Applications deployed in application servers are provided a security framework with authentication, authorization, credential mappers, auditing, and other security plug-ins. Many of the large, monolithic applications, such as HCM and ERP also contain security components embedded within these applications. These security components provide authentication, as well as, role-based access control policies.

A microservice architecture also has these same security requirements. Since a microservice application is usually made up of many distributed services the communication between services usually requires encryption of the data being transmitted. The more complex the system, the more possible ways the system can be attacked maliciously. Therefore, the application architecture needs to include security practices that protect against such attacks.

What follows is a discussion of authentication, authorization, and mutual TLS encryption in a microservices architecture. The Istio service mesh is deployed on a Kubernetes cluster. But, before getting too far into the security features with the Istio service mesh, let’s get some understanding of the high-level architecture of Istio and to understand the basics of authentication and authorization in the service mesh.

# High-level architecture

There are four key components as part of the Istio architecture. These components are the Citadel, Envoy proxy, Pilot, and the Mixer.

* **Citadel** for key and certificate management
* **Envoy proxy** to implement secure communication between clients and servers
* **Pilot** to distribute authentication policies and secure naming information to the proxies
* **Mixer** to manage authorization and auditing

# Authentication

The Kubernetes API server can be configured with one or more authentication plugins and authorization plugins. When a request is received by the API server, it goes through the list of authentication plugins. The first plugin that can extract that information from the request returns the username, user ID, and the group(s) the client belongs to back to the API server core. The API server stops invoking the remaining authentication plugins and continues onto the authorization phase.

The authentication plugins obtain the identity of the client from the client certificate, from an authentication token passed in an HTTP header, basic HTTP authentication, or some other method.

The username and group(s) are returned from the authentication plugin. These attributes are then used to verify whether or not the user is authorized to perform an action.

It’s important to understand that Kubernetes distinguishes two kinds of clients; humans, which are considered users, and Pods, which are more specifically the applications that are running in the pod.

Both types of clients are authenticated with the deployed authentication plugins. Users are managed by an external system; therefore, no resource in Kubernetes represents user accounts. You cannot create, update, or delete users through the API server. Pods on the other hand use service accounts. The service accounts are created and stored in the cluster as ServiceAccount resources.

The API server requires clients to authenticate themselves before they are allowed to perform operations on the server. A service account is a namespaced resource that you can use if your application needs to communicate with the API server. The creation of a service account triggers the creation of a secret, is attached to and managed by the service account. The secret contains a JSON Web Token (jwt). The jwt token is written to the */var/run/secrets/kuberneties.io/serviceaccount/*token. You can verify the creation of the certificate and tokens by doing the following once the service account has been created and you have created the pod.

kubectl exec -it greetandweather-service-v1-5b67c8c4f7-w4w9m -n greet-ns -- ls -ltra /var/run/secrets/kubernetes.io/serviceaccount/

total 0

lrwxrwxrwx 1 root root 12 Jan 4 00:39 **token** -> ..data/token

lrwxrwxrwx 1 root root 16 Jan 4 00:39 **namespace** -> ..data/namespace

lrwxrwxrwx 1 root root 13 Jan 4 00:39 **ca.crt** -> ..data/ca.crt

lrwxrwxrwx 1 root root 31 Jan 4 00:39 **..data** -> **..2019\_01\_04\_00\_39\_53.348735841**

drwxr-xr-x 2 root root 100 Jan 4 00:39 **..2019\_01\_04\_00\_39\_53.348735841**

drwxrwxrwt 3 root root 140 Jan 4 00:39 **.**

drwxr-xr-x 3 root root 28 Jan 4 00:39 **..**

The token contains the jwt token. The namespace file contains the name of the namespace for the pod and the ca.crt contains the automatically generated certificate.

So how does a pod authenticate to the API server? Pods authenticate by sending the contents of the token which is mounted into each container’s filesystem through a secret volume.

Let’s take a look at the token and the filesystem for a Pod.

