Shared with Community

Auto mTLS: Adoption Simplification

Owner: jianfeih
Contributors: mjog, lita, diemtvu, fpesce
Work-Group: Istio Security
Short self link: https://goo.gl/DP2dxU

Status: WIP | In Review | Approved | **Implemented** |
Created: 08/21/2018
Updated: 10/17/2019
Approvers:

Overview

Transparent mutual TLS is one of the most appealing features for Istio. Adopting it in the real world safely, however, is hard. This design aims at simplifying the mTLS adoption experience.

Safety to rollout mTLS when migrating to Istio initially. *A service consists of workloads with or without sidecar*. This applies to both client and server side service. Rolling out mutual TLS blindly without realizing this just breaks traffic.

Security lockdown after migrating finishes. Customers caring about security must be able to reason about the security posture easily based on the Istio API.

We shall see these two needs conflicting each other by nature. We can optimize API, but the user intent of which is needed at a given movement has to be expressed by the user.

Overview

API Enhancements

Detailed Design
- Introduce Envoy Transport Socket Match
- Label Workloads with mTLSReady Label
- Pilot Configure Client Envoy
- Support Other Environments

mTLS Adoption User Journey
- Migration to Istio via PERMISSIVE
- Security Lockdown via Telemetry Dashboard

Security Analysis

Open Question

Test and Rollout Plan
    Pilot Scalability Testing
    mTLS performance Test
    Rollout Plan

FAQ

Alternatives Considered
    Pilot Tracking Endpoints State
    Opportunistic Encryption

Further Work
    Client Side mTLS stats Data
    Grafana Dashboard for mTLS?

API Enhancements

`AuthenticationPolicy`(Authn) specifies server side requirements of mTLS, PERMISSIVE or STRICT. `DestinationRule`(DR) captures the service owners' intent of how its clients connect, load balance to the service at networking level.

Our API is very error prone and user unfriendly. A common misconfiguration is that customers have trouble addressing the mutual TLS config in the right DR:

A. Customer change Authn policy to STRICT, and forget to update DestinationRule, resulting into breakage.
B. DestinationRule supports wildcard scopes but not inheritance. A DR of `*.foo-namespace.svc.cluster.local` with ISTIO_MUTUAL mTLS enables mTLS for services in foo namespace, but when an operator creates a specific DR for a particular service within the foo namespace, often forget to manual config `ISTIO_MUTUAL` again.
C. Customers can't roll out mTLS when server side workloads haven't fully migrated to Istio with sidecar. DR applies mTLS config to all endpoints of a given service.

We proposes API changes to address above problems:

1. If customers configure a DestinationRule with TLS config, that will be used for backward compatibility.
2. If no DestinationRule exists, or it exists but no TLSSettings is configured, Pilot ***smartly*** configures the client Envoy to achieve auto mTLS. Specifically, client sends mTLS traffic to sidecar-ed server workloads, and plaintext for non-sidecar workloads.

Detailed Design

TODO: overall design diagram in Istio system.

Introduce Envoy Transport Socket Match

TLDR: within a Envoy cluster, we can configure a `transport_socket_matches` to use different transport socket configuration for different endpoints. The matching criteria is based on endpoint label matching.

See Envoy Design Doc for feature details.

With this feature, Istio configures client Envoy to send mTLS traffic to sidecar-ed server endpoint and plaintext to non-sidecar Envoy.

Label Workloads with mTLSReady Label

The Istio sidecar injector will modify PodSpec to annotate the Pod with `security.istio.io/mtlsReady: true`. When an Envoy sidecar is injected, Pod deployment YAML is modified with the following annotation:

```
kind: Deployment
Spec:
  metadata:
    lables:
      security.istio.io/mtlsReady: "true"
```

Pilot Configure Client Envoy

Control plane (Pilot) works as following

1. If customer explictly configures client side mTLS in DestinationRule, Pilot honors that for backward comptability.
2. If no (either no DestinatinoRule is found or `TLSSettings` field is empty)
    a. When auto mTLS is disabled, do nothing plaintext as before.
    b. When Auto mTLS is enabled, configure CDS transport_socket_matches as below.

```
cluster:
   transport_socket_matches:
   - name: mtls
```

```
    match:
    - mtlsReady: "true"
    transport_socket:
      name: "tls"
      config: {} // istio mTLS config, cert, validation, root, etc.
  - name: plaintext
     match: {}
     transport_socket:
        name: "raw_buffer"
```

This feature does not introduce any additional runtime complexity in Pilot. Configuring auto mTLS via transport_socket_matches is trivial as a constant config look up operation.

