Concept

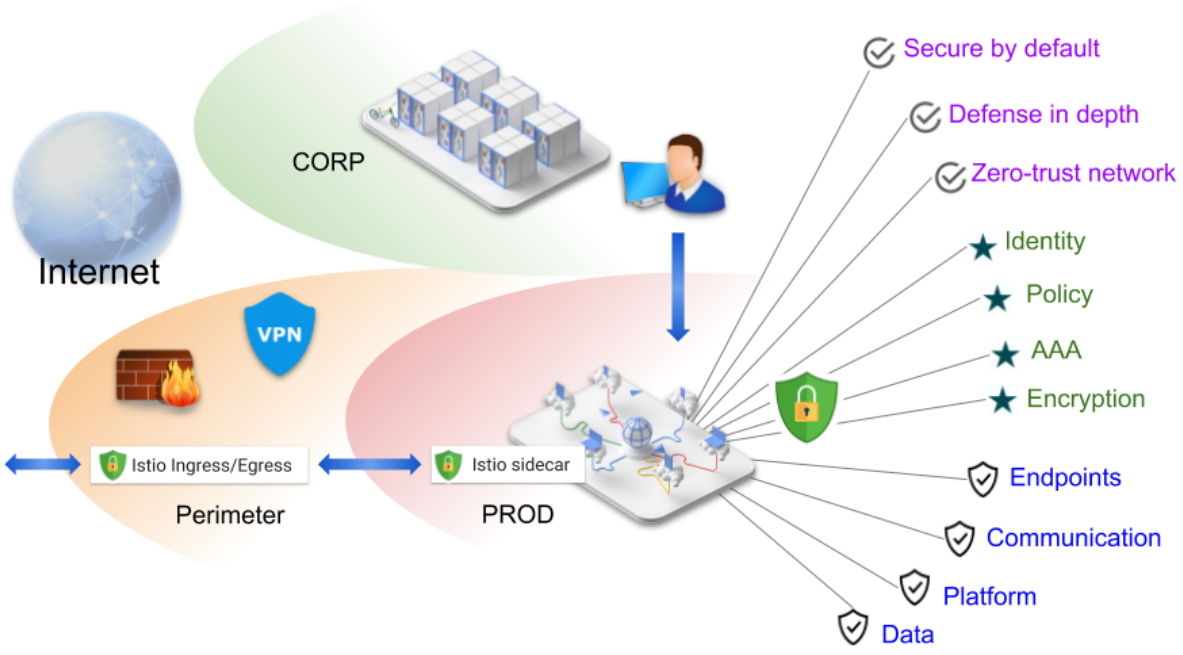
Security

Monolithic 애플리케이션을 원자 서비스로 세분화하면 대응력 향상, 확장성 향상, 서비스 재사용 능력 향상 등 다양한 이점을 얻을 수 있다. 그러나 마이크로 서비스에는 다음과 같은 특정 보안 요구 사항도 있다.

- man-in-the-middle attack를 방어하기 위해서는 Traffic 암호화가 필요
- 유연한 서비스 접근 제어를 위해서는 상호 TLS(Mutual TLS)와 세분화된 접근 정책(fine-grained access policies)이 필요하다.
- 누가 언제 무엇을 했는지 감사하기 위해서는 감사 도구가 필요하다.

Istio Security는 이러한 모든 문제를 해결하기위한 포괄적 인 보안 솔루션을 제공하려고합니다.

이 페이지는 Istio 보안 기능을 사용하여 서비스를 어디에서 실행하든 보안을 유지하는 방법에 대한 개요를 제공한다. 특히 Istio 보안은 data, endpoints, communication 및 platform에 대한 내부 및 외부 위협(insider and external threats)을 모두 완화한다.

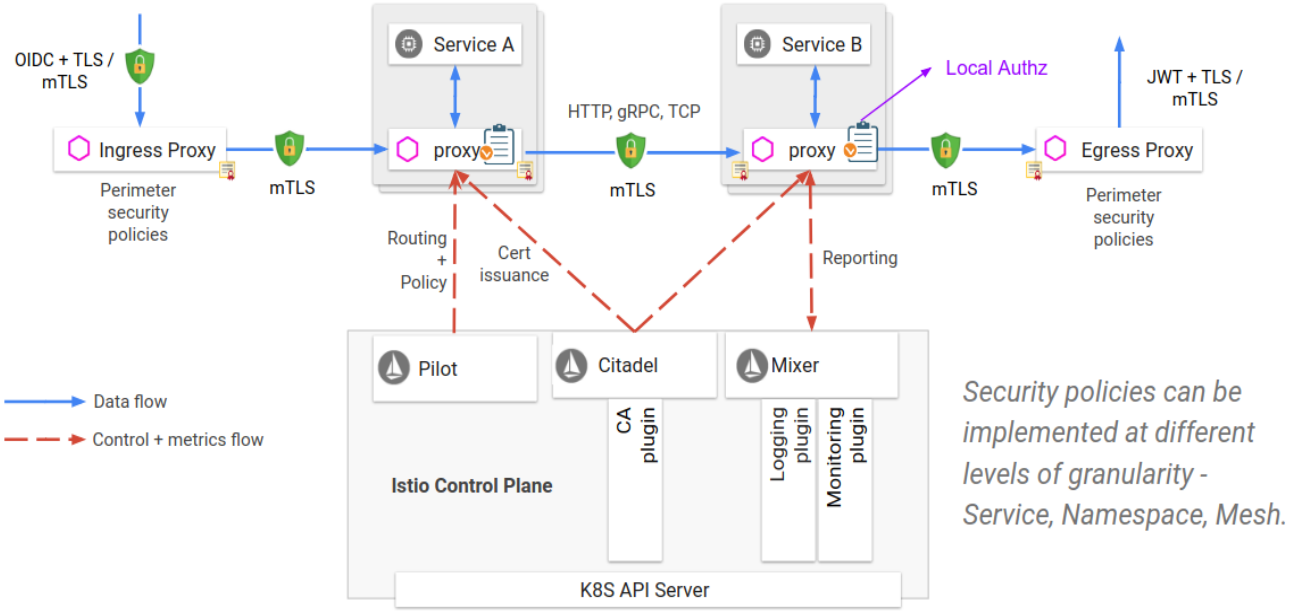


Istio Security Overview

High-level architecture

Istio의 보안에는 여러 가지 구성 요소가 포함된다.

- Key 및 인증서 관리를 위한 Citadel
- 클라이언트와 서버 간의 안전한 통신을 구현하기 위한 Sidecar 및 주변 proxies (perimeter proxies)
- Authentication 정책을 배포하고 프록시에 이름 정보를 안전하게 지정하기 위한 Pilot
- Mixer를 통해 Authorization 및 Auditing



Istio Security Architecture

Istio identity

mutual authentication purposes credentials with their identity information server's identity is checked against the secure naming information the client can access based on the authorization policies the first-class service identity to determine the identity of a service Istio PKI는 Istio Citadel 위에 구축

ID(Identity)는 모든 보안 인프라의 기본 개념입니다. service-to-service communication의 시작에서 양 당사자는 **상호 인증(mutual authentication) 목적으로 신원(identity) 정보와 신임 정보(credentials)를 교환**해야합니다.

클라이언트 측에서는 **서버의 신원(identity)을 보안 명명 정보(secure naming information)와 대조하여 인증된 서비스 Runner인지** 확인합니다.

서버 측에서는 서버가 **인증 정책(authorization policies)을 기반으로 클라이언트가 액세스 할 수있는 정보를 결정**하고, 누가 언제 어떤 시간에 액세스했는지, 사용한 서비스를 기반으로 고객에게 요금을 청구하고, 청구서를 지불하지 않은 클라이언트를 서비스를 액세스하지 못하도록 거부 할 수 있습니다.

Istio ID(identity) 모델에서 Istio는 **First-class 서비스 (identity)를 사용하여 서비스의 ID를 결정**합니다. 따라서 사용자, 개별 서비스 또는 서비스 그룹을 나타낼 수있는 뛰어난 유연성과 세분성이 제공됩니다. 이러한 ID를 사용할 수없는 플랫폼에서는 Istio가 서비스 이름과 같은 서비스 인스턴스를 그룹화 할 수있는 다른 ID를 사용할 수 있습니다.

다른 플랫폼의 Istio 서비스 ID :

- Kubernetes : Kubernetes service account
- GKE / GCE : GCP service account를 사용할 수 있습니다.
- GCP : GCP service account
- AWS : AWS IAM user/role account

- On-premises (non-Kubernetes) : user account, custom service account, 서비스 이름, Istio service account 또는 GCP service account. custom service account는 고객의 ID 디렉터리에서 관리하는 ID와 마찬가지로 기존 service account을 참조합니다.

Istio security vs SPIFFE

SPIFFE[1] 표준은 이기종 환경 전반에서 서비스에 대한 ID를 bootstrapping하고 발급할 수 있는 프레임워크에 대한 규격을 제공한다.

[1]"SPIFFE" : Secure Production Identity Framework for Everyone

Istio와 SPIFFE는 동일한 ID 문서를 공유합니다 : SVID (SPIFFE Verifiable Identity Document). 예를 들어 Kubernetes에서 X.509 인증서의 URI 필드는 spiffe://<domain>/ns/<namespace>/sa/<serviceaccount> 형식입니다. 이를 통해 Istio 서비스는 다른 SPIFFE 호환 시스템과의 연결을 설정하고 수락 할 수 있습니다.

SPIFFE의 구현인 Istio security와 SPIRE는 PKI 구현내역에 차이가 있다. Istio는 인증(authentication), 권한 부여(authorization) 및 감사(auditing) 등 보다 포괄적인 보안 솔루션을 제공합니다.

PKI

Istio PKI는 **Istio Citadel 위에 구축**되어 모든 작업 부하에 강력한 ID를 안전하게 제공합니다. Istio는 X.509 인증서를 사용하여 ID를 SPIFFE 형식으로 전달합니다. 또한 PKI는 규모에 따른 키 및 인증서 순환을 자동화합니다.

Istio는 Kubernetes pods와 on-premises 시스템에서 실행되는 서비스를 지원합니다. 현재 각 시나리오마다 서로 다른 인증서 Key 공급 메커니즘을 사용합니다.

Kubernetes scenario

1. Citadel은 Kubernetes apiserver를 감시하고 기존 및 새로운 서비스 계정 각각에 대해 SPIFFE 인증서와 Key pair를 생성합니다. Citadel은 Kubernetes의 secrets로 인증서와 키 쌍을 저장합니다.
2. Pod를 만들면 Kubernetes는 Kubernetes secret volume을 통해 Service Account에 따라 포드에 인증서 및 키 쌍을 마운트합니다.
3. Citadel은 각 인증서의 수명을 감시하고 Kubernetes secrets를 다시 작성하여 인증서를 자동으로 회전합니다.
4. Pilot은 특정 서비스를 실행할 수 있는 Service Account를 정의하는 secure naming information을 생성합니다. Pilot은 secure naming information을 sidecar Envoy에게 전달합니다.

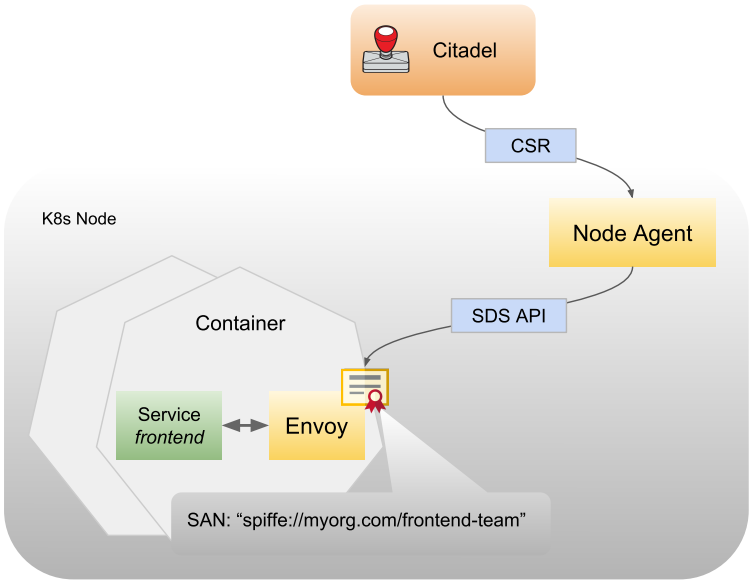
On-premises machines scenario

1. Citadel은 CSR (Certificate Signing Requests)을 취할 수있는 gRPC 서비스를 만듭니다.
2. 노드 에이전트는 개인 키와 CSR을 생성하고 자격 증명이있는 CSR을 Citadel에 보내 서명합니다.
3. Citadel은 CSR과 함께 전달 된 자격 증명의 유효성을 검사하고 CSR에 서명하여 인증서를 생성합니다.
4. 노드 에이전트는 Citadel에서받은 인증서와 개인 키를 Envoy로 보냅니다.
5. 위의 CSR 프로세스는 인증서 및 키 순환을 위해 주기적으로 반복됩니다.

