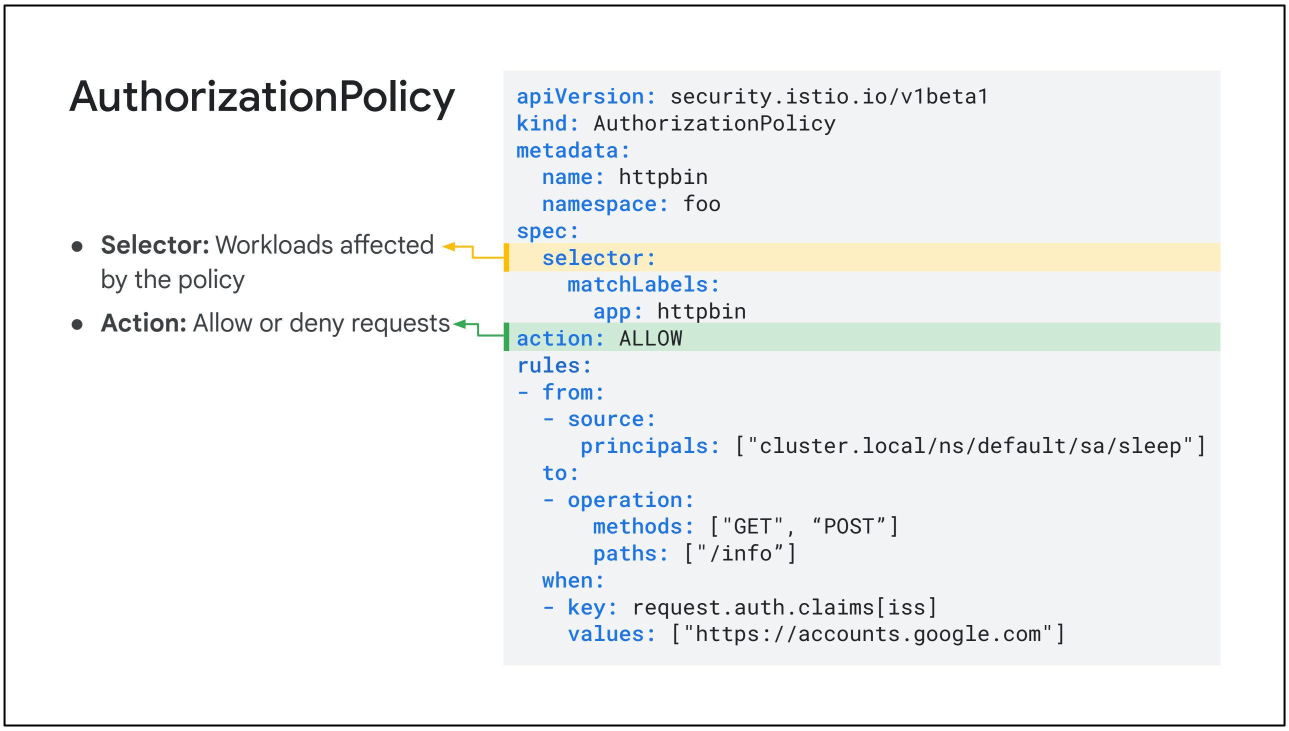


AuthorizationPolicy

- **Selector:** Workloads affected by the policy

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
action: ALLOW
rules:
- from:
  - source:
     principals: ["cluster.local/ns/default/sa/sleep"]
  to:
  - operation:
      methods: ["GET", “POST”]
      paths: ["/info”]
  when:
  - key: request.auth.claims[iss]
    values: ["https://accounts.google.com"]
```

- The selector field specifies the workloads affected by the policy.