Obtain a list of the service accounts in the namespace “greet-ns”:

szern-mac:~ szern$ kubectl get sa -n greet-ns

NAME SECRETS AGE

default 1 5d

greet-and-weather-admin 1 5d

We now have the service accounts for the requested namespace. Now let’s look deeper into the namespace. You will see by creating the service account that Kubernetes automatically creates the secrets.

szern-mac:~ szern$ kubectl describe sa greet-and-weather-admin -n greet-ns

Name: greet-and-weather-admin

Namespace: greet-ns

Labels: <none>

Annotations: kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1","kind":"ServiceAccount","metadata":{"annotations":{},"name":"greet-and-weather-admin","namespace":"greet-ns"}}

Image pull secrets: <none>

Mountable secrets: greet-and-weather-admin-token-fggkl

Tokens: greet-and-weather-admin-token-fggkl

Events: <none>

You can now describe the secret, which is a jwt token.

szern-mac:~ szern$ kubectl describe secret greet-and-weather-admin-token-fggkl -n greet-ns

Name: greet-and-weather-admin-token-fggkl

Namespace: greet-ns

Labels: <none>

Annotations: kubernetes.io/service-account.name=greet-and-weather-admin

kubernetes.io/service-account.uid=3f72e0f5-0fb9-11e9-8f2f-0a580aedbb7f

Type: kubernetes.io/service-account-token

Data

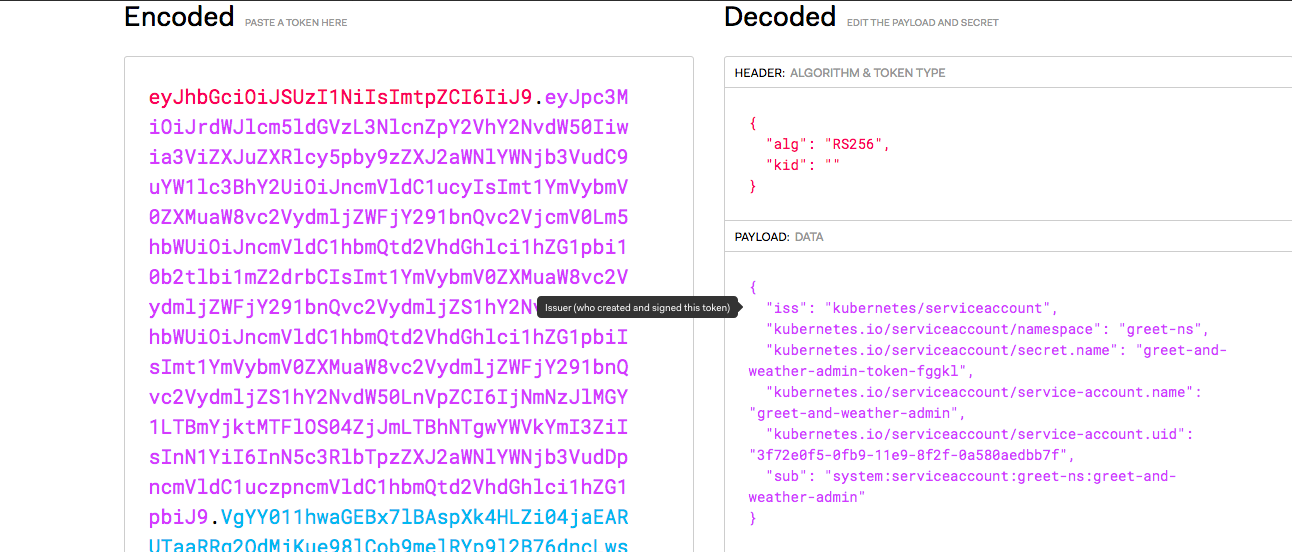
====

ca.crt: 1289 bytes

namespace: 8 bytes

token: eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9..VgYY011hwaGEBx7lBAspXk4HLZi04jaEARUTaaRRq2QdMj-Kue98lCob9me-lRYp9l2B76dncLwsi\_YLEExfu\_ejIhQAHcgxAxRLyrsAwA\_Q9jKVBLF0EYdjZzyD1W\_QT6rhnw3jfMdmj6OFKA0BZiIIVaWpd3yvrKFiog6GRx28KSrDnvibFm9uTX9UC4332T4FAIrtwCjti17CbvjBC-5TihLq0yY\_D9aZ070frKVMXtdKtDk3gteRz6\_DP0aF7Eqap4QNtRf673n1DkW1Y1sgLt7ExZDyN9XCakkleqXW\_ev2sufQmCDk1R1RRskA3kHXBLn3vW9nuGXxU1i7oQ

You can copy the token into the debugger provided by jwt.io to see the payload of the token:



With a basic understanding of authentication types and types of clients we will now focus on transport authentication, known as service-to-service authentication. Istio provides a full stack of support for mutual TLS.

