

Enforcing Service Mesh Structure using Gatekeeper

Sandeep Parikh
@crcsmnky
Google Cloud



Hi, I'm Sandeep



I write code, best practices, and work with ops teams to build and operate cloud native infrastructure.

Find me **@crcsmnky** on [Twitter](#) and [Github](#).

What are policies?

What are policies?

Rules that tell us whether we can
make changes to a resource

What is policy management?

What is policy management?

The practice of developing, deploying,
and using policy objects

Config Management

How you store/control/govern deployment configuration

Process for it (GitOps) and for who can do it (RBAC)

Config management enforces object/resource state

VS.

Policy Management

Policies govern the resource changes that can be made

Allows enforcement over whether changes can be applied

Policies can admit/deny/audit new or existing cluster resources

Guardrails & Governance

Kubernetes is powerful,
and needs controls



Watch for over-granted
privileges



Lock-down exposed services



Prevent data exfiltration

Controls need to
be flexible & agile



Limit scope to only what's
necessary



Delegate access & control to
subject matter experts



Facilitate safe deploys and
continuous monitoring

How do you
enforce structure?

Open Policy Agent

Open Policy Agent (OPA) is a general-purpose policy engine with uses ranging from authorization and admission control to data filtering.



Declarative

Express policies in a high-level declarative language that promotes safe, fine-grained logic.



Unified

Decouple policy decisions from services to achieve unified control across the entire stack.

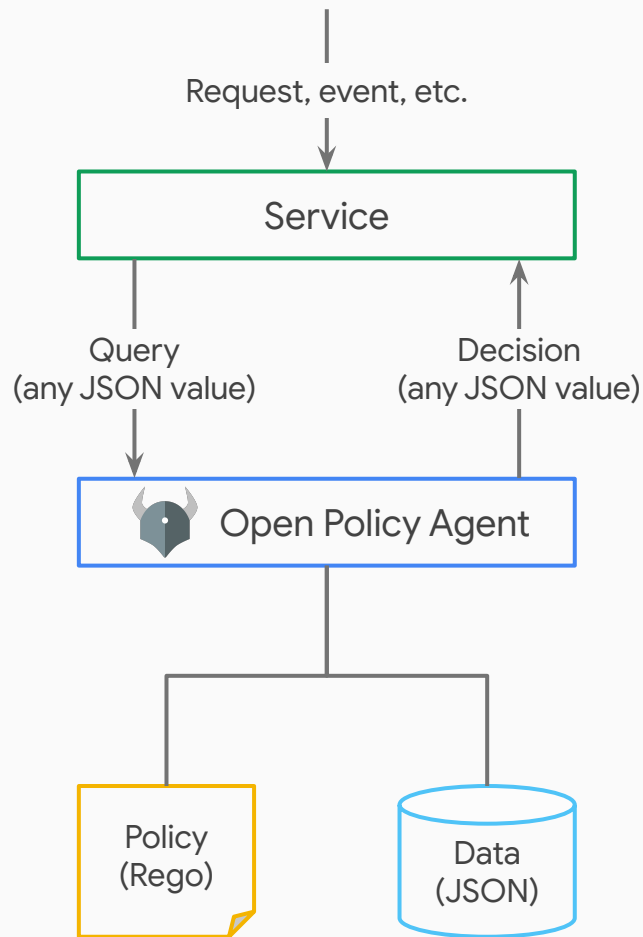


Context Aware

Leverage arbitrary external data in policies to ensure that important requirements are enforced.