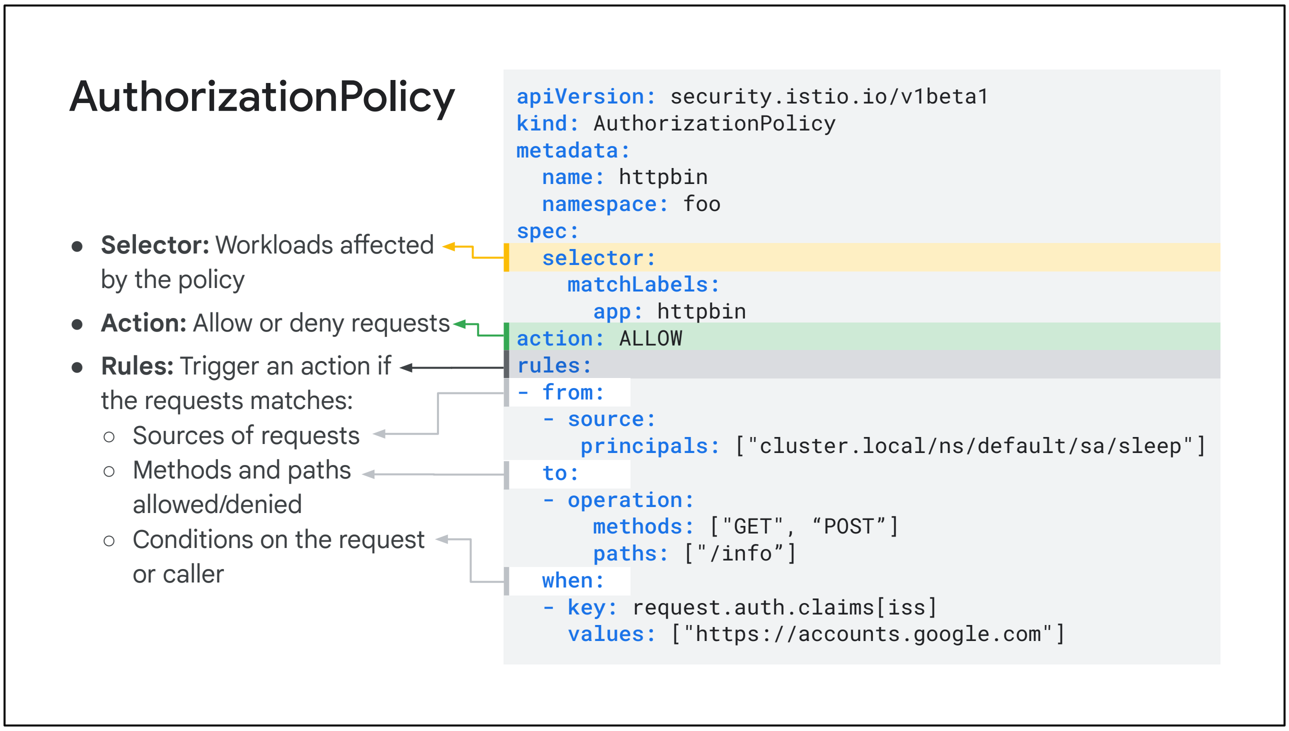


AuthorizationPolicy

- **Selector:** Workloads affected by the policy
- **Action:** Allow or deny requests

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
action: ALLOW
rules:
- from:
  - source:
     principals: ["cluster.local/ns/default/sa/sleep"]
  to:
  - operation:
      methods: ["GET", “POST”]
      paths: ["/info”]
  when:
  - key: request.auth.claims[iss]
    values: ["https://accounts.google.com"]
```

- The action field specifies whether to allow or deny requests.