Node agent in Kubernetes

Istio는 아래 그림과 같이 Kubernetes의 노드 에이전트를 인증서 및 키 프로비저닝에 사용하는 옵션을 제공합니다. 가까운 장래에 on-premises 시스템에 대한 ID 제공 플로우가 유사 할 것이라는 점을 유의하십시오. 여기서

Kubernetes 시나리오 만 설명합니다.



PKI with node agents in Kubernetes

흐름은 다음과 같이 진행됩니다.

1. Citadel은 CSR 요청을 처리하기 위해 gRPC 서비스를 만듭니다.
2. Envoy는 Envoy Secret Discovery Service (SDS) API를 통해 인증서와 키 요청을 보냅니다.
3. 노드 에이전트는 SDS 요청을 수신하면 자격 증명이있는 CSR을 Citadel에 보내 서명하기 전에 개인 키와 CSR을 생성합니다.
4. Citadel은 CSR에 포함 된 자격 증명의 유효성을 검사하고 CSR에 서명하여 인증서를 생성합니다.
5. 노드 에이전트는 Citadel에서받은 인증서와 개인 키를 Envoy SDS API를 통해 Envoy로 보냅니다.
6. 위의 CSR 프로세스는 인증서 및 키 순환을 위해 주기적으로 반복됩니다.

Best practices

In this section, we provide a few deployment guidelines and discuss a real-world scenario.

이 섹션에서는 몇 가지 배포 지침을 제공하고 실제 시나리오에 대해 설명합니다.

Deployment guidelines

중형(medium-size) 또는 대형(large-size) 클러스터에서 서로 다른 서비스를 배포하는 서비스 운영자 (a.k.a. SRE)가 여러 명인 경우 각 SRE 팀마다 별도의 Kubernetes 네임 스페이스를 만들어 액세스를 격리하는 것이 좋습니다. 예를 들어 team1에 대한 team1-ns 네임 스페이스와 team2에 대한 team2-ns 네임 스페이스를 만들 수 있으므로 두 팀이 서로의 서비스에 액세스 할 수 없습니다.

> Citadel이 손상된 경우 클러스터의 모든 관리 키와 인증서가 노출 될 수 있습니다. Citadel을 전용 네임 스페이스 (예 : istio-citadel-ns)에서 실행하여 클러스터에 대한 액세스를 관리자에게만 제한하는 것이 좋습니다.

Example

photo-frontend, photo-backend 및 datastore의 세 가지 서비스로 구성된 3 계층 응용 프로그램을 생각해 봅시다. Photo SRE 팀은 photo-frontend 및 photo-backend 서비스를 관리하고 데이터 저장소 SRE 팀은 데이터 저장소 서비스를 관리합니다. photo-frontend 서비스는 photo-backend에 액세스 할 수 있고 photo-backend 서비스는 데이터 저장소에 액세스 할 수 있습니다. 그러나 photo-frontend 서비스는 데이터 저장소에 액세스 할 수 없습니다.

이 시나리오에서 클러스터 관리자는 istio-citadel-ns, photo-ns 및 datastore-ns의 세 가지 이름 공간을 만듭니다. 관리자는 모든 네임 스페이스에 액세스 할 수 있으며 각 팀은 고유 한 네임 스페이스에만 액세스 할 수 있습니다. photo SRE 팀은 photo-ns 네임 스페이스에서 각각 photo-frontend 및 photo-backend를 실행하는 두 개의 서비스 계정을 만듭니다. 데이터 저장소 SRE 팀은 하나의 서비스 계정을 만들어 datastore-ns 네임 스페이스에서 데이터 저장소 서비스를 실행합니다.

또한 **Istio Mixer**에서 photo-frontend가 데이터 저장소에 액세스 할 수 없도록 서비스 액세스 제어를 적용해야 합니다.

이 설정에서 Kubernetes는 서비스 관리에 대한 운영자 권한을 분리 할 수 있습니다. Istio는 모든 네임 스페이스의 인증서와 키를 관리하고 서비스에 대한 다른 액세스 제어 규칙을 적용합니다.

Authentication

Istio는 두 가지 유형의 인증을 제공합니다.

- **Transport authentication** (**service-to-service** authentication 이라고도 함) : 직접 클라이언트가 연결을 확인합니다. Istio는 전송 인증을위한 전체 스택 솔루션으로서 **Mutual TLS**를 제공합니다. 서비스 코드를 변경하지 않고도이 기능을 쉽게 켤 수 있습니다. 이 솔루션 :
  - 각 서비스에 클러스터 및 클라우드 간의 상호 운용성을 가능하게하는 역할을 나타내는 강력한 신원 정보(Identity)를 제공합니다.
  - 서비스 간 통신 및 최종 사용자 간 통신을 보호합니다.
  - 키 및 인증서 생성, 배포 및 순환을 자동화하는 키 관리 시스템을 제공합니다.
- **End-user authentication**이라고도하는 **Origin authentication** : 최종 사용자 또는 장치로 요청한 원본 클라이언트를 확인합니다. Istio는 오픈 소스 OpenID Connect 공급자 인 ORY Hydra, Keycloak, Auth0, Firebase Auth, Google Auth 및 사용자 정의 인증에 대한 JSON Web Token (JWT) 유효성 검사 및 간소화된 개발자 경험을 통한 요청 수준 인증을 가능하게합니다.

두 경우 모두 Istio는 사용자 지정 Kubernetes API를 통해 Istio config store에 인증 정책을 저장합니다. Pilot은 적절한 경우 각 키와 함께 각 프록시에 대해 최신 정보를 유지합니다. 또한 Istio는 허용 모드에서 인증을 지원하여 정책 변경이 어떻게 적용되는지에 대한 이해를 돕습니다.

Mutual TLS authentication

Istio는 클라이언트 측과 서버 측 Envoy proxy를 통해 서비스 간 통신을 터널링합니다. 클라이언트가 Mutual TLS 인증(authentication)을 사용하여 서버를 호출하려면 다음을 수행하십시오.

1. Istio는 클라이언트의 Outbound 트래픽을 클라이언트의 로컬 Sidecar Envoy로 재 라우팅합니다.
2. 클라이언트 측 Envoy는 서버 측 Envoy와 Mutual TLS 핸드 셰이크를 시작합니다. 핸드 셰이크가 진행되는 동안 클라이언트 측 Envoy는 서버 인증서에 표시된 서비스 계정에 대상 서비스를 실행할 수있는 권한이 있는지 확인하기 위해 Secure naming 검사도 수행합니다.

3. 클라이언트 측 Envoy와 서버 측 Envoy가 상호 TLS 연결을 설정하고 Istio는 클라이언트 측 Envoy에서 서버 측 Envoy로 트래픽을 전달합니다.
4. 인증(authorization) 후 서버 측 Envoy는 로컬 TCP 연결을 통해 트래픽을 서버 서비스로 전달합니다.

Permissive mode

Istio 상호 TLS에는 허용 모드(permissive mode)가있어 서비스가 일반 텍스트 트래픽과 Mutual TLS 트래픽을 동시에 받아 들일 수 있습니다. 이 기능은 Mutual TLS onboarding 경험을 크게 향상시킵니다.

non-Istio 서버와 통신하는 많은 non-Istio 클라이언트는 상호 TLS가 활성화 된 Istio로 해당 서버를 마이그레이션하려는 운영자에게 문제점을 제시합니다. 일반적으로 운영자는 모든 클라이언트에 대해 Istio 사이드카를 동시에 설치할 수 없거나 일부 클라이언트에서 Istio 사이드카를 설치할 권한이 없습니다. 서버에 Istio 사이드카를 설치 한 후에도 운영자는 기존 통신을 중단하지 않고 상호 TLS를 사용할 수 없습니다.

허용 모드를 사용하면 서버는 일반 텍스트 및 상호 TLS 트래픽을 모두 허용합니다. 이 모드는 onboarding 절차에 큰 유연성을 제공합니다. 서버에 설치된 Istio Sidecar는 기존 일반 텍스트 트래픽을 손상시키지 않고 즉시 상호 TLS 트래픽을 처리합니다. 결과적으로 운영자는 점차 클라이언트의 Istio Sidecar를 설치 및 구성하여 상호 TLS 트래픽을 전송할 수 있습니다. 클라이언트의 구성이 완료되면 운영자는 서버를 상호 TLS 전용 모드로 구성할 수 있습니다. 자세한 내용은 상호 TLS 마이그레이션 자습서를 참조하십시오.

Secure naming

Secure naming 정보에는 인증서로 인코딩 된 서버 ID의 검색 서비스 또는 DNS에서 참조하는 서비스 이름에 대한 N 대 N 매핑이 포함됩니다. ID A에서 서비스 이름 B 로의 맵핑은 "A가 허용되고 서비스 B를 실행할 수있는 권한이 있음"을 의미합니다. Pilot은 Kubernetes apiserver를 감시하고 Secure naming 정보를 생성하며 안전하게 Sidecar Envoys에 배포합니다. 다음 예는 인증(authentication)에서 Secure naming이 중요한 이유를 설명합니다.

서비스 데이터 저장소를 실행하는 합법적인(legitimate) 서버가 인프라 팀 ID 만 사용한다고 가정합니다. 악의적인(malicious) 사용자는 테스트 팀 ID에 대한 인증서와 키를 가지고 있습니다. 악의적인 사용자는 클라이언트에서 보낸 데이터를 검사하기 위해 서비스를 가장하려고합니다. 악의적 인 사용자는 테스트 팀 ID에 대한 인증서와 키가있는 위조 된 서버를 배포합니다. 악의적 인 사용자가 데이터 저장소로 전송 된 트래픽을 성공적으로 hijacked (through DNS spoofing, BGP/route hijacking, ARP spoofing, etc.)하여 위조 된 서버로 리디렉션했다고 가정합니다.

클라이언트가 데이터 저장소 서비스를 호출하면 서버 인증서에서 테스트 팀 ID를 추출하고 테스트 팀이 보안 명명 정보로 데이터 저장소를 실행할 수 있는지 확인합니다. 클라이언트는 테스트 팀이 데이터 저장소 서비스를 실행할 수 없으며 인증이 실패 함을 감지합니다.

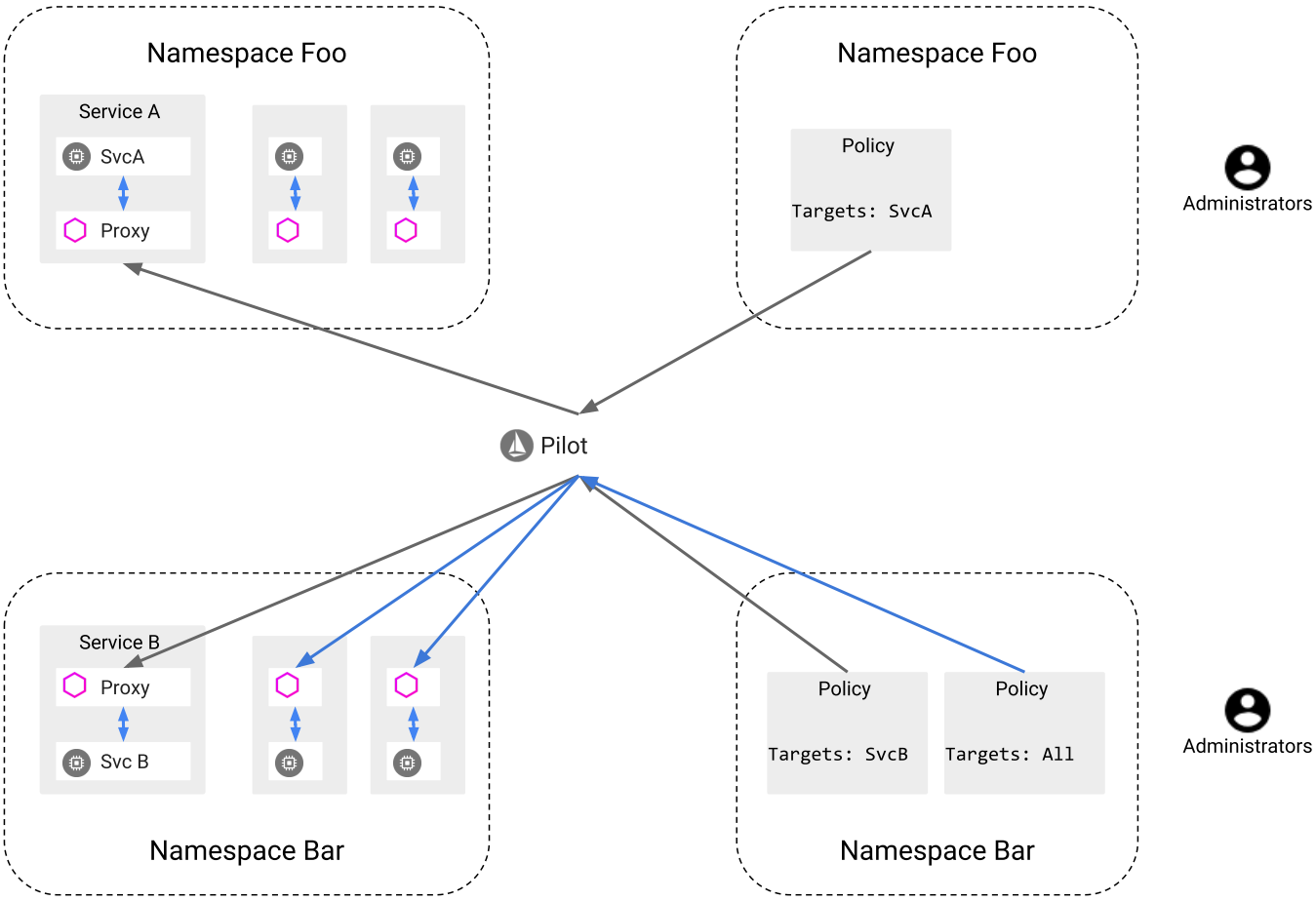
Secure naming은 HTTPS 트래픽에 대한 일반적인 network hijacking을 방지 할 수 있습니다. 또한 DNS spoofing을 제외한 일반적인 network hijacking에서 TCP 트래픽을 보호 할 수 있습니다. 공격자가 DNS를 가로채고 목적지의 IP 주소를 변경하면 TCP 트래픽에 대해 작동하지 않습니다. 이는 TCP 트래픽에 호스트 이름 정보가 포함되어 있지 않기 때문에 라우팅을 위해 IP 주소에만 의존 할 수 있기 때문입니다. 그리고이 DNS 도용(hijack)은 클라이언트 쪽 Envoy가 트래픽을 받기 전에도 발생할 수 있습니다.