Mutual TLS is an authentication technique to ensure the authenticity of the clients to the server and vice versa. It facilitates authentication via certificates followed by the establishment of an encrypted channel between the parties.

The enablement of mTLS can be easily enabled without requiring any service code changes. Mutual TLS provides the following features:

* Provides each service with a strong identity representing its role to enable interoperability across clusters and clouds.
* Secures service-to-service communication and end-user-to-service communication.
* Provides a key management system to automate key and certificate generation, distribution, and rotation.

# Mutual TLS authentication

Istio tunnels service-to-service communication through the client side and server side Envoy proxies. For a client to call a server, the steps followed are:

1. Istio re-routes the outbound traffic from a client to the client’s local proxy (Envoy).
2. The client side proxy starts a mutual TLS handshake with the server side proxy. During the handshake, the client side proxy also does a secure naming check to verify that the service account presented in the server certificate is authorized to run the target service.
3. The client side proxy and the server side proxy establish a mutual TLS connection, and Istio forwards the traffic from the client side proxy to the server side proxy.
4. After authorization, the server side proxy forwards the traffic to the server service through local TCP connections.

When configuring mutual TLS you have options to specify the kind to be either of a MeshPolicy or Policy. A policy of “MeshPolicy” specifies that all workloads in the mesh will only accept encrypted requests using TLS. The other option is to specify “Policy” which lets you specify mutual TLS for specified namespaces or services.

The example will demonstrate using a type of “Policy” and establishing mutual TLS between specified services. Let’s first execute communication between the services without mTLS configured. This is the default behavior when services are deployed.

The two services are deployed without any policy applied. A review of the Prometheus graph shows that there weren’t any mutual TLS connections happening between the two services.

As demonstrated the default service-to-service communication is without mutual TLS. This may be acceptable for some proof-of-concepts and development purposes; however, in production it is recommended to have the data encrypted between the services. In addition, the client and server are authenticated to one another.

We deploy a policy to enable mTLS. The policy states to use mTLS in the “default” namespace and the target is the “weather-proxy-service”.

apiVersion: authentication.istio.io/v1alpha1

kind: Policy

metadata:

name: mtls-to-weatherproxy

namespace: default

spec:

targets:

- name: weather-proxy-service

peers:

- mtls: {}

The policy is deployed, but as you can see the invocation to the second service failed with a 503.

{"Exception:":"Server returned HTTP response code: 503 for URL: http://weather-proxy-service.default.svc.cluster.local:3100/forecast/zip/90224/units/imperial","Status:":"Failed"}

The reason for the failure is the policy is in place, but the server has not been configured to accept the TLS traffic. To have the server side accept TLS traffic a destination rule needs to be deployed. That destination rule is defined as below.

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

name: weatherproxy-istio-mtls

spec:

host: weather-proxy-service

trafficPolicy:

tls:

mode: ISTIO\_MUTUAL

The mTLS established was by service. A review of the Policy specifies the target as “weather-proxy-service” and for the “DestinationRule” the host is specified, which is the same as the target. The TLS mode required is ISTIO\_MUTUAL. There are four different options for the TLS mode.

DISABLE: Do not setup a TLS connection to the upstream endpoint

SIMPLE: Originate a TLS connection to the upstream endpoint

MUTUAL: Secure connections to the upstream using mutual TLS by presenting client certificates for authentication.

ISTIO\_MUTUAL: Secure connections to the upstream using mutual TLS by presenting client certificates for authentication. Compared to Mutual mode, this mode uses certificates generated automatically by Istio for mTLS authentication. When this mode is used, all other fields in TLSSettings should be empty.

You may be wondering where the certificates for authentication came from. You explicitly didn’t pass any certificates, so what certificates were passed in the TLS connection? Remember the discussion earlier about service accounts and the certificates that are automatically created? Well, these are the certificates that are being exchanged between the client and server.

Even though there was not an in-depth discussion of an external user invoking the service and getting authenticated, the same process demonstrated here would be the same for the external user. The external client must pass a jwt token, have it validated by the authentication plug-in, and then the identity is verified. What was shown here accomplished the same thing; however, it was the service accounts that were used to establish the mutual TLS. Remember, the token for a service account is nothing more than a jwt token.