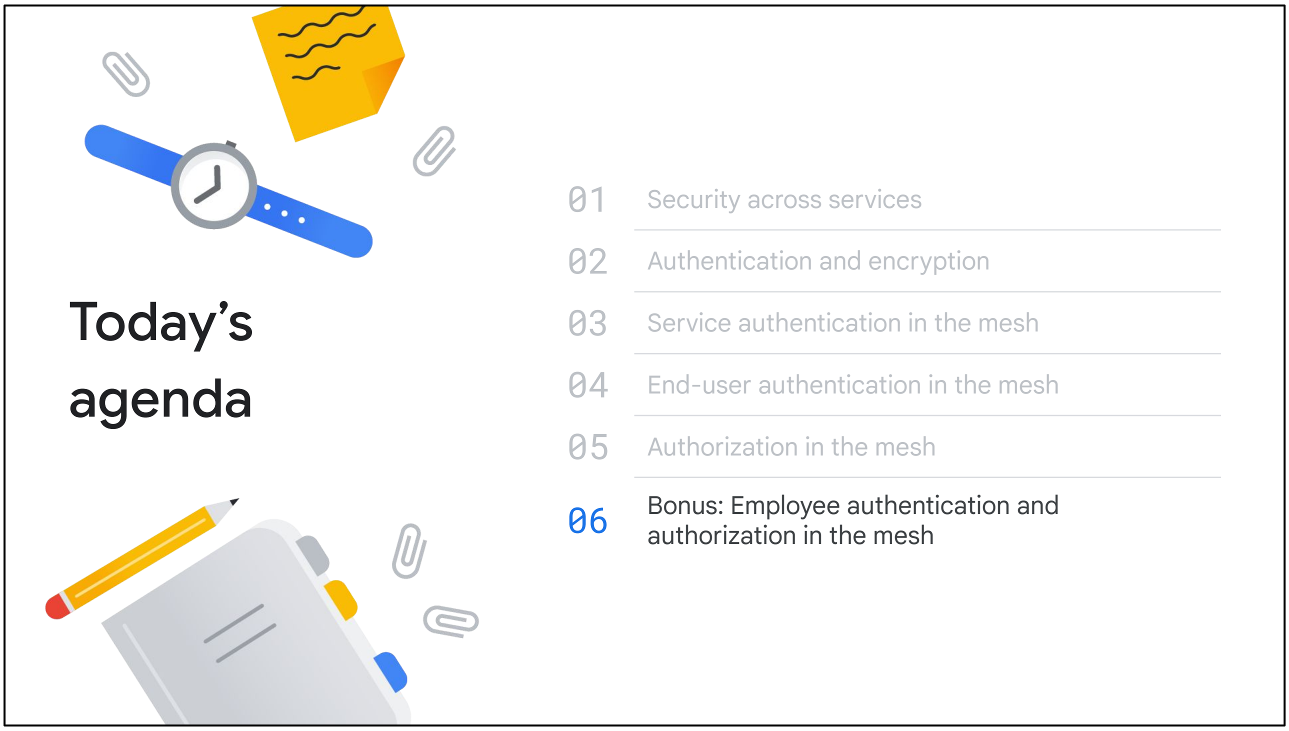


AuthorizationPolicy

- **Selector:** Workloads affected by the policy
- **Action:** Allow or deny requests
- **Rules:** Trigger an action if the requests matches:
  - Sources of requests
  - Methods and paths allowed/denied
  - Conditions on the request or caller

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
action: ALLOW
rules:
- from:
  - source:
     principals: ["cluster.local/ns/default/sa/sleep"]
  to:
  - operation:
      methods: ["GET", “POST”]
      paths: ["/info”]
  when:
  - key: request.auth.claims[iss]
    values: ["https://accounts.google.com"]
```

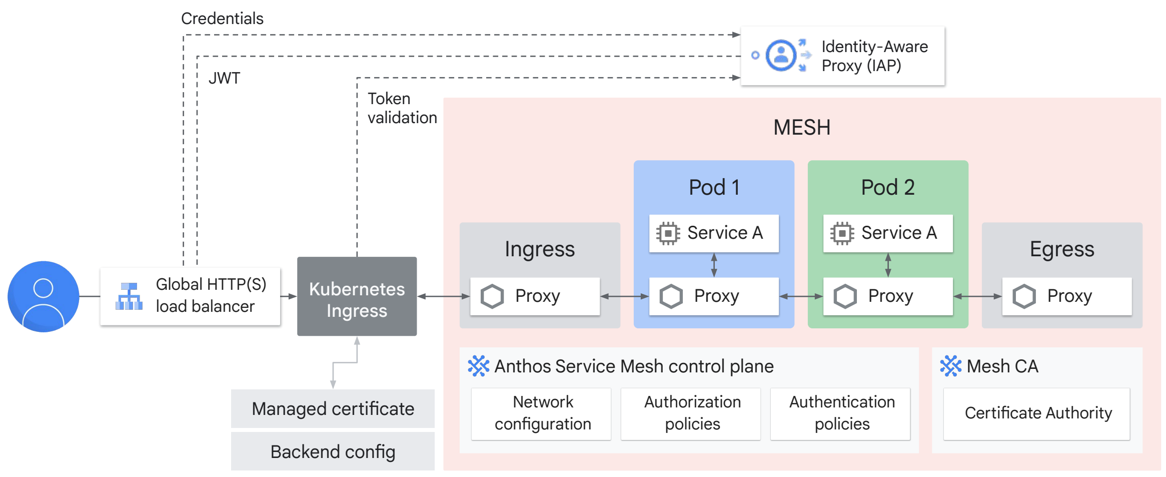
- The rules specify when to trigger an action.