Authentication architecture

인증 정책을 사용하여 Istio 메시에서 요청을 수신하는 서비스에 대한 인증 요구 사항을 지정할 수 있습니다. 메쉬 운영자는 .yaml 파일을 사용하여 정책을 지정합니다. 정책은 일단 배포되면 Istio 구성 저장소에 저장됩니다.

Istio 컨트롤러 인 Pilot은 구성 저장 장치를 감시합니다. 정책이 변경되면 Pilot은 새 정책을 적절한 구성으로 변환하여 Envoy Sidecar Proxy에 필요한 인증 메커니즘을 수행하는 방법을 알려줍니다. Pilot은 공개 키를 가져 와서 JWT 유효성 검사를위한 구성에 첨부 할 수 있습니다. 또는 Pilot은 Istio 시스템이 관리하는 키와 인증서에 대한 경로를 제공하고 이를 상호 TLS 용 Application pod에 설치합니다. 자세한 내용은 PKI 섹션을 참조하십시오. Istio는 대상 엔드 포인트에 구성을 비동기적으로 전송합니다. 프록시가 구성을 수신하면 새 인증 요구 사항이 해당 Pod에서 즉시 적용됩니다.

요청을 보내는 클라이언트 서비스는 필요한 인증 메커니즘을 수행 할 책임이 있습니다. origin authentication (JWT)의 경우, 애플리케이션은 JWT credential을 획득하여 요청에 첨부해야합니다. 상호 TLS(mutual TLS)의 경우 Istio는 대상 규칙(destination rule)을 제공합니다. 운영자는 destination rule을 사용하여 클라이언트 프록시가 서버 측에서 예상되는 인증서로 TLS를 사용하여 초기 연결을하도록 지시 할 수 있습니다. Mutual TLS authentication에서 Istio에서 상호 TLS가 작동하는 방법에 대해 자세히 알아볼 수 있습니다.



Authentication Architecture

Istio는 두 가지 유형의 인증뿐만 아니라 자격 증명의 다른 클레임을 사용하여 다음 계층에 ID를 출력합니다. 또한 운영자는 전송(transport) 또는 원본 인증(origin authentication)에서 Istio가 '주체(principal)'로 사용할 ID를 지정할 수 있습니다.

Authentication policies

이 섹션에서는 Istio 인증 정책(Istio authentication policies)의 작동 방식에 대해 자세히 설명합니다. 아키텍처 섹션에서 기억 하듯이 인증 정책(authentication policies)은 서비스가 **받는** 요청에 적용됩니다. 상호 TLS에서 클라이언트 측 인증 규칙(authentication policies)을 지정하려면 DestinationRule에 TLSSettings를 지정해야합니다. 자세한 내용은 TLS 설정 참조 설명서를 참조하십시오. 다른 Istio 구성과 마찬가지로 .yaml 파일에 인증 정책을 지정할 수 있습니다. kubectl을 사용하여 정책을 배포합니다.

다음 예제 인증 정책은 검토 서비스에 대한 전송 인증(transport authentication)이 상호 TLS를 사용해야 함을 지정합니다.

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
    name: "reviews"
spec:
    targets:
    - name: reviews
    peers:
    - mtls: {}
```

Policy storage scope

Istio는 namespace-scope 또는 mesh-scope storage에 인증 정책(authentication policies)을 저장할 수 있습니다.

- Mesh-scope policy는 kind 필드에 "MeshPolicy"값을, "default"라는 이름으로 지정됩니다. For example:

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "MeshPolicy"
metadata:
    name: "default"
spec:
    peers:
    - mtls: {}
```

- Namespace-scope 정책은 kind 필드와 지정된 네임 스페이스에 대해 "Policy"값으로 지정됩니다. 지정하지 않으면 기본 네임 스페이스가 사용됩니다. 네임 스페이스 ns1의 예를 들면 다음과 같습니다.

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "default"
  namespace: "ns1"
spec:
  peers:
  - mtls: {}
```

namespace-scope storage의 정책은 동일한 네임 스페이스의 서비스에만 영향을 미칩니다. mesh-scope의 정책은 Mesh의 모든 서비스에 영향을 줄 수 있습니다. 충돌과 오용을 방지하기 위해 mesh-scope storage에 하나의 정책 만 정의 할 수 있습니다. 이 정책은 default라는 이름을 가져야하며 empty targets: 섹션이 있어야합니다. Google의 target selectors 섹션에 대한 자세한 내용을 확인할 수 있습니다.

Kubernetes는 현재 Custom Resource Definitions (CRDs)에 Istio 구성을 구현합니다. 이러한 CRD는 namespace-scope 및 cluster-scope CRDs에 해당하며 Kubernetes RBAC를 통해 액세스 보호를 자동으로 상속합니다. Kubernetes CRD 문서에 대한 자세한 내용을 볼 수 있습니다.

Target selectors

인증 정책(authentication policy)의 대상은 정책이 적용되는 서비스를 지정합니다. 다음 예제는 정책을 적용 할 대상을 지정하는 targets : 섹션을 보여줍니다.

- 모든 Port의 product-page service.
- Port 9000의 리뷰 서비스.

```
targets:
 - name: product-page
 - name: reviews
   ports:
   - number: 9000
```

targets : 섹션을 제공하지 않으면 Istio는 정책의 스토리지 범위에있는 모든 서비스에 정책을 일치시킵니다. 따라서 targets : 섹션은 정책의 범위를 지정하는 데 도움을 줄 수 있습니다.

- Mesh-wide policy : target selector 섹션이 없는 mesh-scope storage에 정의 된 정책입니다. **Mesh에는** 최대한 **하나**의 mesh-wide 정책이 있을 수 있습니다.
- Namespace-wide policy : 이름이 default이며 target selector 섹션이 없는 namespace-scope storage에 정의 된 정책입니다. **네임스페이스 당** namespace-wide 정책이 최대한 **하나** 있을 수 있습니다.
- Service-specific policy : 비어 있지 않은 target selector 섹션을 사용하여 namespace-scope storage에 정의 된 정책입니다. 네임 스페이스는 **0 개, 1 개 또는 여러 개**의 서비스 별 정책을 가질 수 있습니다.

각 서비스에 대해 Istio는 narrowest matching 정책을 적용합니다. 순서는 **service-specific > namespace-wide > mesh-wide**입니다. 둘 이상의 서비스 특정 정책이 서비스와 일치하는 경우 Istio는 임의로 서비스 중 하나를 선택합니다. 운영자는 정책을 구성 할 때 이러한 충돌을 피해야합니다.

mesh-wide 및 namespace-wide 정책에 고유성을 적용하기 위해 Istio는 Mesh 당 하나의 인증 정책(authentication policy)과 네임 스페이스 당 하나의 인증 정책 만 허용합니다. 또한 Istio는 mesh-wide 및 namespace-wide 정책에서 특정 이름을 default으로 지정해야합니다.

Transport authentication

The following example shows the peers: section enabling transport authentication using mutual TLS.

peers: 섹션은 정책에서 전송 인증(transport authentication)을 위해 지원되는 인증(authentication) 방법 및 관련 매개 변수를 정의합니다. 섹션에는 둘 이상의 메소드를 나열 할 수 있으며 인증(authentication)을 통과하기

위해 하나의 메소드 만 충족시켜야합니다. **그러나 Istio 0.7 릴리스부터 현재 지원되는 전송 인증 (transport authentication) 방법은 상호 TLS (Mutual TLS)뿐입니다.** 전송 인증이 필요하지 않은 경우이 절을 완전히 건너 뛰세요.

다음 예는 상호 TLS를 사용하여 전송 인증을 사용하는 peers: 섹션을 보여줍니다.

```
peers:
  - mtls: {}
```

Currently, the mutual TLS setting doesn't require any parameters. Hence, -mtls: {}, - mtls: or - mtls: null declarations are treated the same. In the future, the mutual TLS setting may carry arguments to provide different mutual TLS implementations.

현재 상호 TLS 설정에는 parameters가 필요하지 않습니다. 따라서 -mtls : {}, - mtls : 또는 - mtls : null 선언은 동일하게 취급됩니다. 향후에는 상호 TLS 설정은 서로 다른 TLS 구현을 제공하기 위해 arguments를 수행 할 수 있습니다.

Origin authentication

The origins: 섹션은 origin authentication을 위해 지원되는 인증 메소드 및 관련 Parameters를 정의합니다. Istio는 JWT origin authentication 만 지원합니다. 허용 된 JWT 발급자를 지정하고 특정 경로에 대해 JWT authentication을 활성화 또는 비활성화 할 수 있습니다. 요청 경로에 대해 모든 JWT가 비활성화 된 경우 인증이 정의되지 않은 것처럼 전달됩니다. peer authentication과 마찬가지로, 나열된 메소드 중 하나만 충족시켜 인증을 통과해야합니다.

다음 예제 정책은 Google에서 발행 한 JWT를 허용하는 origin authentication 위한 originins: section을 지정합니다. 경로 /health에 대한 JWT 인증이 비활성화됩니다.

```
origins:
- jwt:
    issuer: "https://accounts.google.com"
    jwksUri: "https://www.googleapis.com/oauth2/v3/certs"
    trigger_rules:
    - excluded_paths:
      - exact: /health
```

Principal binding

주요 바인딩 key-value 쌍은 정책의 주요 인증(principal authentication)을 정의합니다. 기본적으로 Istio는 peers: 섹션에 구성된 인증을 사용합니다. peers: 섹션에 인증이 구성되어 있지 않으면 Istio는 인증을 설정 해제합니다. 정책 작성자는 이 동작으로 USE_ORIGIN 값을 덮어 쓸 수 있습니다. 이 값은 대신 Istio가 주체 인증(principal authentication)으로 원본 인증(origin's authentication)을 사용하도록 구성합니다. 나중에 조건부 바인딩을 지원합니다 (예 : Peer가 X 일 때 USE_PEER, 그렇지 않으면 USE_ORIGIN).

다음 예는 값이 USE_ORIGIN 인 principalBinding Key를 보여줍니다.

```
principalBinding: USE_ORIGIN
```

Updating authentication policies

언제든지 인증 정책을 변경할 수 있으며 **Istio는 거의 실시간으로 변경 사항을 엔드 포인트에 적용**합니다. 그러나 Istio는 **모든 엔드 포인트가 동시에 새 정책을 수신하도록 보장 할 수 없습니다**. 다음은 인증 정책을 업데이트 할 때 방해를 피하기 위한 권장 사항입니다.

- To enable or disable mutual TLS : mode: key 및 PERMISSIVE value와 함께 임시 policy를 사용하십시오. 두 가지 유형의 트래픽 (일반 텍스트 및 TLS)을 허용하도록 수신 서비스를 구성합니다. 따라서 요청이 삭제되지 않습니다. 모든 클라이언트가 상호 TLS의 유무에 관계없이 예상 프로토콜로 전환하면 PERMISSIVE 정책을 최종 정책으로 바꿀 수 있습니다. 자세한 내용은 상호 TLS 마이그레이션 자습서를 참조하십시오.