Now that we have a basic understanding of authentication let’s move on to authorization. For the authorization demonstration we need to do a bit of preparation work. We will create two service accounts, assign those service accounts to different namespaces, and deploy each service to one of the namespaces created.

The first demonstration will show how to do authorization by a namespace. After the first demonstration we will then show authorization by a specific service. Let’s begin.

First, we will clean up the services that we had recently deployed and then deploy those services with service accounts and to different namespaces.

**Weather Proxy Service Greet and Weather Service**

apiVersion: v1

kind: ServiceAccount

metadata:

name: ***greet-and-weather-admin***

namespace: ***greet-ns***

---

apiVersion: v1

kind: Service

metadata:

name: greetandweather-service

namespace: ***greet-ns***

labels:

app: greeter-weather

version: v1

spec:

. . . . .

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: greetandweather-service-v1

namespace: greet-ns

labels:

app: greeter-weather

version: v1

spec:

replicas: 1

. . . . .

spec:

serviceAccountName: ***greet-and-weather-admin***

containers:

. . . . .

apiVersion: v1

kind: ServiceAccount

metadata:

name: ***weather-proxy-sa***

namespace: ***default***

---

kind: Service

apiVersion: v1

metadata:

name: weather-proxy-service

namespace: ***default***

labels:

app: weather-proxy

version: v1

spec:

. . . . .

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: weather-proxy-service-v1

namespace: default

labels:

app: weather-proxy

version: v1

spec:

replicas: 1

. . . . . .

spec:

serviceAccountName: ***weather-proxy-sa***

containers:

. . . . . .

The definition of the two services are provided above. The “Greet and Weather Service (GWS)” invokes the “Weather Proxy Service” when the URI is /weather. The “Weather Proxy Service (WPS)” will then make an outbound call to the open weather API service. This is an external service from the cluster. In case you’re wondering, yes, a ServiceEntry has been deployed to establish an egress gateway. Deploy these two services to the cluster. You will notice that the services are being deployed to two different namespaces. The “WPS” is deployed to the default namespace and the “GWS” is deployed to the “greet-ns”. In addition, take note that the ServiceAccounts created are also being assigned to specific namespaces and Pods. This is important to take note. Keep in mind the discussion previously about service accounts.

$ kubectl apply -f weather-proxy-svc-sa.yaml

serviceaccount "weather-proxy-sa" created

service "weather-proxy-service" created

deployment.extensions "weather-proxy-service-v1" created

$ kubectl apply -f greeter-weather-svc-sa.yaml

serviceaccount "greet-and-weather-admin" created

service "greetandweather-service" created

deployment.extensions "greetandweather-service-v1" created

Let’s verify that the ServiceAccounts, services, and deployments have been created.

$ kubectl get sa

NAME SECRETS AGE

default 1 9d

weather-proxy-sa 1 3m

We requested that the ServiceAccounts be shown; however, the “greet-and-weather-admin” did not appear in the list. Why? The ServiceAccount, “greet-and-weather-admin”, is in the namespace “greet-ns”. To list that ServiceAccount you need to specify the namespace, as such.

$ kubectl get sa -n greet-ns

NAME SECRETS AGE

default 1 7d

greet-and-weather-admin 1 4m

You should also notice that for every namespace there is also a default ServiceAccount, called “default”. Let’s verify that the services are deployed properly and will return a successful response when invoked.

$ curl -v http://129.146.149.89/Sherwood/weather/90224/units/imperial

\* Trying 129.146.149.89...