Support Other Environments

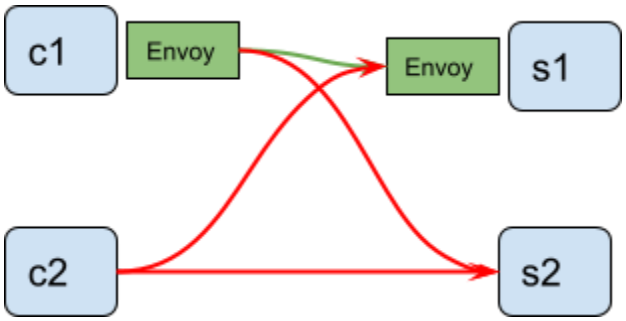
In order for the proposal to work under other environments, as multi cluster, mesh expansion (VM), we simply need to update our documentation to add lablels to `ServiceEntry` with `security.istio.io/mTLSReady: true` to endpoints label.

Tooling via `istioctl` can be provided to simplify UX. Note currently incremental EDS updates, which this design depends on to work efficiently, does not support `ServiceEntry` discovery. We can and should add support for that as well.

mTLS Adoption User Journey

Migration to Istio via PERMISSIVE

Customers install Istio with `PERMISSIVE` on server side, and no DestinationRule. Server and client workloads can both have partial Envoy sidecar injected.



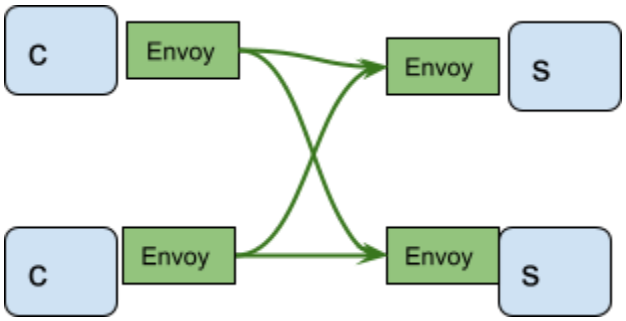
Traffic between c1/s1 is in mTLS and nothing breaks.

Customer Steps, no configuration/steps are required for customer for both approaches.

Security Lockdown via Telemetry Dashboard

Before locking down the server side to only accept mTLS traffic, service operators must ensure it can be done safely. Legacy clients can still rely on services being in `PERMISSIVE` mode in order to continue to work.

Effectively service operators need to ensure the data plane traffic are 100% mTLS encrypted by Istio via telemetry information.



We need to have server side mTLS telemetry data to achieve that.

Customer Steps, service operators

- *Checks Istio mTLS dashboard to ensure last X days 100% traffic is mTLS*
- *Change policy to be STRICT.*

Security Analysis

This feature auto upgrade more traffic to mutual TLS, enhancing the default security posture. Instead we should focus on whether this auto mechanism is reliable and can be immune to plain text downgrade.

mTLS decision is based on endpoint labeling. In order to downgrade connection to plaintext, attacker must be able to take over EDS labeling results, which is guarded by Kubernetes API server admission control, (pods for k8s case, ServiceEntry CRD update permission for other cases).

Open Question

Do we need DR for client side lockdown? `ISTIO_MUTUAL` for lock down on client side or not?

Currently client Envoy is configured with `transport_socket_matches` to be able to send both mTLS and plaintext, even if the server side is STRICT.

Test and Rollout Plan

As the moment, auto mTLS is enabled on by default in master branch, running all CI/CD test since 1.4 release is cut.

Functionality Test

Feature-wise, we have E2E test to ensure the mTLS connectivity when feature is enabled.

Additional work might be scoped to cover more edge cases,

1. Multi cluster.
2. Mesh expansion.
3. Headless service, TCP.
4. Server first protocols as MySQL.

1) and 2) will be both covered by community testing. 1) is currently also having *some* CI integration test, and auto mTLS works fine with it in master.

Open to more ideas on 3)? What are more TCP traffic pattern we need to test?