  - The `from` field in the rules specifies sources of requests.
  - The `to` field in the rules specifies the methods and paths allowed/denied. Notice that we apply this rules to layer 7 networking, while in Kubernetes, we can only apply network policies at layer 4.
  - The `when` field specifies the conditions that would cause a request to match the rule.



Today's agenda

Employee authentication is a specialized use case of end-user authentication.

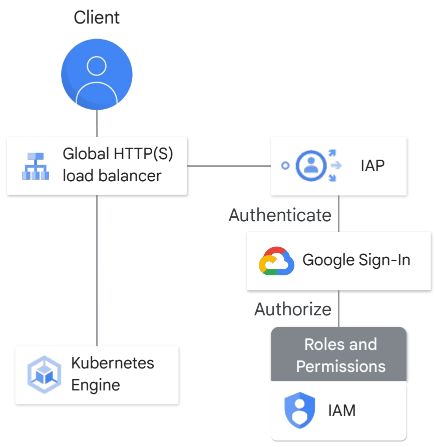
Employee authentication and authorization with IAP



In Google Cloud, you can use Identity Aware Proxy, or IAP, as the authentication and authorization provider for employees of your organization. That way, you allow access to internal applications without relying on network-level firewalls or the need of a VPN. With IAP, you can establish a central authorization layer for applications accessed by HTTPS with group-based controls, so that departments are granted individual access to different applications.

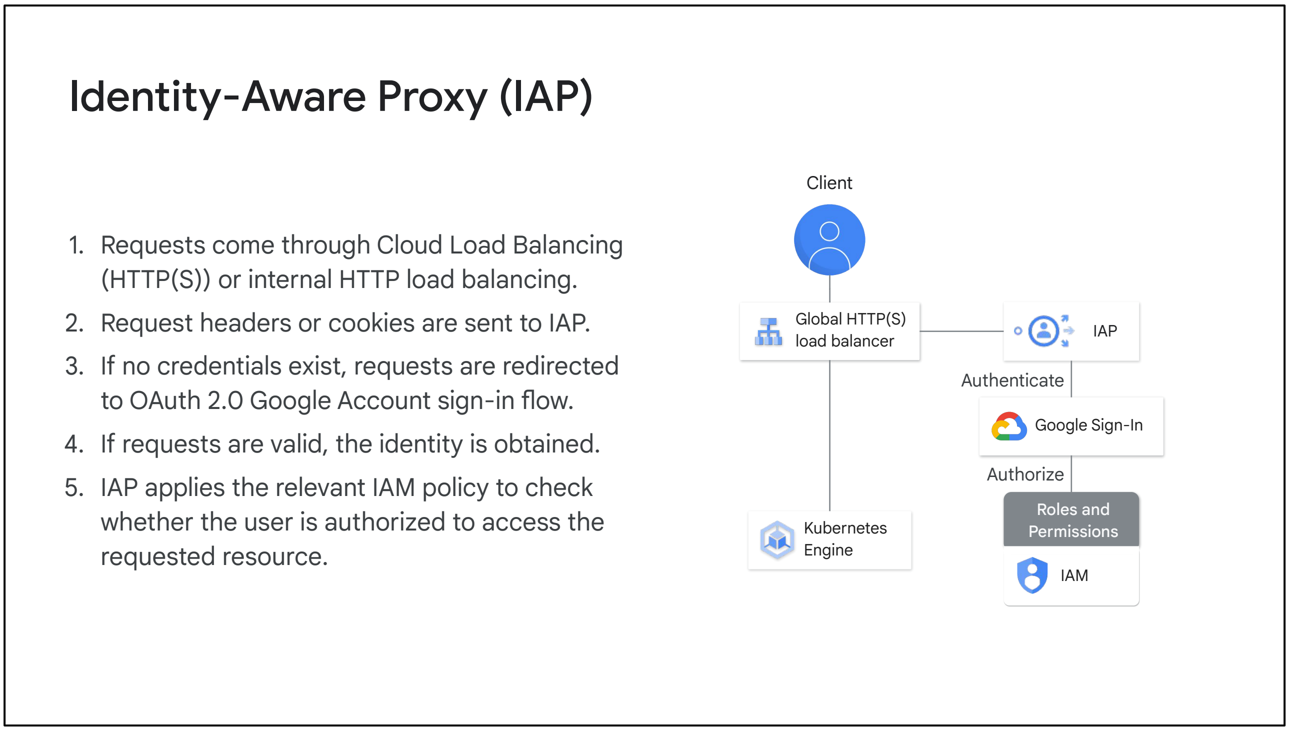
Identity-Aware Proxy (IAP)