\* TCP\_NODELAY set

\* Connected to 129.146.149.89 (129.146.149.89) port 80 (#0)

> GET /Sherwood/weather/90224/units/imperial HTTP/1.1

> Host: 129.146.149.89

> User-Agent: curl/7.54.0

> Accept: \*/\*

>

< HTTP/1.1 200 OK

< content-type: application/json

< date: Sat, 12 Jan 2019 00:15:23 GMT

< x-envoy-upstream-service-time: 1513

< server: envoy

< transfer-encoding: chunked

<

\* Connection #0 to host 129.146.149.89 left intact

{"coord":{"lon":-118.31,"lat":33.78},"weather":[{"id":721,"main":"Haze","description":"haze","icon":"50d"}],"base":"stations","main":{"temp":62.08,"pressure":1016,"humidity":77,"temp\_min":59,"temp\_max":64.94},"visibility":14484,"wind":{"speed":5.82,"deg":170},"clouds":{"all":1},"dt":1547250900,"sys":{"type":1,"id":4699,"message":0.0047,"country":"US","sunrise":1547305111,"sunset":1547341490},"id":420004581,"name":"Rolling Hills","cod":200}

We have success; therefore, we know that everything is deployed and configured properly. Now let’s set up authorization. We will demonstrate how to protect services by namespace first.

You enable Istio authorization using a RbacConfig object. This object is a mesh-wide singleton with a fixed name value of default. You can only use one RbacConfig instance in the mesh. The RbacConfig object is a Kubernetes CustomResourceDefinition (CRD) object.

In the RbacConfig object, the operator can specify a mode value, which can be:

* OFF: Istio authorization is disabled.
* ON: Istio authorization is enabled for all services in the mesh.
* ON\_WITH\_INCLUSION: Istio authorization is enabled only for services and namespaces specified in the inclusion field.
* ON\_WITH\_EXCLUSION: Istio authorization is enabled for all services in the mesh except the services and namespaces specified in the exclusion field.

For our demonstration purposes we will use an ON\_WITH\_INCLUSION mode.

apiVersion: "rbac.istio.io/v1alpha1"

kind: RbacConfig

metadata:

name: default

spec:

mode: 'ON\_WITH\_INCLUSION'

inclusion:

namespaces: ["default"]

## The RbacConfig object states to enable authorization for all services that are in the “default” namespace. We will now demonstrate the protection of a service by a namespace. Recall that the “WPS” service is deployed to the “default” namespace. With the RbacConfig deployed then all services in the “default” namespace have authorization enabled. To prove that the “WPS” now has authorization enabled let’s execute the curl command again.

$ kubectl apply -f rbac-config-enable.yaml

rbacconfig.rbac.istio.io "default" created

With the RbacConfig deployed another attempt to submit the curl command shows a different response. The service returns a HTTP 403 error code – the requester is not authorized to invoke the “WPS” service.

{"Exception:":"Server returned HTTP response code: 403 for URL: [http://weather-proxy-service.default.svc.cluster.local:3100/forecast/zip/90224/units/imperial","Status:":"Failed](http://weather-proxy-service.default.svc.cluster.local:3100/forecast/zip/90224/units/imperial%22,%22Status:%22:%22Failed)"}

To rectify the failure you must deploy an Authorization Policy. The Authorization Policy is made up of two parts – the ServiceRole and a ServiceRoleBinding. The ServiceRole and ServiceRoleBinding objects will be discussed.

The ServiceRole defines a group of permissions to access services, which is a specification that includes a list rules, or permissions. Each rule consists of the following standard fields:

* Services: A list of service names. You can set the value to \* to include all services in the specified namespace.
* Methods: A list of HTTP method names, such as GET, POST, etc. You can set the value to \* to include all HTTP methods. If it is a gRPC request then the verb is always POST.
* Paths: This can either be HTTP or gRPC methods. The gRPC methods must be in the form of /packageName.serviceName/methodName and are case sensitive.

A ServiceRole specification only applies to the namespace specified in the metadata section. The services and methods fields are required in a rule. If a rule is not specified or it is set to \*, it applies to any instance.

The ServiceRoleBinding specification includes two parts:

* roleRef refers to a ServiceRole resource in the same namespace.
* A list of subjects that are assigned to the role.

You can either explicitly specify a subject with a user or with a set of properties. A property in a ServiceRoleBinding subject is similar to a constraint in a ServiceRole specification. A property also lets you use conditions to specify a set of accounts assigned to this role. It contains a key and its allowed values.