Open Policy Agent

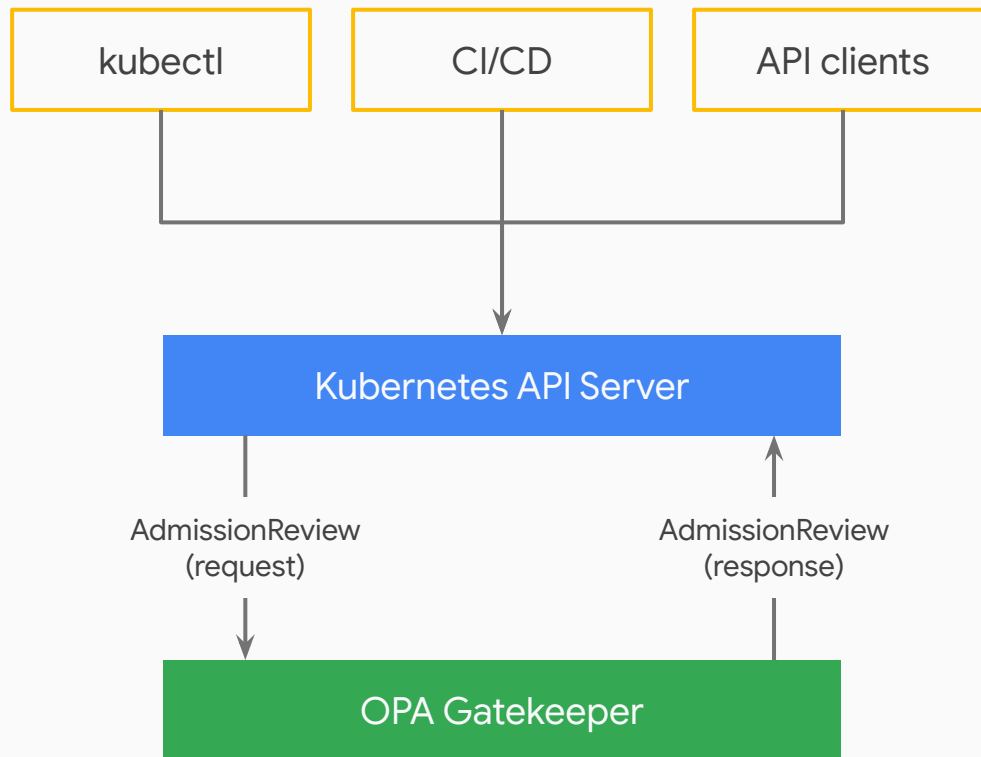
When your software needs to make policy decisions, it queries OPA and supplies structured data (JSON) as input.



Gatekeeper

OPA Gatekeeper provides first-class integration between OPA and Kubernetes via a custom controller.

Gatekeeper turns Rego policies into Kubernetes objects, allowing them to be customized and deployed using standard workflows.



Policy Objects

Policies are written in Rego and packaged as parameterized `ConstraintTemplate` objects.

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: destinationruletlsenabled
spec:
  crd:
    spec:
      names:
        kind: DestinationRuleTLSEnabled
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package asm.guardrails.destinationruletlsenabled

        # spec.trafficPolicy.tls.mode == DISABLE
        violation[{"msg": msg}] {
          d := input.review.object
          tlsdisable := { "tls": {"mode": "DISABLE"}}

          ktpl := "trafficPolicy"
          tpl := d.spec[ktpl][_ ]
          not tpl != tlsdisable["tls"]

          msg := sprintf("%v %v.%v mode == DISABLE",
            [d.kind, d.metadata.name, d.metadata.namespace])
        }
      }
```

Policy Objects

Policies are written in Rego and packaged as parameterized `ConstraintTemplate` objects.

The `ConstraintTemplate` extends Gatekeeper by adding a new policy that can be invoked via a new CR.

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: destinationruletlsenabled
spec:
  crd:
    spec:
      names:
        kind: DestinationRuleTLSEnabled
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package asm.guardrails.destinationruletlsenabled

        # spec.trafficPolicy.tls.mode == DISABLE
        violation[{"msg": msg}] {
          d := input.review.object
          tlsdisable := { "tls": {"mode": "DISABLE"}}

          ktpl := "trafficPolicy"
          tpl := d.spec[ktpl][_ ]
          not tpl != tlsdisable["tls"]

          msg := sprintf("%v %v.%v mode == DISABLE",
            [d.kind, d.metadata.name, d.metadata.namespace])
        }
```

Policy Objects

Constraints are instantiations of a **ConstraintTemplate** CR and can be optionally scoped to specific objects and/or namespaces.

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: DestinationRuleTLSEnabled
metadata:
  name: dr-tls-enabled
spec:
  enforcementAction: deny
  match:
    kinds:
      - apiGroups: ["networking.istio.io"]
        kinds: ["DestinationRule"]
    namespaces: ["default"]
#   alternatively, scope by a selector
#   namespaceSelector:
#     matchExpressions:
#       - key: istio-injection
#         operator: In
#         values: ["enabled"]
```

Policy Objects

Constraints are instantiations of a **ConstraintTemplate** CR and can be optionally scoped to specific objects and/or namespaces.

When violated, **Constraints** can either deny admission or allow entry, and audit the violation in the **status** field.

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: DestinationRuleTLSEnabled
metadata:
  name: dr-tls-enabled
spec:
  enforcementAction: deny
  match:
    kinds:
      - apiGroups: ["networking.istio.io"]
        kinds: ["DestinationRule"]
    namespaces: ["default"]
#   alternatively, scope by a selector
#   namespaceSelector:
#     matchExpressions:
#       - key: istio-injection
#         operator: In
#         values: ["enabled"]
```


Gatekeeper Config

Existing cluster objects can be synced into OPA Gatekeeper so that they can be used for complex multi-object policies or for auditing existing resources.