1. Requests come through Cloud Load Balancing (HTTP(S)) or internal HTTP load balancing.
2. Request headers or cookies are sent to IAP.
3. If no credentials exist, requests are redirected to OAuth 2.0 Google Account sign-in flow.



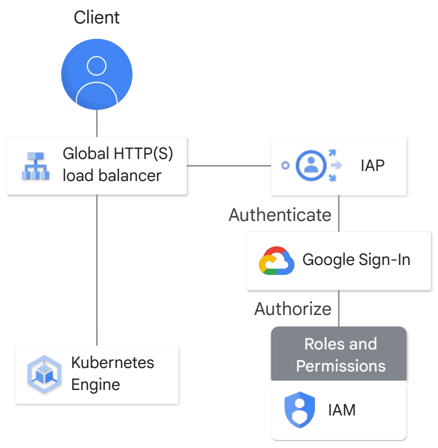
Requests come through Cloud Load Balancing (HTTPS), or internal HTTP load balancing. The serving infrastructure code for these products checks if IAP is enabled for the app or backend service. If IAP is enabled, information about the protected resource is sent to the IAP authentication server. This includes information like the Google Cloud project number, the request URL, and any IAP credentials in the request headers or cookies.

Next, IAP checks the user's browser credentials. If none exist, the user is redirected to an OAuth 2.0 Google Account sign-in flow that stores a token in a browser cookie for future sign-ins. If you need to create Google Accounts for your existing users, you can use Google Cloud Directory Sync to synchronize with your Active Directory or LDAP server.