```
peers:
- mtls:
    mode: PERMISSIVE
```

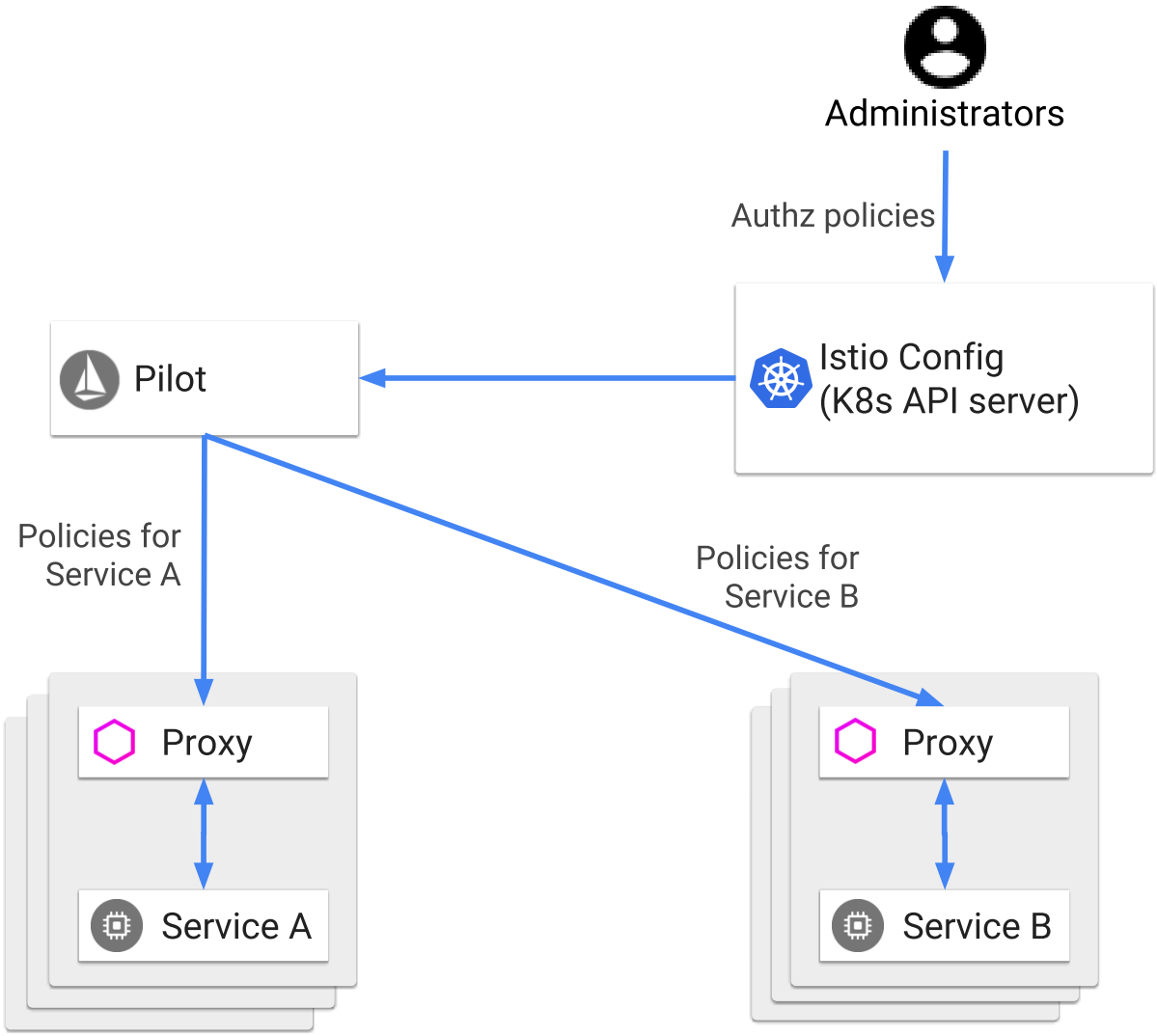
- For JWT authentication migration: 정책을 변경하기 전에 요청에 새 JWT가 있어야합니다. 서버 측에서 새 정책으로 완전히 전환하면 이전 JWT가 있을 경우 제거 할 수 있습니다. 이러한 변경 사항을 적용하려면 클라이언트 응용 프로그램을 변경해야합니다.

Authorization

Role-based Access Control (RBAC)이라고 하는 Istio의 권한 부여 (Istio's authorization) 기능은 Istio Mesh에서 서비스에 대한 namespace-level, service-level, and method-level 액세스 제어를 제공합니다. 특징 :

- 간단하고 사용하기 쉬운 역할 기반 의미론 (**Role-Based semantics**).
- **Service-to-service** and **end-user-to-service** authorization.
- **Flexibility through custom properties** support, for example conditions, in roles and role-bindings.
- Istio 인증이 Envoy에 기본적으로 적용되므로 **High performance**.
- 호환성이 뛰어나고 (**High compatibility**) HTTP, HTTPS 및 HTTP2를 기본적으로 지원하며 일반 TCP 프로토콜도 지원합니다.

Authorization architecture



Istio Authorization Architecture

위 다이어그램은 기본 Istio authorization architecture를 보여줍니다. 운영자는 .yaml 파일을 사용하여 Istio 인증 정책을 지정합니다. 일단 배포되면 Istio는 Istio Config Store에 정책을 저장합니다.

Pilot은 Istio authorization 정책의 변경을 감시합니다. 변경 사항이있는 경우 updated authorization policy를 fetch합니다. Pilot은 Istio authorization 정책을 서비스 인스턴스와 함께 배치 된 (co-located with the service instances) Envoy proxies에 배포(Distribute)합니다.

각 Envoy proxy는 런타임시 Request 권한 부여 하는 것을 authorization engine에서 실행합니다. 요청이 프록시에 도달하면 authorization engine은 현재 권한 policy에 대해 요청 컨텍스트를 평가하고 권한 결과 인 ALLOW 또는 DENY를 리턴합니다.

Enabling authorization

ClusterRbacConfig Object를 사용하여 Istio Authorization을 활성화합니다. ClusterRbacConfig Object는 고정된 이름 값 default를 가지는 cluster-scoped 단일 개체(singleton) 입니다. Mesh에서 하나의 ClusterRbacConfig 인스턴스 만 사용할 수 있습니다. 다른 Istio 구성 객체와 마찬가지로 ClusterRbacConfig는 Kubernetes CustomResourceDefinition (CRD) 객체로 정의됩니다.

ClusterRbacConfig 객체에서 연산자는 다음과 같은 mode 값을 지정할 수 있습니다.

- OFF : Istio 인증이 비활성화됩니다.
- ON : 메쉬의 모든 서비스에 대해 Istio 권한이 활성화됩니다.

- ON_WITH_INCLUSION : Istio 인증은 포함 필드 (inclusion field)에 지정된 서비스 및 네임 스페이스에만 사용할 수 있습니다.
- ON_WITH_EXCLUSION : Istio 인증은 제외 필드 (exclusion/span> field)에 지정된 서비스와 네임 스페이스를 제외하고 메쉬의 모든 서비스에 대해 활성화됩니다.

다음 예에서 기본 네임 스페이스에 대해 Istio authorization이 사용됩니다.

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ClusterRbacConfig
metadata:
  name: default
spec:
  mode: 'ON_WITH_INCLUSION'
  inclusion:
    namespaces: ["default"]
```

Authorization policy

To configure an Istio authorization policy, you specify a ServiceRole and ServiceRoleBinding. Like other Istio configuration objects, they are defined as Kubernetes CustomResourceDefinition (CRD) objects.

Istio authorization policy를 구성하려면 ServiceRole과 ServiceRoleBinding을 지정하십시오. 다른 Istio 구성 객체와 마찬가지로, 이들은 Kubernetes CustomResourceDefinition (CRD) 객체로 정의됩니다.

- ServiceRole은 서비스에 액세스하기위한 권한 그룹을 정의합니다.
- ServiceRoleBinding은 사용자, 그룹 또는 서비스와 같은 특정 주제에 ServiceRole을 부여합니다.

The combination of ServiceRole and ServiceRoleBinding specifies: who is allowed to do what under which conditions. Specifically:

- who refers to the subjects section in ServiceRoleBinding.
- what refers to the permissions section in ServiceRole.
- which conditions refers to the conditions section you can specify with the Istio attributes in either ServiceRole or ServiceRoleBinding.

ServiceRole과 ServiceRoleBinding의 조합은 **누가(Who) 어떤 조건(Which Condition)** 에서 **무엇(What)**을 할 수 있는지를 지정합니다. 구체적으로 :

- ServiceRoleBinding의 subjects 섹션을 참조하는 **사용자(Who)**.
- ServiceRole의 permissions 섹션을 참조하는 **무엇(What)**
- 어떤 조건은 ServiceRole 또는 ServiceRoleBinding에서 Istio 속성으로 지정할 수있는 conditions 섹션을 참조하는 **어느 조건 (Which Condition)**

ServiceRole

ServiceRole 사양에는 list of rules, AKA permissions 이 포함됩니다. 각 규칙에는 다음과 같은 표준 필드가 있습니다.

- services : 서비스 이름 목록. 값을 *로 설정하여 지정된 네임 스페이스의 모든 서비스를 포함 할 수 있습니다.
- methods : HTTP 메소드 이름 목록, gRPC 요청에 대한 사용 권한의 경우 HTTP 동사는 항상 POST입니다. 값을 *로 설정하여 모든 HTTP 메소드를 포함 할 수 있습니다.
- Path : HTTP 경로 또는 gRPC 메소드. gRPC 메소드는 /packageName.serviceName/methodName 형식이어야하며 대소 문자를 구분합니다.

A ServiceRole specification only applies to the namespace specified in the metadata section. A rule requires the services field and the other fields are optional. If you do not specify a field or if you set its value to *, Istio applies the field to all instances.

ServiceRole 사양은 metadata 섹션에 지정된 네임 스페이스에만 적용됩니다. 규칙에는 service 필드가 필요하고 다른 필드는 선택 사항입니다. 필드를 지정하지 않거나 값을 *로 설정하면 Istio는 해당 필드를 모든 인스턴스에 적용합니다.

아래 예제는 간단한 역할을 보여줍니다. service-admin은 default 네임 스페이스의 모든 서비스에 대한 전체 액세스 권한을가집니다.

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: service-admin
  namespace: default
spec:
  rules:
  - services: ["*"]
```

Here is another role: products-viewer, which has read, "GET" and "HEAD", access to the service products.default.svc.cluster.local in the default namespace.

여기에는 또 다른 역할이 있습니다. "GET" 및 "HEAD"를 읽은 products-viewer는 default 네임 스페이스의 products.default.svc.cluster.local 서비스에 액세스합니다.

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: products-viewer
  namespace: default
spec:
  rules:
  - services: ["products.default.svc.cluster.local"]
    methods: ["GET", "HEAD"]
```

또한 규칙의 모든 필드에 대해 접두어 일치 및 접미사 일치를 지원합니다. 예를 들어, default 네임 스페이스에서 다음 권한으로 tester 역할을 정의 할 수 있습니다.

- 접두사 "test-*" 가 있는 모든 서비스에 대한 전체 액세스 (예 : test-bookstore, test-performance, test-api.default.svc.cluster.local).
- "*/reviews" 접미어가 붙은 모든 경로 (예 : bookstore.default.svc.cluster.local 서비스의 / books / reviews, / events / booksale / reviews, / reviews)에 대한 읽기 ("GET") 액세스.

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: tester
  namespace: default
spec:
  rules:
  - services: ["test-*"]
    methods: ["*"]
  - services: ["bookstore.default.svc.cluster.local"]
    paths: ["*/reviews"]
    methods: ["GET"]
```

ServiceRole에서 namespace + services + paths + methods 의 조합은 **service 또는 services에 액세스하는 방법**을 정의합니다. 경우에 따라 규칙에 대한 추가 조건을 지정해야 할 수도 있습니다. 예를 들어 규칙은 특정 **Version**의 서비스에만 적용되거나 "foo"와 같은 특정 **Label**이 있는 서비스에만 적용될 수 있습니다. constraints을 사용하여 이러한 조건을 쉽게 지정할 수 있습니다.

예를 들어 다음 ServiceRole 정의는 request.headers [version]이 이전 products-viewer 역할을 확장하는 "v1"또는 "v2"중 하나라는 제약 조건을 추가합니다. 제약 조건의 지원되는 키 값은 constraints and properties page에 나열됩니다. 속성이 map 인 경우 (예 : request.headers), Key는 맵의 항목입니다 (예 : request.headers [version]).

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: products-viewer-version
  namespace: default
spec:
  rules:
  - services: ["products.default.svc.cluster.local"]
    methods: ["GET", "HEAD"]
    constraints:
    - key: request.headers[version]
      values: ["v1", "v2"]
```

ServiceRoleBinding

ServiceRoleBinding 사양에는 다음 두 부분이 포함됩니다.

- roleRef는 동일한 네임 스페이스의 ServiceRole 리소스를 참조합니다.
- 역할에 할당 된 subjects 의 목록.

user 또는 properties 집합을 사용하여 제목을 명시적으로 지정할 수 있습니다. ServiceRoleBinding subject의 특성은 ServiceRole 스펙의 제한 조건과 유사합니다. 또한 속성을 사용하면 조건을 사용하여이 역할에 할당 된 일련의 계정을 지정할 수 있습니다. 여기에는 Key와 허용되는 값이 포함됩니다. 제약 조건의 지원되는 Key 값은 제약 조건 및 속성 페이지에 나열됩니다.