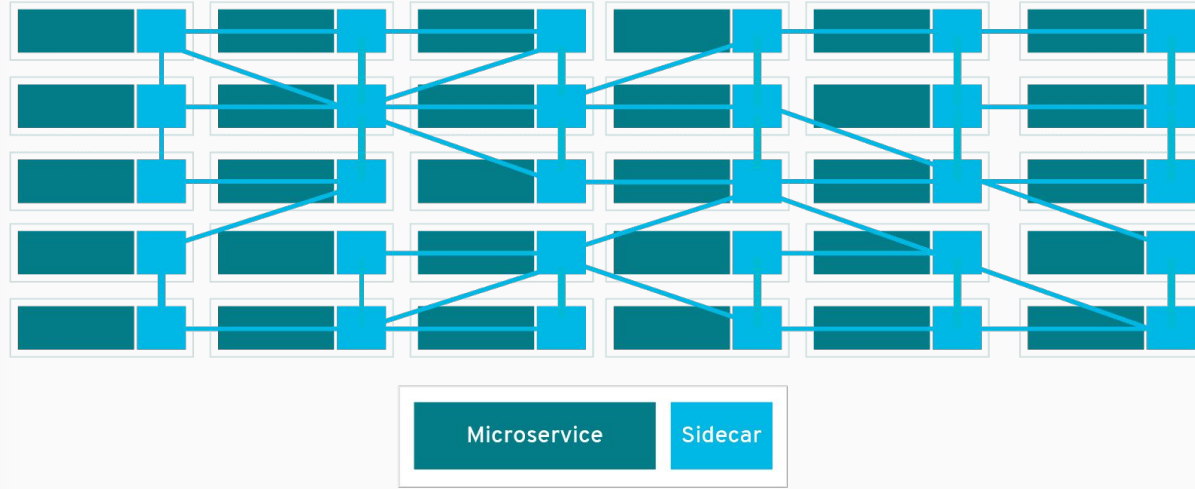
```
apiVersion: config.gatekeeper.sh/v1alpha1
kind: Config
metadata:
  name: config
  namespace: gatekeeper-system
spec:
  sync:
    syncOnly:
      - group: ""
        version: "v1"
        kind: "Namespace"
      - group: ""
        version: "v1"
        kind: "Service"
      - group: "networking.istio.io"
        version: "v1alpha3"
        kind: "Gateway"
      - group: "networking.istio.io"
        version: "v1alpha3"
        kind: "VirtualService"
      - group: "networking.istio.io"
        version: "v1alpha3"
        kind: "DestinationRule"
      - group: "authentication.istio.io"
        version: "v1alpha1"
        kind: "Policy"
```

Applying enforcement to Service Mesh

Why Service Mesh?

More services = more complexity

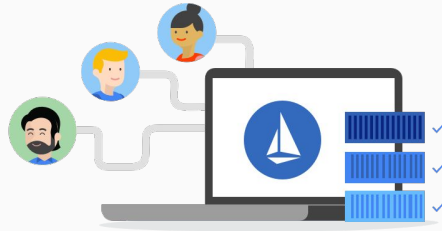
A service mesh provides a transparent and language-independent way to flexibly and easily automate application network functions.