Identity-Aware Proxy (IAP)

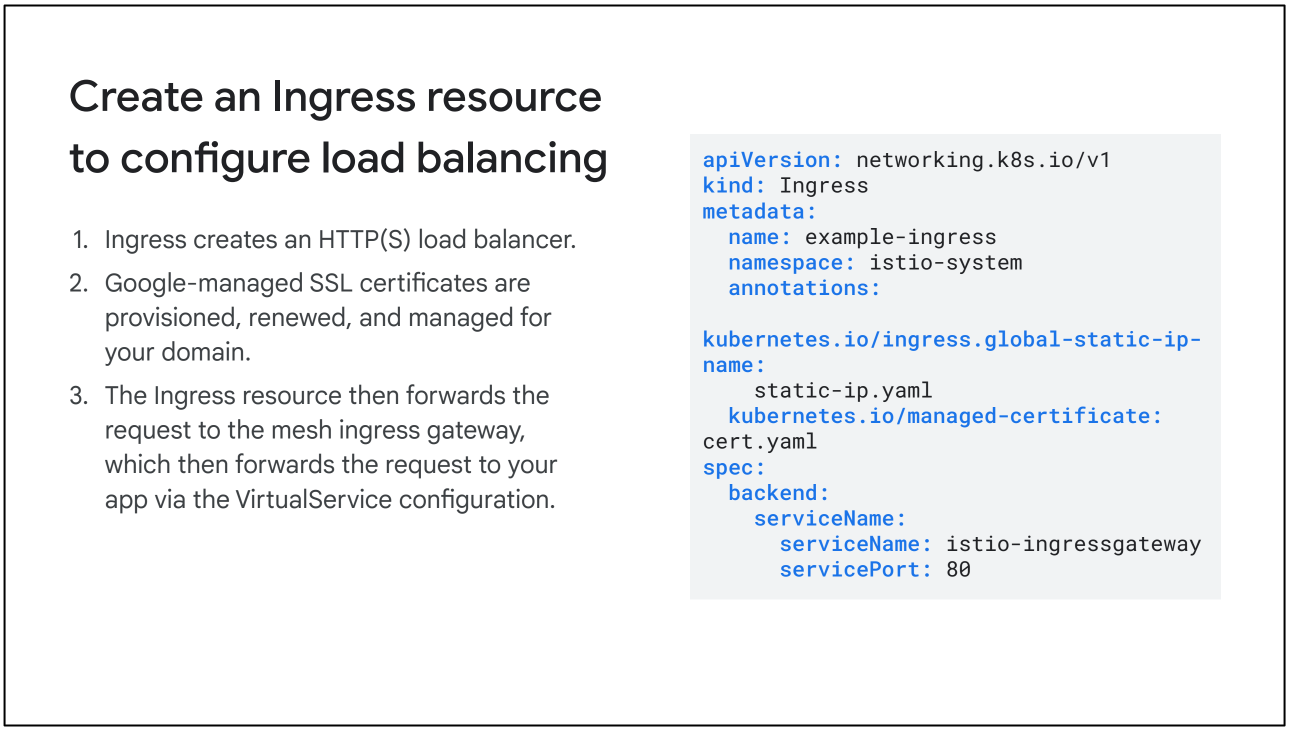
1. Requests come through Cloud Load Balancing (HTTP(S)) or internal HTTP load balancing.
2. Request headers or cookies are sent to IAP.
3. If no credentials exist, requests are redirected to OAuth 2.0 Google Account sign-in flow.
4. If requests are valid, the identity is obtained.
5. IAP applies the relevant IAM policy to check whether the user is authorized to access the requested resource.



If the request credentials are valid, the authentication server uses those credentials to get the user's identity (email address and user ID).

After authentication, IAP applies the relevant IAM policy to check if the user is authorized to access the requested resource. If the user has the **IAP-secured Web App User** role on the Cloud Console project where the resource exists, they're authorized to access the application. To manage the **IAP-secured Web App User** role list, use the IAP panel on the Cloud Console.

When you turn on IAP for a resource, it automatically creates an OAuth 2.0 client ID and secret. If you delete the automatically generated OAuth 2.0 credentials, IAP won't function correctly. You can view and manage OAuth 2.0 credentials in the Cloud Console APIs & services.



Create an Ingress resource to configure load balancing

1. Ingress creates an HTTP(S) load balancer.
2. Google-managed SSL certificates are provisioned, renewed, and managed for your domain.
3. The Ingress resource then forwards the request to the mesh ingress gateway, which then forwards the request to your app via the VirtualService configuration.