With the understanding of the ServiceRole and ServiceRoleBinding we can now create an authorization policy for the deployed services. The authorization policy below allows the “GWS” to invoke the “WPS” service. Let’s take a look.

apiVersion: "rbac.istio.io/v1alpha1"

kind: ServiceRole

metadata:

name: greeter-weather-requester

namespace: default

spec:

rules:

- services: ["\*"]

methods: ["GET"]

constraints:

- key: "destination.labels[app]"

values: ["weather-proxy"]

---

apiVersion: "rbac.istio.io/v1alpha1"

kind: ServiceRoleBinding

metadata:

name: greeter-weather-requester-binding

namespace: default

spec:

subjects:

- properties:

source.namespace: "istio-system"

- properties:

source.namespace: "default"

- properties:

source.namespace: "greet-ns"

roleRef:

kind: ServiceRole

name: "greeter-weather-requester"

## Let’s dissect the authorization policy. Taking a look at the rules it specifies that all services in the default namespace can be accessed. However, look at the constraints. The constraints are limiting this access to only one service. The service that has the label, “weather-proxy”. The rules also specify that only the GET HTTP is allowed.

## A dissection of the ServiceRoleBinding specifies the subjects which can access the service. A subject can either be a user or a set of properties. In this policy, the specification dictates a set of properties. These properties state that any of the requesters that are from one of the three namespaces specified have authorization to invoke the service(s) indicated in the ServiceRole, stated by the roleRef: “greeter-weather-requester”.

## The deployment of the authorization policy grants permission to the “GWS” service to access the “GPS” service.

$ kubectl apply -f greeter-weather-namespace-policy.yaml

servicerole.rbac.istio.io "greeter-weather-requester" created

servicerolebinding.rbac.istio.io "greeter-weather-requester-binding" created

When the curl command is executed a successful response is returned.

$ curl -v http://129.146.149.89/Sherwood/weather/90224/units/imperial

< HTTP/1.1 200 OK

< content-type: application/json

< date: Sat, 12 Jan 2019 01:35:03 GMT

< x-envoy-upstream-service-time: 364

< server: envoy

< transfer-encoding: chunked

<

\* Connection #0 to host 129.146.149.89 left intact

{"coord":{"lon":-118.31,"lat":33.78},"weather":[{"id":721,"main":"Haze","description":"haze","icon":"50n"}],"base":"stations","main":{"temp":58.55,"pressure":1016,"humidity":87,"temp\_min":55.4,"temp\_max":60.8},"visibility":16093,"wind":{"speed":9.17,"deg":140},"clouds":{"all":75},"dt":1547254680,"sys":{"type":1,"id":6037,"message":0.1366,"country":"US","sunrise":1547305111,"sunset":1547341493},"id":420004581,"name":"Rolling Hills","cod":200}

## We exhibited how to protect resources by a namespace. We will now change the authorization policy to implement Service-level access control. First let’s remove the Namespace-level access control authorization policy by doing a delete with the kubectl CLI.

## A review of the Service-level access control is given below:

apiVersion: "rbac.istio.io/v1alpha1"

kind: ServiceRole

metadata:

name: weather-viewer

namespace: default

spec:

rules:

- services: ["weather-proxy-service.default.svc.cluster.local"]

methods: ["GET"]

---

apiVersion: "rbac.istio.io/v1alpha1"

kind: ServiceRoleBinding

metadata:

name: weather-viewer-binding

namespace: default

spec:

subjects:

- user: "cluster.local/ns/greet-ns/sa/greet-and-weather-admin"

roleRef:

kind: ServiceRole

name: "weather-viewer"

The ServiceRole rules state that the policy applies to the weather-proxy-service in the default namespace and HTTP “GET” operations are allowed. The binding specifies the user that is allowed to access the service. The user in this case is the ServiceAccount user we had created earlier.

Another scenario to demonstrate another authorization policy which is very similar to the one just shown. The difference in this scenario is that the authorization policy is granting permission for all users. This can be seen in the subjects section of the ServiceRoleBinding. The user parameter contains “\*”. This wildcard states that all requesters are granted permission to invoke the service specified in the ServiceRole mentioned by the roleRef in the ServiceRoleBinding.

apiVersion: "rbac.istio.io/v1alpha1"

kind: ServiceRole

metadata:

name: weather-viewer

namespace: default

spec:

rules:

- services: ["weather-proxy-service.default.svc.cluster.local"]

methods: ["GET"]

---

apiVersion: "rbac.istio.io/v1alpha1"

kind: ServiceRoleBinding

metadata:

name: weather-viewer-binding

namespace: default

spec:

subjects:

- user: "\*"

roleRef:

kind: ServiceRole

name: "weather-viewer"