Pilot Scalability Testing

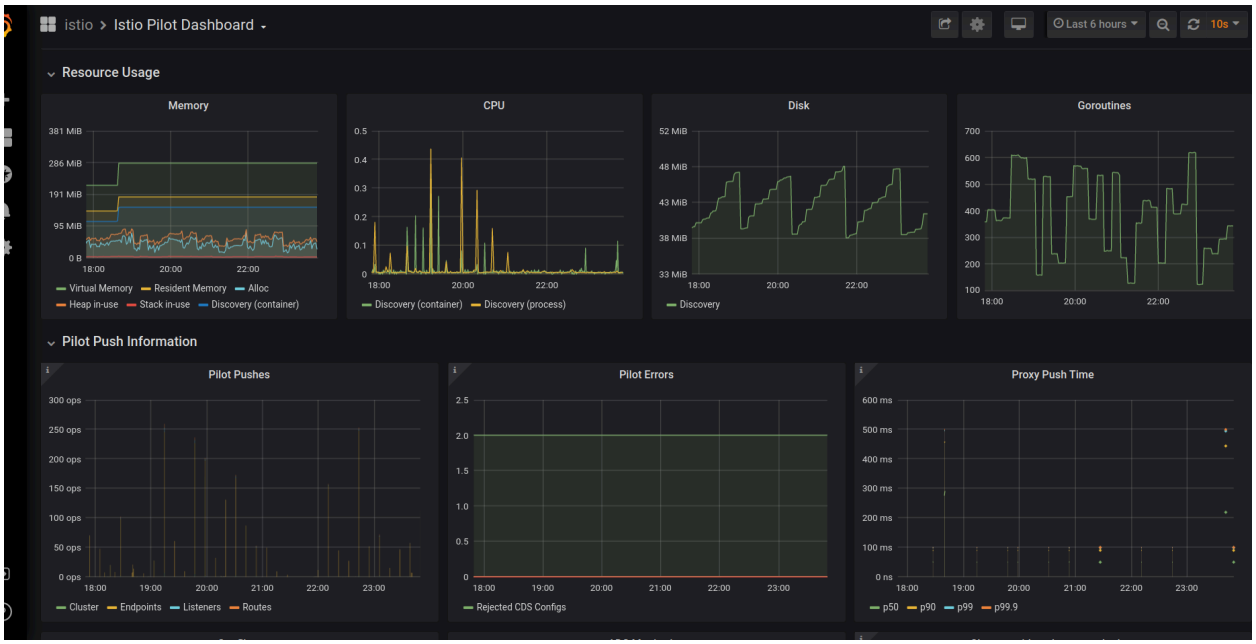
Automatic mTLS allows customers to roll out Istio sidecar and let Istio automatically encapsulate the traffic. Thus we design our test setup to simulate a sidecar rollout/rollback situation:

1. Client (frontend) is sending HTTP traffic to server.
2. Server are constantly scaled up and down with different portions of the sidecars are injected.

Throughout this test running, we see the success rate remaining stable, and no major Pilot performance differences.