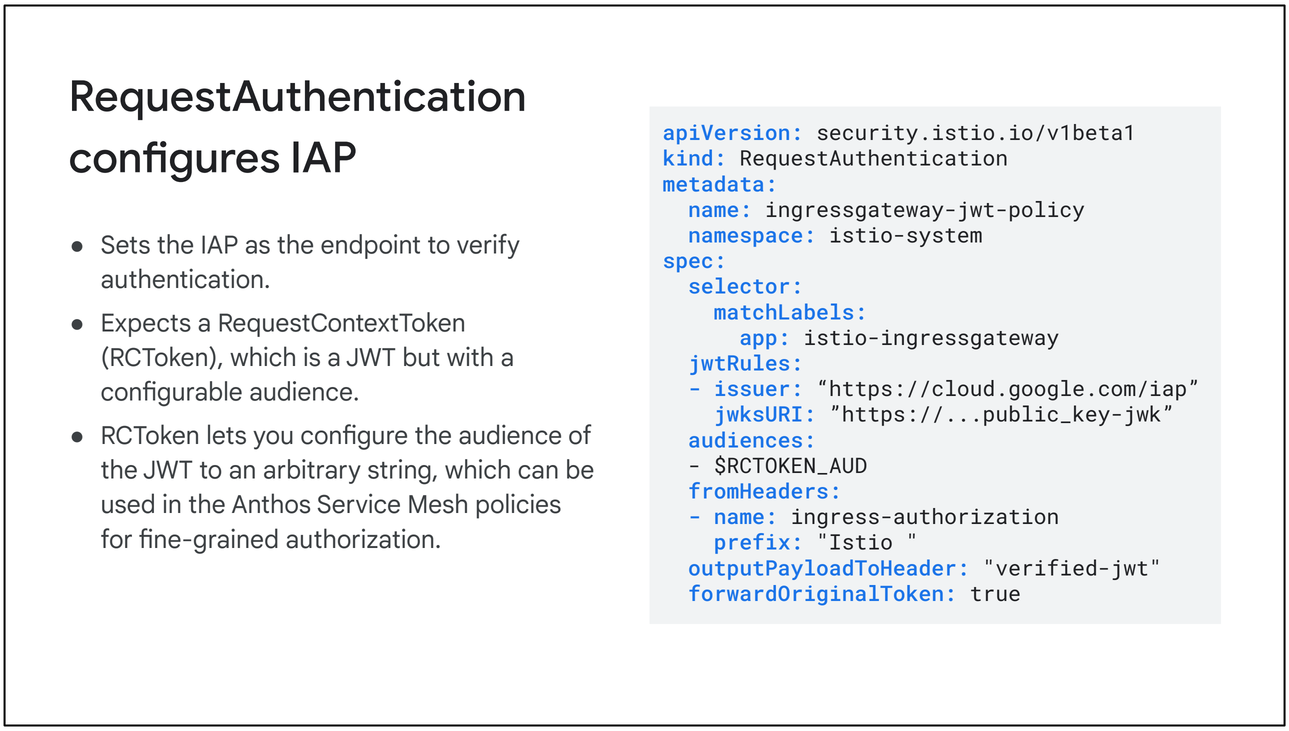
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  namespace: istio-system
  annotations:

kubernetes.io/ingress.global-static-ip-
name:
    static-ip.yaml
  kubernetes.io/managed-certificate:
cert.yaml
spec:
  backend:
    serviceName:
      serviceName: istio-ingressgateway
      servicePort: 80
```

Create an ingress resource to configure an HTTPS Load Balancer.

Google-managed SSL certificates are provisioned, renewed, and managed for your domain.

The Ingress resource then forwards the request to the mesh ingress gateway, which then forwards the request to your app via the VirtualService configuration.