The following example shows a ServiceRoleBinding named test-binding-products, which binds two subjects to the ServiceRole named "product-viewer" and has the following subjects

다음 예제에서는 test-binding-products라는 ServiceRoleBinding을 보여줍니다. ServiceRoleBinding은 두 개의 제목을 "product-viewer"라는 ServiceRole에 바인딩하며 다음과 같은 subjects가 있습니다

- 서비스 a를 나타내는 서비스 계정, "service-account-a".
- Ingress 서비스 "istio-ingress-service-account"를 나타내는 서비스 계정과 JWT 전자 메일 클레임이 "a@foo.com"인 서비스 계정.

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: test-binding-products
  namespace: default
spec:
  subjects:
  - user: "service-account-a"
  - user: "istio-ingress-service-account"
    properties:
      request.auth.claims[email]: "a@foo.com"
  roleRef:
    kind: ServiceRole
    name: "products-viewer"
```

In case you want to make a service publicly accessible, you can set the subject to user: "*". This value assigns the ServiceRole to all (both authenticated and unauthenticated) users and services, for example:

서비스에 공개적으로 액세스 할 수 있게 하려는 경우 subject를 user:"*"로 설정할 수 있습니다. 이 값은 ServiceRole을 **모든 (인증 된 사용자와 인증되지 않은)** 사용자 및 서비스에 할당합니다. 예를 들면 다음과 같습니다.

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: binding-products-allusers
```

```
    namespace: default
spec:
  subjects:
  - user: "*"
  roleRef:
    kind: ServiceRole
    name: "products-viewer"
```

인증 된 사용자와 서비스에만 ServiceRole을 할당하려면 대신 source.principal:"*"을 사용하십시오. 예를 들면 다음과 같습니다.

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: binding-products-all-authenticated-users
  namespace: default
spec:
  subjects:
  - properties:
      source.principal: "*"
  roleRef:
    kind: ServiceRole
    name: "products-viewer"
```

Using Istio authorization on plain TCP protocols

Service role 및 Service role binding의 예는 HTTP 프로토콜을 사용하는 서비스에서 Istio 인증을 사용하는 일반적인 방법을 보여줍니다. 이 예에서는 서비스 역할 및 서비스 역할 바인딩의 모든 필드가 지원됩니다.

Istio 인증은 MongoDB와 같은 일반 TCP 프로토콜을 사용하는 서비스를 지원합니다. 이 경우 HTTP 서비스와 동일한 방식으로 서비스 역할 및 서비스 역할 바인딩을 구성합니다. 차이점은 특정 필드, 제약 조건 및 속성은 HTTP 서비스에만 적용된다는 것입니다. 이 필드에는 다음이 포함됩니다.

- 서비스 역할 구성 객체의 paths 및 methods 필드
- 서비스 역할 바인딩 구성 객체의 group 필드입니다.

지원되는 제한 조건 및 특성은 constraints and properties page에 나열됩니다.

TCP 서비스에 대해 HTTP 전용 필드를 사용하는 경우 Istio는 서비스 역할 또는 서비스 역할 바인딩 사용자 지정 리소스와 완전히 설정된 정책을 무시합니다.

다음 예는 Istio 메쉬의 bookinfo-ratings-v2가 MongoDB 서비스에 액세스 할 수 있도록 허용하는 서비스 역할과 서비스 역할 바인딩을 27017 포트에 MongoDB 서비스가 있다고 가정합니다.

```
apiVersion: "rbac.istio.io/v1alpha1"
```

```
    kind: ServiceRole
    metadata:
      name: mongodb-viewer
      namespace: default
    spec:
      rules:
      - services: ["mongodb.default.svc.cluster.local"]
        constraints:
        - key: "destination.port"
          values: ["27017"]
    ---
    apiVersion: "rbac.istio.io/v1alpha1"
    kind: ServiceRoleBinding
    metadata:
      name: bind-mongodb-viewer
      namespace: default
    spec:
      subjects:
      - user: "cluster.local/ns/default/sa/bookinfo-ratings-v2"
      roleRef:
        kind: ServiceRole
        name: "mongodb-viewer"
```

Authorization permissive mode

허가 허용 모드(authorization permissive mode)는 Istio 1.1 릴리스의 실험적 기능입니다. 인터페이스는 향후 릴리스에서 변경 될 수 있습니다.

허가 허용 모드(authorization permissive mode)에서는 프로덕션 환경에서 적용하기 전에 권한 policy를 검증 할 수 있습니다.

전역 권한 부여 구성(global authorization configuration) 및 개별 정책(individual policies)에서 authorization permissive 모드를 사용 가능하게 할 수 있습니다. 전역 권한 부여 구성에서 허용 모드를 설정하면 모든 정책이 자체 설정 모드(their own set mode)와 상관없이 허용 모드로 전환됩니다. 전역 권한 모드를 ENFORCED 설정하면 개별 정책에 의해 설정된 적용 모드가 적용됩니다. 모드를 설정하지 않으면, 전역 권한 부여 구성과 개별 정책이 모두 기본적으로 ENFORCED 모드로 설정됩니다.

To enable the permissive mode globally, set the value of the enforcement_mode: key in the global Istio RBAC authorization configuration to PERMISSIVE as shown in the following example.

허용 모드를 전역적으로 사용하려면 global Istio RBAC authorization configuration의 enforcement_mode: 키 값을 PERMISSIVE 설정하십시오 (다음 예 참조).

```
    apiVersion: "rbac.istio.io/v1alpha1"
    kind: ClusterRbacConfig
    metadata:
      name: default
    spec:
      mode: 'ON_WITH_INCLUSION'
```

```
    inclusion:
      namespaces: ["default"]
    enforcement_mode: PERMISSIVE
```

특정 정책에 대해 허용 모드를 사용하려면 다음 예와 같이 정책 구성 파일에서 mode: 키의 값을 PERMISSIVE로 설정하십시오.

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: bind-details-reviews
  namespace: default
spec:
  subjects:
    - user: "cluster.local/ns/default/sa/bookinfo-productpage"
  roleRef:
    kind: ServiceRole
    name: "details-reviews-viewer"
  mode: PERMISSIVE
```

Using other authorization mechanisms

Istio 인증 메커니즘 사용을 강력하게 권장하지만 Istio는 Mixer 구성 요소를 통해 자체 인증 및 권한 부여 메커니즘을 연결할 수 있도록 충분히 유연합니다. Mixer에서 플러그인을 사용하고 구성하려면 policies and telemetry adapters docs를 방문하십시오.

Task

Authentication Policy

이 작업에서는 Istio 인증 정책을 활성화, 구성 및 사용할 때 수행해야하는 주요 활동에 대해 설명합니다. 인증(authentication) 개요의 기본 개념에 대해 자세히 알아보십시오.

Before you begin

Istio 인증 정책(authentication policy) 및 관련 상호 TLS 인증(mutual TLS authentication) 개념을 이해합니다.

Global mutual TLS를 사용하지 않고 Kubernetes 클러스터를 Istio와 함께 설치합니다.

> Installation steps에서 설명한대로 install/kubernetes /istio-demo.yaml을 사용하거나 Helm을 사용하여 global.mtls.enabled를 false로 설정

Setup

우리의 예제는 두 개의 네임 스페이스 foo와 bar를 사용하는데 httpbin과 sleep이라는 두 개의 서비스가 Envoy Sidecar Proxy로 실행됩니다. 또한 legacy 네임 스페이스에서 Sidecar 없이 실행되는 httpbin 및 sleep의 두 번째 인스턴스를 사용합니다. 작업을 시도 할 때 동일한 예제를 사용하려면 다음을 실행하십시오.

```
    $ kubectl create ns foo
    $ kubectl apply -f <(istioctl kube-inject -f samples/httpbin/httpbin.yaml) -n
foo
    $ kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.yaml) -n foo
    $ kubectl create ns bar
    $ kubectl apply -f <(istioctl kube-inject -f samples/httpbin/httpbin.yaml) -n
bar
    $ kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.yaml) -n bar
    $ kubectl create ns legacy
    $ kubectl apply -f samples/httpbin/httpbin.yaml -n legacy
    $ kubectl apply -f samples/sleep/sleep.yaml -n legacy
```

네임 스페이스 foo, bar 또는 legacy의 모든 sleep pod에서 curl으로 HTTP 요청을 httpbin.foo, httpbin.bar 또는 httpbin.legacy로 보내 설정을 확인할 수 있습니다. 모든 요청은 HTTP 코드 200에서 성공해야 합니다.

예를 들어 sleep.bar를 httpbin.foo 접근성으로 확인하는 명령은 다음과 같습니다.

```
    $ kubectl exec $(kubectl get pod -l app=sleep -n bar -o jsonpath=
{.items..metadata.name}) -c sleep -n bar -- curl http://httpbin.foo:8000/ip -s -o
/dev/null -w "%{http_code}\n"
    200
```

아래의 명령은 모든 접근 가능한 조합을 편리하게 반복합니다.

```
    $ for from in "foo" "bar" "legacy"; do for to in "foo" "bar" "legacy"; do
kubectl exec $(kubectl get pod -l app=sleep -n ${from} -o jsonpath=
{.items..metadata.name}) -c sleep -n ${from} -- curl
"http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from} to httpbin.${to}:
%{http_code}\n"; done; done
    sleep.foo to httpbin.foo: 200
    sleep.foo to httpbin.bar: 200
    sleep.foo to httpbin.legacy: 200
    sleep.bar to httpbin.foo: 200
    sleep.bar to httpbin.bar: 200
```

```
sleep.bar to httpbin.legacy: 200
sleep.legacy to httpbin.foo: 200
sleep.legacy to httpbin.bar: 200
sleep.legacy to httpbin.legacy: 200
```

또한 다음과 같이 시스템에 기본 메쉬 인증 정책(default mesh authentication policy)이 있는지 확인해야합니다.

```
$ kubectl get policies.authentication.istio.io --all-namespaces
No resources found.
```

```
$ kubectl get meshpolicies.authentication.istio.io
NAME      AGE
default   3m
```

마지막으로, 예제 서비스에 적용되는 대상 규칙(destination rules)이 없는지 확인하세요. 기존 대상 규칙(destination rules)의 host: value를 검사하고 일치되지 않는 것이 확인되면 작업을 수행 할 수 있습니다.

예를들면 :

```
$ kubectl get destinationrules.networking.istio.io --all-namespaces -o yaml |
grep "host:"
        host: istio-policy.istio-system.svc.cluster.local
        host: istio-telemetry.istio-system.svc.cluster.local
```

> Istio의 버전에 따라 표시된 호스트 이외의 호스트에 대한 대상 규칙이 표시 될 수 있습니다. 그러나 foo, bar 및 legacy 네임 스페이스의 호스트에는 아무 것도 없어야하며 모두 일치인 와일드 카드 *도 없어야 합니다.

Globally enabling Istio mutual TLS

상호 TLS(mutual TLS)를 가능하게하는 메쉬 전체 인증 정책(mesh-wide authentication policy)을 설정하려면 다음과 같이 Mesh 인증 정책(mesh authentication policy)을 제출하십시오.

```
$ kubectl apply -f - <<EOF
apiVersion: "authentication.istio.io/v1alpha1"
kind: "MeshPolicy"
```

```
    metadata:
      name: "default"
    spec:
      peers:
      - mtls: {}
    EOF
```

> Mesh 인증 정책은 클러스터 범위(cluster-scoped)의 MeshPolicy CRD에 정의 된 일반 인증 정책 API를 사용합니다.

이 정책은 Mesh의 모든 작업 부하가 TLS를 사용하여 암호화 된 요청만 수락하도록 지정합니다. 보시다시피 이 인증 정책에는 kind: MeshPolicy가 있습니다. 정책의 이름은 default 이며 targets 지정이 없습니다 (Mesh의 모든 서비스에 적용하기 위한 것임).

이 시점에서 수신 측만 상호 TLS를 사용하도록 구성됩니다. Istio 서비스 (Sidecar가 있는 서비스)간에 curl 명령을 실행하면 클라이언트 측에서 여전히 일반 텍스트를 사용하므로 503 오류 코드로 모든 요청이 실패합니다.

```
    $ for from in "foo" "bar"; do for to in "foo" "bar"; do kubectl exec $(kubectl
get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name}) -c sleep -n
${from} -- curl "http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from}
to httpbin.${to}: %{http_code}\n"; done; done
    sleep.foo to httpbin.foo: 503
    sleep.foo to httpbin.bar: 503
    sleep.bar to httpbin.foo: 503
    sleep.bar to httpbin.bar: 503
```

클라이언트 측을 구성하려면 대상 TLS를 사용하도록 대상 규칙(destination rules)을 설정해야합니다. 적용 가능한 각 서비스 (또는 네임 스페이스)마다 하나씩 여러 대상 규칙을 사용할 수 있습니다. 그러나 * 와일드 카드가 있는 규칙을 사용하면 모든 서비스를 일치시켜 메쉬 전체 인증 정책(mesh-wide authentication policy)과 동등하게 적용하는 것이 더 편리합니다.

```
    $ kubectl apply -f - <<EOF
    apiVersion: "networking.istio.io/v1alpha3"
    kind: "DestinationRule"
    metadata:
      name: "default"
      namespace: "istio-system"
    spec:
      host: "*.local"
      trafficPolicy:
        tls:
          mode: ISTIO_MUTUAL
    EOF
```

> - Istio 1.1부터 클라이언트 네임 스페이스, 서버 네임 스페이스 및 글로벌 네임 스페이스 (기본적으로 istio-system)의 대상 규칙(destination rules) 만 서비스 순서로 고려됩니다.
> - 호스트 값 *.local은 외부 서비스가 아닌 클러스터의 서비스에만 일치하도록 제한합니다. 또한 대상 규칙의 이름 또는 네임 스페이스에는 제한이 없습니다.
> - ISTIO_MUTUAL TLS 모드를 사용하면 Istio는 내부 구현에 따라 키와 인증서 (예 : 클라이언트 인증서, 개인 키 및 CA 인증서)의 경로를 설정합니다.