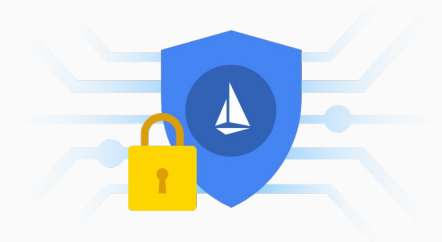
What Istio provides



Uniform
observability



Operational
agility



Policy driven
security

What Istio does



Observability

Examine **everything** happening with your services with minimal instrumentation



Traffic

Manage the flow of traffic **into**, **out of**, and **within** your complex deployments



Security

Secure **access** and **communications** between some or all services

What Istio does

Observability

Traffic

Security

Network automation at scale

Learn **what** is **happening** with your services with minimal instrumentation

Manage the flow of traffic **into**, **out of**, and **within** your complex deployments

Secure **access** to **communications** between some or all services

A cartoon illustration of Kenny McCormick from the animated series South Park. He is in a kitchen, wearing a teal button-down shirt, and is cooking at a stove. He has a wide-eyed, nervous expression. In the background, his family is watching him: his mother (Kathy McCormick) in a pink robe, his sister (Katie McCormick) in a purple shirt, and his brother (Eric McCormick) in a blue hat. On the counter, there is a plate of scrambled eggs, a blender, and a bottle of ketchup. The kitchen has yellow tiled floors and brown cabinets.

I'LL JUST FLIP THIS HERE OMELETTE

AAAAAAND
I'M HAVING SCRAMBLED EGGS

Enmeshing

In order for **Pods** and **Services** to be part of the mesh, they must use specific port-naming conventions.

```
apiVersion: v1
kind: Service
metadata:
  name: app-backend
  labels:
    app: app-backend
spec:
  ports:
    - port: 5000
      name: backend-port
  selector:
    app: app-backend
```


Enmeshing

In order for **Pods** and **Services** to be part of the mesh, they must use specific port-naming conventions.

How do you catch that in advance?

```
apiVersion: v1
kind: Service
metadata:
  name: app-backend
  labels:
    app: app-backend
spec:
  ports:
    - port: 5000
      name: backend-port
  selector:
    app: app-backend
```

Enforcing mTLS

Tell all services in a specific Namespace to only accept mTLS connections using **Policy** objects.

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: default
spec:
  peers:
  - mtls: {}
    mode: STRICT
```

Enforcing mTLS

Tell all services in a specific Namespace to only accept mTLS connections using **Policy** objects.

How do you prevent someone from overriding that governance on a per-host basis?

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: default
spec:
  peers:
  - mtls: {}
    mode: STRICT
```

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: my-app-policy
spec:
  peers:
  - mtls: {}
    mode: PERMISSIVE
  targets:
  - name: app-backend
```

app-backend will accept unauthenticated connections

Mismatched mTLS

Tell services to only accept mTLS connections using **Policy** objects.

Tell clients to use mTLS (if available) using **DestinationRule** objects.

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: app-policy
spec:
  peers:
    - mtls: {}
      mode: STRICT
  targets:
    - name: app-backend
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: app-dest-rule
  namespace: default
spec:
  host: app-backend
  trafficPolicy:
    tls:
      mode: DISABLE
```

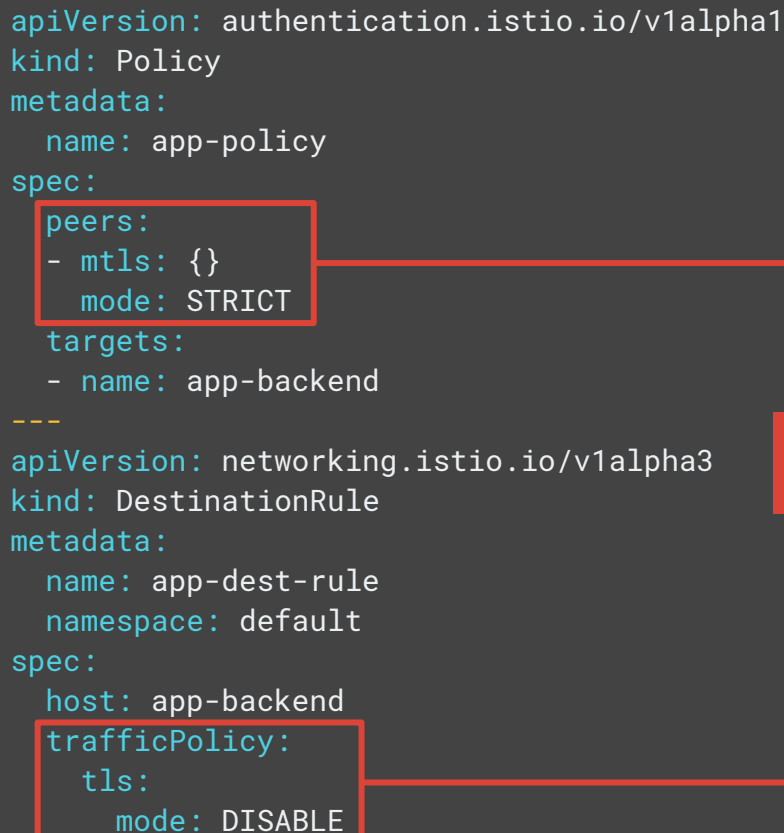
Mismatched mTLS

Tell services to only accept mTLS connections using **Policy** objects.