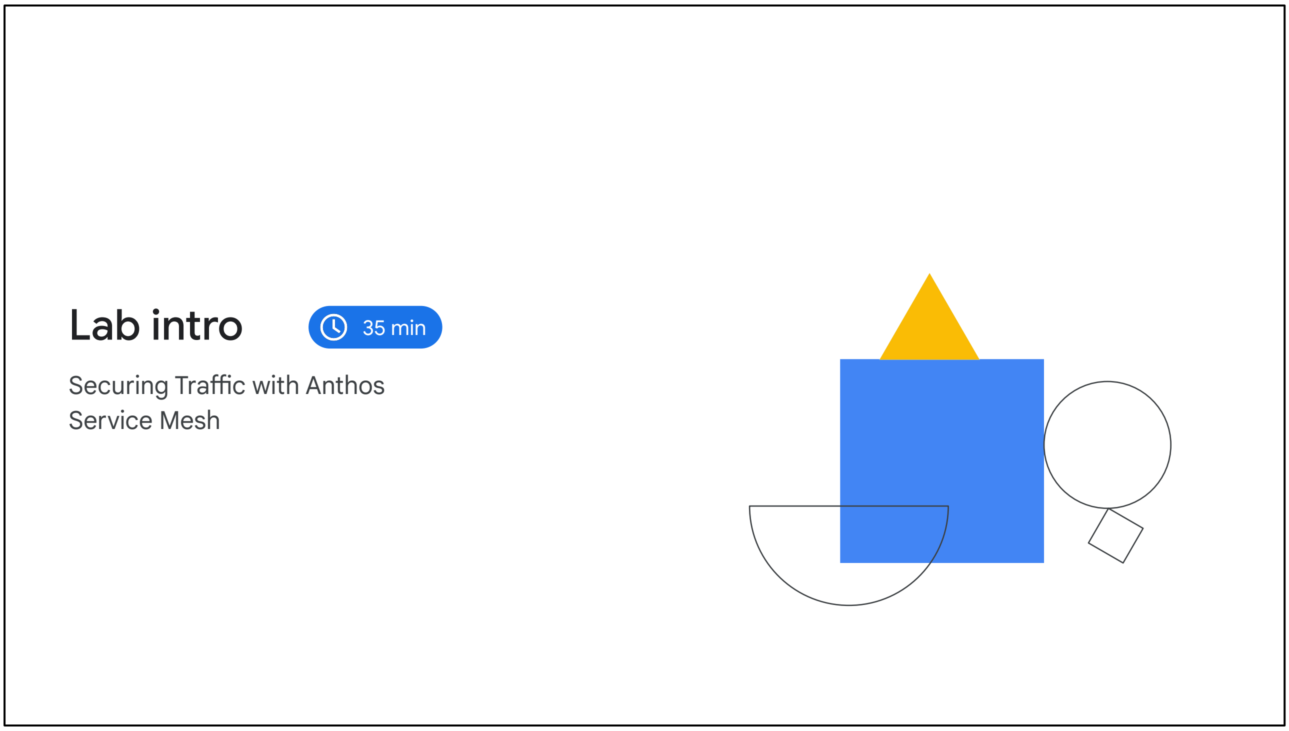
RequestAuthentication configures IAP

- Sets the IAP as the endpoint to verify authentication.
- Expects a RequestContextToken (RCToken), which is a JWT but with a configurable audience.
- RCToken lets you configure the audience of the JWT to an arbitrary string, which can be used in the Anthos Service Mesh policies for fine-grained authorization.

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: ingressgateway-jwt-policy
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway
  jwtRules:
  - issuer: “https://cloud.google.com/iap”
    jwksURI: ”https://...public_key-jwk”
  audiences:
  - $RCTOKEN_AUD
  fromHeaders:
  - name: ingress-authorization
    prefix: "Istio "
  outputPayloadToHeader: "verified-jwt"
  forwardOriginalToken: true
```

RequestAuthentication configures IAP as the request authentication method.

By default, IAP generates a JSON Web Token, or JWT, that is scoped to the OAuth client. For Anthos Service Mesh, you can configure IAP to generate a RequestContextToken, or RCToken, which is a JWT but with a configurable audience. RCToken lets you configure the audience of the JWT to an arbitrary string, which can be used in the Anthos Service Mesh policies for fine-grained authorization.



Lab intro

35 min

Securing Traffic with Anthos Service Mesh

Next, you will do a lab exercise to get some practice with the Anthos Service Mesh.

In the lab, you perform the following tasks:

- Enforce STRICT mTLS mode across the service mesh
- Enforce STRICT mTLS mode on a single namespace
- Explore the security configurations in the Anthos Service Mesh Dashboard
- Add authorization policies to enforce access based on a JSON Web Token (JWT)
- Add authorization policies for HTTP traffic in an Istio mesh