Destination rules이 canarying 설정과 같은 비인증 이유로도 사용되지만 동일한 우선 순위가 적용됨을 잊지 마세요. 따라서 서비스가 어떤 이유로 특정 대상 대상 규칙을 필요로하는 경우 (for example, for a configuration load balancer) 규칙에 ISTIO_MUTUAL 모드를 사용하는 유사한 TLS 블록이 있어야합니다. 그렇지 않으면 mesh- 또는 namespace-wide TLS settings을 무시하고 TLS 사용하지 않도록 설정합니다.

위와 같이 testing 명령을 다시 실행하면 Istio 서비스 간의 모든 요청이 성공적으로 완료됩니다.

```
    $ for from in "foo" "bar"; do for to in "foo" "bar"; do kubectl exec $(kubectl
get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name}) -c sleep -n
${from} -- curl "http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from}
to httpbin.${to}: %{http_code}\n"; done; done
    sleep.foo to httpbin.foo: 200
    sleep.foo to httpbin.bar: 200
    sleep.bar to httpbin.foo: 200
    sleep.bar to httpbin.bar: 200
```

Request from non-Istio services to Istio services

예를 들어 Sidecar가 없는 sleep.legacy와 같은 non-Istio 서비스는 Istio 서비스와 연결에 필요한 TLS Connection을 시작할 수 없습니다. 결과적으로 sleep.legacy에서 httpbin.foo 또는 httpbin.bar로 가는 요청은 실패합니다 :

```
    $ for from in "legacy"; do for to in "foo" "bar"; do kubectl exec $(kubectl
get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name}) -c sleep -n
${from} -- curl "http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from}
to httpbin.${to}: %{http_code}\n"; done; done
    sleep.legacy to httpbin.foo: 000
    command terminated with exit code 56
    sleep.legacy to httpbin.bar: 000
    command terminated with exit code 56
```

> Envoy가 일반 텍스트 요청(plain-text requests)을 거부하는 방식으로 인하여 이 경우에는 curl 종료 코드 56 (네트워크 데이터 수신 실패)이 표시됩니다.

이는 의도 한대로 작동하지만 유감스럽게도 이러한 서비스에 대한 인증 요구 사항(authentication requirements)을 줄이지 않으면 해결할 수 없습니다.

Request from Istio services to non-Istio services

sleep.foo (또는 sleep.bar)에서 httpbin.legacy로 요청을 보냅니다. Istio가 상호 TLS를 사용하기 위해 목적지 규칙(destination rule)에 지시한 대로 클라이언트를 구성하므로 요청이 실패하는 것을 볼 수 있지만 httpbin.legacy에는 Sidecar가 없으므로 이를 처리 할 수 없습니다.

```
    $ for from in "foo" "bar"; do for to in "legacy"; do kubectl exec $(kubectl
get pod -l app=sleep -n ${from} -o jsonpath={.items..metadata.name}) -c sleep -n
${from} -- curl "http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from}
to httpbin.${to}: %{http_code}\n"; done; done
    sleep.foo to httpbin.legacy: 503
    sleep.bar to httpbin.legacy: 503
```

이 문제를 해결하기 위해 대상 규칙을 추가하여 httpbin.legacy의 TLS 설정을 덮어 쓸 수 있습니다.

예를 들면 :

```
    $ kubectl apply -f - <<EOF
    apiVersion: networking.istio.io/v1alpha3
    kind: DestinationRule
    metadata:
     name: "httpbin-legacy"
     namespace: "legacy"
    spec:
     host: "httpbin.legacy.svc.cluster.local"
     trafficPolicy:
       tls:
         mode: DISABLE
    EOF
```

> 이 대상 규칙은 서버의 네임 스페이스 (httpbin.legacy)에 있으므로 istio-system에 정의 된 global destination rule 보다 선호됩니다.

> https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
> https://arisu1000.tistory.com/27859 ~/istio-1.1.3$ kubectl exec -it sleep-bcc758cff-sj8qx -n foo -- cat /etc/

Request from Istio services to Kubernetes API server

Kubernetes API 서버에는 Sidecar가 없으므로 Istio가 없는 서비스(on-Istio service)에 요청을 보낼 때와 같은 문제로 인해 sleep.foo와 같은 Istio 서비스 요청이 실패합니다.

```
    $ TOKEN=$(kubectl describe secret $(kubectl get secrets | grep default-token |
cut -f1 -d ' ' | head -1) | grep -E '^token' | cut -f2 -d':' | tr -d '\t')
    $ kubectl exec $(kubectl get pod -l app=sleep -n foo -o jsonpath=
{.items..metadata.name}) -c sleep -n foo -- curl https://kubernetes.default/api --
header "Authorization: Bearer $TOKEN" --insecure -s -o /dev/null -w "%
{http_code}\n"
    000
    command terminated with exit code 35
```

다시 말하지만, API 서버 (kubernetes.default)의 대상 규칙을 재정의하여 이를 수정할 수 있습니다.

```
    $ kubectl apply -f - <<EOF
    apiVersion: networking.istio.io/v1alpha3
    kind: DestinationRule
    metadata:
     name: "api-server"
     namespace: istio-system
    spec:
     host: "kubernetes.default.svc.cluster.local"
     trafficPolicy:
       tls:
         mode: DISABLE
    EOF
```

이 규칙은 위의 글로벌 인증 정책(global authentication policy) 및 대상 규칙(destination rule)과 함께 상호 TLS가 활성화 된 Istio를 설치할 때 시스템에 자동으로 주입됩니다(injected). 아래와 같이 규칙을 추가 한 후, 위의 테스트 명령을 다시 실행하여 200을 리턴하는지 확인하세요.

```
    $ TOKEN=$(kubectl describe secret $(kubectl get secrets | grep default-token |
cut -f1 -d ' ' | head -1) | grep -E '^token' | cut -f2 -d':' | tr -d '\t')
    $ kubectl exec $(kubectl get pod -l app=sleep -n foo -o jsonpath=
{.items..metadata.name}) -c sleep -n foo -- curl https://kubernetes.default/api --
header "Authorization: Bearer $TOKEN" --insecure -s -o /dev/null -w "%
{http_code}\n"
    200
```

Cleanup part 1

세션에 추가 된 전역 인증 정책(global authentication policy) 및 대상 규칙(destination rules)을 제거합니다.

```
$ kubectl delete meshpolicy default
$ kubectl delete destinationrules httpbin-legacy -n legacy
$ kubectl delete destinationrules api-server -n istio-system
```

Enable mutual TLS per namespace or service

전체 Mesh에 대한 인증 정책을 지정하는 것 외에도 Istio를 사용하여 특정 네임 스페이스 또는 서비스에 대한 정책을 지정할 수 있습니다. 네임 스페이스 차원의 정책(namespace-wide policy)은 메시 전체의 정책(over the mesh-wide policy)보다 우선 순위가 높지만 서비스 별 정책(service-specific policy)은 우선 순위가 더 높습니다.

Namespace-wide policy

아래 예제는 네임 스페이스 foo의 모든 서비스에 대해 상호 TLS를 사용하도록 설정하는 정책을 보여줍니다. 보시다시피, "MeshPolicy"가 아닌 kind: “Policy”를 사용하고 네임 스페이스를 지정합니다.이 경우에는 foo입니다. 네임 스페이스 값을 지정하지 않으면 정책이 default 네임 스페이스에 적용됩니다.

```
$ kubectl apply -f - <<EOF
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "default"
  namespace: "foo"
spec:
  peers:
  - mtls: {}
EOF
```

> 메쉬 전체 정책(mesh-wide policy)과 마찬가지로 네임 스페이스 차원의 정책(namespace-wide policy)은 default라는 이름을 가져야 하며 특정 서비스를 제한하지 않습니다 (targets 섹션 없음)

해당 대상 규칙(destination rule) 추가 :

```
$ kubectl apply -f - <<EOF
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "default"
  namespace: "foo"
spec:
  host: "*.foo.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

```
    EOF
```

> Host *.foo.svc.cluster.local은 일치를 foo 네임 스페이스의 서비스에만 제한합니다.

```
    $ for from in "foo" "bar" "legacy"; do for to in "foo" "bar" "legacy"; do
kubectl exec $(kubectl get pod -l app=sleep -n ${from} -o jsonpath=
{.items..metadata.name}) -c sleep -n ${from} -- curl
"http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from} to httpbin.${to}:
%{http_code}\n"; done; done
    sleep.foo to httpbin.foo: 200
    sleep.foo to httpbin.bar: 503
    sleep.foo to httpbin.legacy: 503
    sleep.bar to httpbin.foo: 200
    sleep.bar to httpbin.bar: 503
    sleep.bar to httpbin.legacy: 503
    sleep.legacy to httpbin.foo: 000
    command terminated with exit code 56
    sleep.legacy to httpbin.bar: 200
    sleep.legacy to httpbin.legacy: 200
```

Service-specific policy

특정 서비스에 대한 인증 정책 및 대상 규칙을 설정할 수도 있습니다. 이 명령을 실행하여 httpbin.bar 서비스에 대해서만 다른 정책을 설정하십시오.

```
    $ cat <<EOF | kubectl apply -n bar -f -
    apiVersion: "authentication.istio.io/v1alpha1"
    kind: "Policy"
    metadata:
      name: "httpbin"
    spec:
      targets:
      - name: httpbin
      peers:
      - mtls: {}
    EOF
```

And a destination rule:

```
    $ cat <<EOF | kubectl apply -n bar -f -
    apiVersion: "networking.istio.io/v1alpha3"
```

```
    kind: "DestinationRule"
    metadata:
      name: "httpbin"
    spec:
      host: "httpbin.bar.svc.cluster.local"
      trafficPolicy:
        tls:
          mode: ISTIO_MUTUAL
    EOF
```

이 정책 및 대상 규칙은 네임 스페이스 foo의 서비스에만 적용되므로 Sidecar가 없는 Client (sleep.legacy)에서 httpbin.foo의 요청이 실패하기 시작해야합니다.

```
    $ for from in "foo" "bar" "legacy"; do for to in "foo" "bar" "legacy"; do
kubectl exec $(kubectl get pod -l app=sleep -n ${from} -o jsonpath=
{.items..metadata.name}) -c sleep -n ${from} -- curl
"http://httpbin.${to}:8000/ip" -s -o /dev/null -w "sleep.${from} to httpbin.${to}:
%{http_code}\n"; done; done
    sleep.foo to httpbin.foo: 200
    sleep.foo to httpbin.bar: 200
    sleep.foo to httpbin.legacy: 200
    sleep.bar to httpbin.foo: 200
    sleep.bar to httpbin.bar: 200
    sleep.bar to httpbin.legacy: 200
    sleep.legacy to httpbin.foo: 000
    command terminated with exit code 56
    sleep.legacy to httpbin.bar: 200
    sleep.legacy to httpbin.legacy: 200
```

> - 이 예제에서는 Meta Data에 네임 스페이스를 지정하지 않지만 명령 줄 (-n bar)에 넣어두면 동일한 효과가 있습니다.
> - 인증 정책(authentication policy) 및 대상 규칙 이름(destination rule name)에는 제한이 없습니다. 이 예제는 단순화를 위해 서비스 자체의 이름을 사용합니다.

다시 probing 명령을 실행하세요. 예상했든 대로 sleep.legacy에서 httpbin.bar 로의 요청하면 같은 이유로 실패하기 시작 합니다.

```
    ...
    sleep.legacy to httpbin.bar: 000
    command terminated with exit code 56
```

우리가 네임 스페이스 바(namespace bar)에 더 많은 서비스를 가지고 있다면, 그들에 대한 트래픽이 영향을받지 않을 것입니다. 이 동작을 보여주기 위해 더 많은 서비스를 추가하는 대신 정책을 약간 편집하여 특정 포트에 적용합니다.

```
$ cat <<EOF | kubectl apply -n bar -f -
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "httpbin"
spec:
  targets:
  - name: httpbin
    ports:
    - number: 1234
  peers:
  - mtls: {}
EOF
```

대상 규칙(destination rule)에 대한 해당 변경 사항은 다음과 같습니다.