With automatic mutual TLS,

Without automatic mutual TLS,

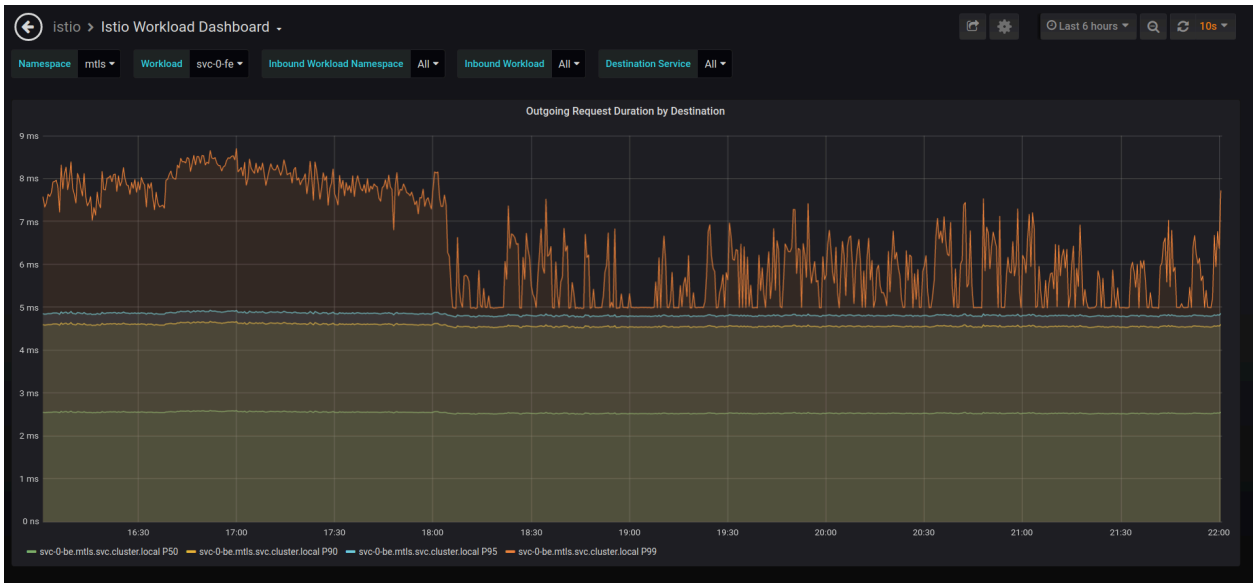


mTLS performance Test

This feature automatically enrolls traffic to mutual TLS, which can incur additional data plane overhead. We have fortio the client and server serving HTTP traffic, running under plaintext and mTLS. We observed

1. 12% additional `istio-proxy` container CPU usage. mTLS to plaintext change:

2. P99 latency increased from 6ms to 8ms.



For the detailed setup, please see here.

Long running test suite as part of release qualification test, to ensure the stability and performance of Pilot

a. Large number of pod/endpoints as basic echo client/server setup.
b. Toggle the sidecar injector on and off for new deployments to simulate the rollout and opt-out of rollout, trigger CDS mutual TLS configuration decision changes.
c. Toggle Authentication Policy.
d. Measure CPU/Memory/Data Plane HTTP 200 ratio.

Rollout Plan

Currently, auto mTLS is controlled via `MeshConfig.EnableAutomaticMtls`. This applies to entire mesh. To enable the feature, changes are applied to entire mesh at once. We learned Env WG is planning to add support for easier feature rollout using Istio revisions.

Admin thus can roll out auto mTLS using different Istio control plane revisions, and gradually migrate workloads to target different Istio versions.

Auto mTLS Verification

Users need a way to verify auto mTLS is in effect. This is as same as before, checking Grafana which displays mTLS properties of inbound traffic:

- mTLS effect, server workloads showing traffic shifted from plaintext to mTLS.
- Data plane perf: client workload dashboard, showing the outbound request duration, success rate.
- Control plane perf, perf dashboard for Pilot CPU/Memory etc.

FAQ

Why name the annotation as mtlsReady rather than "hasSidecar"?

We carefully choose to use `mtlsReady` instead of referring to "sidecar". This allows us to support other potential workload deployments that can accept mTLS traffic. For example, a gRPC implementation can have the same mTLS capability as Envoy does.

Why adding new mode in DestinationRule.TLSSettings rather than a boolean in MeshConfig?

DestinationRule already has 3 TLS modes, why adding another one, won't that be more confusing?

1. Global config settings like MeshConfig have the risk of global outage in case of misconfiguration or bug in Istio software.
2. We would like make this feature default in future Istio release (1.5 and beyond), but for availability critical customers, global config change is no-go due to the potential risk.
3. We already have "*.cluster.local" DestinationRule in `istio-system` namespace for mesh wide defaults. While Istio API, Sidecar/DestinationRule can evolve, adding to something existing already is reasonable.
4. Ultimately customers don't have to touch DestinationRule at all given to inheritance from `istio-config` root namespace.

This design proposes the feature controlled by a MeshConfig flag, why not Istio API?

The improvements in this doc have to be guarded via a feature switch. This design uses a mesh config flag plus per deployment annotations for incremental opt-in the feature. Other alternative to control the feature has been suggested, e.g. extending Istio API by adding another mode in DestinationRule TLS. That is not taken because

1. More user cognitive overhead.
2. The feature eventually becomes a default and the only behavior after proven to be stable. We don't need to opt-in and out very dynamically, which is what Istio API's benefits.

Alternatives Considered

Pilot Tracking Endpoints State

TODO: summary of previous attempt of a solution purely based on control plane tracking endpoints state.

Opportunistic Encryption

TODO: write it up, reason we abaond the approach is because this introduce the mTLS downgrade attack.

Further Work

Make this work for workload-oriented (i.e using label selector, instead of service host) authN API?

Client Side mTLS stats Data

This is a WIP changes on Istio Telemetry WG.

Grafana Dashboard for mTLS?