Tell clients to use mTLS (if available) using **DestinationRule** objects.

What if they don't match?

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: app-policy
spec:
  peers:
  - mtls: {}
    mode: STRICT
  targets:
  - name: app-backend
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: app-dest-rule
  namespace: default
spec:
  host: app-backend
  trafficPolicy:
    tls:
      mode: DISABLE
```



HTTP 503

Authz Controls

Istio's `ServiceRole` and `ServiceRoleBinding` objects

allow you to grant access to specific services and methods based on request, source, or destination attributes.

```
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRole
metadata:
  name: authz-role
spec:
  rules:
  - services: ["backend.foo.svc.cluster.local"]
    methods: ["GET"]
---
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRoleBinding
metadata:
  name: authz-role-binding
spec:
  subjects:
  - properties:
      source.principal: "cluster.local/ns/bar/sa/frontend"
      source.namespace: "test"
  roleRef:
    kind: ServiceRole
    name: "authz-role"
```

Authz Controls

Istio's `ServiceRole` and `ServiceRoleBinding` objects

allow you to grant access to specific services and methods based on request, source, or destination attributes.

How do we ensure that an authz policy doesn't allow access from arbitrary sources?

```
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRole
metadata:
  name: authz-role
spec:
  rules:
  - services: ["backend.foo.svc.cluster.local"]
    methods: ["GET"]
  ---
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRoleBinding
metadata:
  name: authz-role-binding
spec:
  subjects:
  - properties:
    source.principal: "cluster.local/ns/bar/sa/frontend"
    source.namespace: "test"
  roleRef:
    kind: ServiceRole
    name: "authz-role"
```

VirtualServices

Assume you have a multi-tenant deployment with multiple `VirtualServices` using the `istio-ingressgateway`.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: helloworld-v1
spec:
  hosts:
  - "*"
  gateways:
  - helloworld-gateway
  http:
  - match:
    - uri:
        exact: /hello
    route:
  - destination:
        host: helloworld-v1
        port:
            number: 5000
```

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: helloworld-v2
spec:
  hosts:
  - "*"
  gateways:
  - helloworld-gateway
  http:
  - match:
    - uri:
        exact: /hello
    route:
  - destination:
        host: helloworld-v2
        port:
            number: 5000
```


VirtualServices

Assume you have a multi-tenant deployment with multiple `VirtualServices` using the `istio-ingressgateway`.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: helloworld-v1
spec:
  hosts:
  - "*"
  gateways:
  - helloworld-gateway
  http:
  - match:
    - uri:
        exact: /hello
    route:
    - destination:
        host: helloworld-v1
        port:
            number: 5000
```



```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: helloworld-v2
spec:
  hosts:
  - "*"
  gateways:
  - helloworld-gateway
  http:
  - match:
    - uri:
        exact: /hello
    route:
    - destination:
        host: helloworld-v2
        port:
            number: 5000
```

Istio Policies

Gatekeeper allows you to enforce any organizational, regulatory, or compliance policies.



Require strict mTLS for all clients/services in a namespace



Require fine-grained service authorization controls



Require access logging to be enabled for a cluster / mesh



Require services to disable unauthorized access



Require annotations for mesh objects to track ownership



Only allow whitelisted fields in telemetry specifications

Gatekeeper + Istio = Structure

Demos

GKE 1.14.8-gke.2

Istio 1.3.3

Gatekeeper 3.0.4-beta.2



Audit Services for not using
correct port-naming convention



Prevent VirtualService hostname
matching collisions



Require strict mTLS for all
services in a namespace



Require services to disable
unauthorized access

Questions & Resources

Questions?

Find me

@crcsmnky on

[Twitter](#) or [Github](#)

Gatekeeper Policies for Istio

github.com/crcsmnky/gatekeeper-istio

Gatekeeper

github.com/open-policy-agent/gatekeeper

Rego

openpolicyagent.org/docs/latest/policy-language

Thank you