```
$ cat <<EOF | kubectl apply -n bar -f -
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "httpbin"
spec:
  host: httpbin.bar.svc.cluster.local
  trafficPolicy:
    tls:
      mode: DISABLE
    portLevelSettings:
    - port:
        number: 1234
      tls:
        mode: ISTIO_MUTUAL
EOF
```

이 새로운 정책은 Port 1234의 httpbin 서비스에만 적용됩니다. 따라서 Port 8000에서 상호 TLS(mutual TLS)가 비활성화되고 sleep.legacy의 요청이 다시 시작됩니다.

```
$ kubectl exec $(kubectl get pod -l app=sleep -n legacy -o jsonpath=
{.items..metadata.name}) -c sleep -n legacy -- curl http://httpbin.bar:8000/ip -s
-o /dev/null -w "%{http_code}\n"
```

```
200
```

Policy precedence

서비스 별 정책(service-specific policy)이 네임 스페이스 차원의 정책(namespace-wide policy)보다 우선하는 방식을 설명하기 위해 아래와 같이 httpbin.foo의 상호 TLS를 사용하지 않도록 설정하는 정책을 추가 할 수 있습니다. 네임 스페이스 foo에있는 모든 서비스에 대해 상호 TLS를 가능하게하고 sleep.legacy에서 httpbin.foo 로의 요청이 실패하고 있음을 관찰하는 네임 스페이스 차원의 정책(namespace-wide policy)을 이미 만들었습니다 (위 참조).

```
$ cat <<EOF | kubectl apply -n foo -f -
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "overwrite-example"
spec:
  targets:
  - name: httpbin
EOF
```

and destination rule:

```
$ cat <<EOF | kubectl apply -n foo -f -
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "overwrite-example"
spec:
  host: httpbin.foo.svc.cluster.local
  trafficPolicy:
    tls:
      mode: DISABLE
EOF
```

sleep.legacy에서 요청을 다시 실행하면 서비스별 정책(service-specific policy)이 네임 스페이스 전체 정책(namespace-wide policy)보다 우선하는 것을 확인하고 성공 반환 코드를 다시 표시해야합니다 (200).

```
    $ kubectl exec $(kubectl get pod -l app=sleep -n legacy -o jsonpath=
{.items..metadata.name}) -c sleep -n legacy -- curl http://httpbin.foo:8000/ip -s
-o /dev/null -w "%{http_code}\n"
```

```
200
```

Cleanup part 2

Remove policies and destination rules created in the above steps:

```
$ kubectl delete policy default overwrite-example -n foo
$ kubectl delete policy httpbin -n bar
$ kubectl delete destinationrules default overwrite-example -n foo
$ kubectl delete destinationrules httpbin -n bar
```

End-user authentication

이 기능을 시험하기 위해서는 유효한 JWT가 필요합니다. JWT는 데모에 사용할 JWKS 엔드 포인트와 일치해야 합니다. 이 자습서에서는이 JWT 테스트와 Istio 코드 기반의 JWKS 끝점을 사용합니다. 또한 편의를 위해 httpb.foo를 ingressgateway를 통해 노출 시키십시오 (자세한 내용은 ingress 작업 참조).

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: httpbin-gateway
  namespace: foo
spec:
  selector:
    istio: ingressgateway # use Istio default gateway implementation
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
EOF
```

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
```

```
    namespace: foo
  spec:
    hosts:
    - "*"
    gateways:
    - httpbin-gateway
    http:
    - route:
      - destination:
          port:
            number: 8000
          host: httpbin.foo.svc.cluster.local
  EOF
```

Get ingress IP

```
$ export INGRESS_HOST=$(kubectl -n istio-system get service istio-
ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
```

And run a test query

```
$ curl $INGRESS_HOST/headers -s -o /dev/null -w "%{http_code}\n"
200
```

> 시간이 좀 소요됨

이제 httpbin.foo에 대한 최종 사용자 JWT가 필요한 정책을 추가하세요. 다음 명령은 httpbin.foo에 대한 서비스 특정 정책이 없다고 가정합니다 (위에서 설명한대로 정리를 실행하는 경우에 해당). kubectl get policies.authentication.istio.io -n foo을 실행하여 확인할 수 있습니다.

```
$ cat <<EOF | kubectl apply -n foo -f -
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "jwt-example"
spec:
  targets:
  - name: httpbin
  origins:
  - jwt:
```

```
            issuer: "testing@secure.istio.io"
            jwksUri: "https://raw.githubusercontent.com/istio/istio/release-
1.1/security/tools/jwt/samples/jwks.json"
      principalBinding: USE_ORIGIN
    EOF
```

이전과 같은 curl 명령은 서버가 JWT를 기대하지만 401이 제공되지 않았기 때문에 401 오류 코드와 함께 반환됩니다.

```
    $ curl $INGRESS_HOST/headers -s -o /dev/null -w "%{http_code}\n"
    401
```

> 시간이 좀 소요 됨

위에 생성 된 유효한 토큰을 첨부하면 성공을 반환합니다.

```
    $ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-
1.1/security/tools/jwt/samples/demo.jwt -s)
    $ curl --header "Authorization: Bearer $TOKEN" $INGRESS_HOST/headers -s -o
/dev/null -w "%{http_code}\n"
    200
```

> 시간이 좀 소요됨

JWT 유효성 검사(JWT validation)의 다른 측면을 관찰하려면 gen-jwt.py 스크립트를 사용하여 다른 발급자, 대상, 만료 날짜 등을 테스트 할 새로운 토큰을 생성하세요. 스크립트는 Istio 저장소에서 다운로드 할 수 있습니다.

```
    $ wget https://raw.githubusercontent.com/istio/istio/release-
1.1/security/tools/jwt/samples/gen-jwt.py
    $ chmod +x gen-jwt.py
```

You also need the key.pem file:

```
    $ wget https://raw.githubusercontent.com/istio/istio/release-
1.1/security/tools/jwt/samples/key.pem
```

예를 들어, 아래 명령은 5 초 후에 만료되는 토큰을 만듭니다. 보시다시피 Istio는 먼저 해당 토큰을 사용하여 요청을 인증하지만 5 초 후에 거부합니다.

```
$ TOKEN=$(./gen-jwt.py ./key.pem --expire 5)
$ for i in `seq 1 10`; do curl --header "Authorization: Bearer $TOKEN"
$INGRESS_HOST/headers -s -o /dev/null -w "%{http_code}\n"; sleep 1; done
200
200
200
200
200
401
401
401
401
401
```

> gen-jwt.py 실행할 때 jwcrypto 모듈이 없어서 Error가 발생하면 pip install jwcrypto 명령으로 해당 모듈 설치 후 제실행할 것

또한 ingress gateway에 JWT 정책을 추가 할 수도 있습니다 (예 : service istio-ingressgateway.istio-system.svc.cluster.local). 이것은 종종 개별 서비스 대신 게이트웨이에 바인딩 된 모든 서비스에 대한 JWT 정책을 정의하는 데 사용됩니다.

End-user authentication with per-path requirements

최종 사용자 인증(End-user authentication)은 요청 경로(request path)를 기반으로 활성화 또는 비활성화 할 수 있습니다. 일부 경로 (예 : 상태 확인 또는 상태 보고서에 사용되는 경로)에 대해 인증을 사용하지 않으려는 경우에 유용합니다. 예를 들어 health check 또는 status report, 다른 경로(different paths)에서 다른 JWT 요구 사항(JWT requirements)을 지정할 수도 있습니다.

> 경로 별 요구 사항(per-path requirements)을 가진 최종 사용자 인증(end-user authentication)은 Istio 1.1의 실험적 기능이므로 프로덕션 용도로 권장되지 않습니다.

Disable End-user authentication for specific paths

path /user-agent에 대한 최종 사용자 인증(End-user authentication)을 사용하지 않도록 jwt-example 정책을 수정합니다.

```
$ cat <<EOF | kubectl apply -n foo -f -
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "jwt-example"
```

```
    spec:
      targets:
      - name: httpbin
      origins:
      - jwt:
          issuer: "testing@secure.istio.io"
          jwksUri: "https://raw.githubusercontent.com/istio/istio/release-
1.1/security/tools/jwt/samples/jwks.json"
          trigger_rules:
          - excluded_paths:
            - exact: /user-agent
      principalBinding: USE_ORIGIN
    EOF
```

JWT 토큰없이 path /user-agent에 액세스 할 수 있는지 확인합니다.

```
$ curl $INGRESS_HOST/user-agent -s -o /dev/null -w "%{http_code}\n"
200
```

Confirm it's denied to access paths other than /user-agent without JWT tokens: JWT 토큰없이 /user-agent 이외의 경로에 액세스하는 것이 거부되었는지 확인합니다.

```
$ curl $INGRESS_HOST/headers -s -o /dev/null -w "%{http_code}\n"
401
```

Enable End-user authentication for specific paths

경로 /ip에 대해서만 최종 사용자 인증(End-user authentication)을 사용하도록 jwt-example 정책을 수정합니다.

```
$ cat <<EOF | kubectl apply -n foo -f -
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "jwt-example"
spec:
  targets:
  - name: httpbin
  origins:
  - jwt:
      issuer: "testing@secure.istio.io"
      jwksUri: "https://raw.githubusercontent.com/istio/istio/release-
```

```
1.1/security/tools/jwt/samples/jwks.json"
          trigger_rules:
          - included_paths:
            - exact: /ip
      principalBinding: USE_ORIGIN
    EOF
```

JWT 토큰없이 /ip 이외의 경로에 액세스 할 수 있는지 확인합니다.

```
$ curl $INGRESS_HOST/user-agent -s -o /dev/null -w "%{http_code}\n"
200
```

JWT 토큰없이 경로 /ip에 액세스하는 것이 거부되었는지 확인합니다.

```
$ curl $INGRESS_HOST/ip -s -o /dev/null -w "%{http_code}\n"
401
```

유효한 JWT 토큰을 사용하여 경로 /ip에 액세스 할 수 있는지 확인합니다.

```
$ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-
1.1/security/tools/jwt/samples/demo.jwt -s)
$ curl --header "Authorization: Bearer $TOKEN" $INGRESS_HOST/ip -s -o
/dev/null -w "%{http_code}\n"
200
```

End-user authentication with mutual TLS

최종 사용자 인증(End-user authentication) 및 상호 TLS(mutual TLS)를 함께 사용할 수 있습니다. 위의 정책을 수정하여 상호 TLS 및 최종 사용자 JWT 인증(end-user JWT authentication)을 모두 정의하세요.

```
$ cat <<EOF | kubectl apply -n foo -f -
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "jwt-example"
spec:
  targets:
```

```
      - name: httpbin
      peers:
      - mtls: {}
      origins:
      - jwt:
          issuer: "testing@secure.istio.io"
          jwksUri: "https://raw.githubusercontent.com/istio/istio/release-
1.1/security/tools/jwt/samples/jwks.json"
      principalBinding: USE_ORIGIN
    EOF
```

> jwt-example 정책을 제출하지 않은 경우 istio create를 사용하세요.

And add a destination rule:

```
    $ kubectl apply -f - <<EOF
    apiVersion: "networking.istio.io/v1alpha3"
    kind: "DestinationRule"
    metadata:
      name: "httpbin"
      namespace: "foo"
    spec:
      host: "httpbin.foo.svc.cluster.local"
      trafficPolicy:
        tls:
          mode: ISTIO_MUTUAL
    EOF
```

> 이미 상호 TLS(mutual TLS) 메시 전체(mesh-wide) 또는 네임 스페이스 전체(namespace-wide)를 사용하도록 설정 한 경우 호스트 httpbin.foo는 다른 대상 규칙(other destination rule)에 이미 적용됩니다. 따라서 이 대상 규칙을 추가 할 필요가 없습니다. 한편, 서비스 특정 정책(service-specific policy)은 mesh-wide (or namespace-wide) 정책을 완전히 무시하므로 mtls stanza를 인증 정책에 추가해야 합니다.

이러한 변경 후에는 ingress gateway를 포함한 Istio 서비스에서 httpbin.foo로 가는 트래픽이 상호 TLS(mutual TLS)를 사용합니다. 위의 테스트 명령은 계속 작동합니다. 올바른 토큰이 주어지면, httpbin.foo로 직접 Istio 서비스로 부터의 요청도 가능합니다. : -> 아래와 같이

```
    $ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-
1.1/security/tools/jwt/samples/demo.jwt -s)
    $ kubectl exec $(kubectl get pod -l app=sleep -n foo -o jsonpath=
{.items..metadata.name}) -c sleep -n foo -- curl http://httpbin.foo:8000/ip -s -o
/dev/null -w "%{http_code}\n" --header "Authorization: Bearer $TOKEN"
    200
```

그러나 일반 텍스트(plain-text)를 사용하는 비 Istio 서비스(non-Istio services)의 요청은 실패합니다.

```
    $ kubectl exec $(kubectl get pod -l app=sleep -n legacy -o jsonpath=
{.items..metadata.name}) -c sleep -n legacy -- curl http://httpbin.foo:8000/ip -s
-o /dev/null -w "%{http_code}\n" --header "Authorization: Bearer $TOKEN"
    000
    command terminated with exit code 56
```

Cleanup part 3

1. Remove authentication policy:

```
    $ kubectl -n foo delete policy jwt-example
```

2. Remove destination rule:

```
    $ kubectl -n foo delete destinationrule httpbin
```

3. If you are not planning to explore any follow-on tasks, you can remove all resources simply by deleting test namespaces.

```
    $ kubectl delete ns foo bar legacy
```

Authorization for HTTP Services

Authorization for TCP Services

Authorization for groups and list claims

Authorization permissive mode

권한 허용 모드(authorization permissive mode)에서는 프로덕션 환경에서 적용하기 전에 권한 policy(authorization policies)를 검증 할 수 있습니다.

허가 허용 모드(authorization permissive mode)는 버전 1.1의 시험적인 기능입니다. 인터페이스는 향후 릴리스에서 변경 될 수 있습니다. 허용 모드(permissive mode) 기능을 시험하고 싶지 않은 경우 Istio 인증(Istio authorization)을 직접 허용 모드(permissive mode)로 설정하여 건너 뛰게 할 수 있습니다.

이 작업에서는 권한 부여(authorization)를 위해 허용 모드(permissive mode) 사용과 관련된 두 가지 시나리오를 다룹니다.

- 권한 부여(authorization)가 사용 불가능한 환경의 경우,이 태스크는 권한 부여(authorization)를 사용하는 것이 안전한지 여부를 테스트하는데 도움이됩니다.
- 권한 부여(authorization)가 사용 가능한 환경의 경우,이 태스크는 새로운 권한(authorization) policy를 추가하는 것이 안전한지 여부를 테스트하는데 도움이됩니다.

Before you begin

이 작업을 완료하려면 먼저 다음 작업을 수행해야합니다.

- Read the authorization concept.
- Follow the instructions in the Kubernetes quick start to install Istio with mutual TLS enabled.
- Deploy the Bookinfo sample application.
- Bookinfo 응용 프로그램에 대한 서비스 계정(service accounts)을 만듭니다. 다음 명령을 실행하여 제품 페이지에 대한 서비스 계정 bookinfo-productpage 및 검토를 위한 서비스 계정 bookinfo-reviews를 만듭니다.

```
    $ kubectl apply -f <(istioctl kube-inject -f
samples/bookinfo/platform/kube/bookinfo-add-serviceaccount.yaml)
```

Test enabling authorization globally

다음 단계에서는 승인 허용 모드(authorization permissive mode)를 사용하여 globally 인증(authorization)을 사용하는 것이 안전한지 테스트하는 방법을 보여줍니다.

1. To enable the permissive mode in the global authorization configuration, run the following command:

```
$ kubectl apply -f - <<EOF
apiVersion: "rbac.istio.io/v1alpha1"
kind: ClusterRbacConfig
metadata:
  name: default
spec:
  mode: 'ON_WITH_INCLUSION'
  inclusion:
    namespaces: ["default"]
  enforcement_mode: PERMISSIVE
EOF
```

2. Go to the productpage at http://$GATEWAY_URL/productpage and verify that everything works fine.

3. Apply the rbac-permissive-telemetry.yaml YAML file to enable the metric collection for the permissive mode:

```
$ kubectl apply -f samples/bookinfo/platform/kube/rbac/rbac-permissive-
telemetry.yaml
logentry.config.istio.io/rbacsamplelog created
stdio.config.istio.io/rbacsamplehandler created
rule.config.istio.io/rabcsamplestdio created
```

4. Send traffic to the sample application with the following command:

```
$ curl http://$GATEWAY_URL/productpage
```

5. Go to the productpage at http://$GATEWAY_URL/productpage and verify that everything works fine.

6. Get the log for telemetry and search for the permissiveResponseCode with the following command:

```
$ kubectl -n istio-system logs -l istio-mixer-type=telemetry -c mixer | grep
\"instance\":\"rbacsamplelog.logentry.istio-system\"
{"level":"warn","time":"2018-08-
30T21:53:42.059444Z","instance":"rbacsamplelog.logentry.istio-
system","destination":"ratings","latency":"9.158879ms","permissiveResponseCo
de":"denied","permissiveResponsePolicyID":"","responseCode":200,"responseSiz
e":48,"source":"reviews","user":"cluster.local/ns/default/sa/bookinfo-
reviews"}
```

```
{"level":"warn","time":"2018-08-
30T21:53:41.037824Z","instance":"rbacsamplelog.logentry.istio-
system","destination":"reviews","latency":"1.091670916s","permissiveResponse
Code":"denied","permissiveResponsePolicyID":"","responseCode":200,"responseS
ize":379,"source":"productpage","user":"cluster.local/ns/default/sa/bookinfo
-productpage"}
{"level":"warn","time":"2018-08-
30T21:53:41.019851Z","instance":"rbacsamplelog.logentry.istio-
system","destination":"productpage","latency":"1.112521495s","permissiveResp
onseCode":"denied","permissiveResponsePolicyID":"","responseCode":200,"respo
nseSize":5723,"source":"istio-
ingressgateway","user":"cluster.local/ns/istio-system/sa/istio-
ingressgateway-service-account"}
```


7. Verify that the the log shows a responseCode of 200 and a permissiveResponseCode of denied.

8. Apply the productpage-policy.yaml authorization policy in permissive mode with the following command:

```
$ kubectl apply -f samples/bookinfo/platform/kube/rbac/productpage-
policy.yaml
```

9. Send traffic to the sample application with the following command:

```
$ curl http://$GATEWAY_URL/productpage
```

10. Get the log for telemetry and search for the permissiveResponseCode with the following command:


```
$ kubectl -n istio-system logs -l istio-mixer-type=telemetry -c mixer | grep
\"instance\":\"rbacsamplelog.logentry.istio-system\"
{"level":"warn","time":"2018-08-
30T21:55:53.590430Z","instance":"rbacsamplelog.logentry.istio-
system","destination":"ratings","latency":"4.415633ms","permissiveResponseCode":"d
enied","permissiveResponsePolicyID":"","responseCode":200,"responseSize":48,"sourc
e":"reviews","user":"cluster.local/ns/default/sa/bookinfo-reviews"}
{"level":"warn","time":"2018-08-
30T21:55:53.565914Z","instance":"rbacsamplelog.logentry.istio-
system","destination":"reviews","latency":"32.97524ms","permissiveResponseCode":"d
enied","permissiveResponsePolicyID":"","responseCode":200,"responseSize":379,"sour
ce":"productpage","user":"cluster.local/ns/default/sa/bookinfo-productpage"}
{"level":"warn","time":"2018-08-
```

```
30T21:55:53.544441Z","instance":"rbacsamplelog.logentry.istio-
system","destination":"productpage","latency":"57.800056ms","permissiveResponseCod
e":"allowed","permissiveResponsePolicyID":"productpage-
viewer","responseCode":200,"responseSize":5723,"source":"istio-
ingressgateway","user":"cluster.local/ns/istio-system/sa/istio-ingressgateway-
service-account"}
```


11. Verify that the the log shows a responseCode of 200 and a permissiveResponseCode of allowed for productpage service.

12. Remove the YAML files related to enabling the permissive mode:

```
$ kubectl delete -f samples/bookinfo/platform/kube/rbac/productpage-
policy.yaml
$ kubectl delete -f samples/bookinfo/platform/kube/rbac/rbac-config-on-
permissive.yaml
$ kubectl delete -f samples/bookinfo/platform/kube/rbac/rbac-permissive-
telemetry.yaml
```

13. Congratulations! You tested an authorization policy with permissive mode and verified it works as expected. To enable the authorization policy, follow the steps described in the Enabling Istio authorization task.

Test adding authorization policy

The following steps show how to test a new authorization policy with permissive mode when authorization has already been enabled.

1. Allow access to the producepage service by following the instructions in Enabling authorization for HTTP services step 1.

2. Allow access to the details and reviews service in permissive mode with the following command:

```
    $ kubectl apply -f samples/bookinfo/platform/kube/rbac/details-reviews-
policy-permissive.yaml
```

3. Verify there are errors Error fetching product details and Error fetching product reviews on the Bookinfo productpage by pointing your browser at the productpage (http://$GATEWAY_URL/productpage), These errors are expected because the policy is in PERMISSIVE mode.

4. Apply the rbac-permissive-telemetry.yaml YAML file to enable the permissive mode metric collection.

```
$ kubectl apply -f samples/bookinfo/platform/kube/rbac/rbac-permissive-
```

```
telemetry.yaml
```

5. Send traffic to the sample application:

```
$ curl http://$GATEWAY_URL/productpage
```

6. Get the log for telemetry and search for the permissiveResponseCode with the following command:

```
$ kubectl -n istio-system logs -l istio-mixer-type=telemetry -c mixer |
grep \"instance\":\"rbacsamplelog.logentry.istio-system\"
{"level":"warn","time":"2018-08-
30T22:59:42.707093Z","instance":"rbacsamplelog.logentry.istio-
system","destination":"details","latency":"423.381µs","permissiveResponseCod
e":"allowed","permissiveResponsePolicyID":"details-reviews-
viewer","responseCode":403,"responseSize":19,"source":"productpage","user":"
cluster.local/ns/default/sa/bookinfo-productpage"}
{"level":"warn","time":"2018-08-
30T22:59:42.763423Z","instance":"rbacsamplelog.logentry.istio-
system","destination":"reviews","latency":"237.333µs","permissiveResponseCod
e":"allowed","permissiveResponsePolicyID":"details-reviews-
viewer","responseCode":403,"responseSize":19,"source":"productpage","user":"
cluster.local/ns/default/sa/bookinfo-productpage"}
```

7. Verify that the the log shows a responseCode of 403 and a permissiveResponseCode of allowed for ratings and reviews services.

8. Remove the YAML files related to enabling the permissive mode:

```
$ kubectl delete -f samples/bookinfo/platform/kube/rbac/details-reviews-
policy-permissive.yaml
$ kubectl delete -f samples/bookinfo/platform/kube/rbac/rbac-permissive-
telemetry.yaml
```

9. Congratulations! You tested adding an authorization policy with permissive mode and verified it will work as expected. To add the authorization policy, follow the steps described in the Enabling Istio authorization task.

Istio Vault CA Integration

Mutual TLS Deep-Dive

이 작업을 통해 상호 TLS(mutual TLS)를 면밀히 관찰하고 설정을 학습 할 수 있습니다. 이 작업에서는 다음을 전제로합니다.

- 인증 정책(authentication policy) 작업을 완료했습니다.
- 인증 정책(authentication policy)을 사용하여 상호 TLS(mutual TLS)를 사용하는 것에 익숙합니다.
- **Istio는 글로벌 상호 TLS(global mutual TLS)가 활성화 된 Kubernetes에서 실행됩니다. 지침에 따라 Istio를 설치할 수 있습니다. 이미 Istio가 설치되어있는 경우 인증 정책(authentication policies) 및 대상 규칙(destination rules)을 추가 또는 수정하여 이 작업(Task)에서 설명한대로 상호 TLS(mutual TLS)를 사용할 수 있습니다.**
- Default namespace에 httpbbin과 Envoy Sidecar 설치된 sleep를 배포 합니다. 예를 들어, manual sidecar injection으로 이러한 서비스를 배치하는 명령은 다음과 같습니다.

```
$ kubectl apply -f <(istioctl kube-inject -f samples/httpbin/httpbin.yaml)
$ kubectl apply -f <(istioctl kube-inject -f samples/sleep/sleep.yaml)
```

Verify Citadel runs properly

Citadel은 Istio의 key management 서비스입니다. Citadel는 올바르게 작동하려면 상호 TLS(mutual TLS)를 제대로 실행해야 합니다. 다음 명령을 사용하여 클러스터 수준의 Citadel(cluster-level Citadel)이 올바르게 실행되는지 확인하세요.

```
$ kubectl get deploy -l istio=citadel -n istio-system
NAME            DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
istio-citadel   1         1         1            1           1m
```

"AVAILABLE" Column이 1 인 경우 Citadel이 작동 중인 상태 입니다.

Verify keys and certificates installation

Istio는 모든 sidecar containers에 상호 TLS 인증(mutual TLS authentication)을 위해 필요한 키와 인증서를 자동으로 설치합니다. /etc/certs에 키와 인증서 파일이 있는지 확인하려면 아래 명령을 실행하세요.

```
    $ kubectl exec $(kubectl get pod -l app=httpbin -o jsonpath=
{.items..metadata.name}) -c istio-proxy -- ls /etc/certs
    cert-chain.pem
    key.pem
```

```
    root-cert.pem
```

> cert-chain.pem은 Envoy의 인증서(cert)로 상대방(the other side)에게 제공해야 합니다. key.pem은 Envoy의 개인 키(private key)이며 Envoy의 cert와 cert-chain.pem이 쌍을 이룹니다. root-cert.pem은 Peer의 인증서(peer's cert)를 확인하기 위한 Root 인증서입니다. 이 예에서는 클러스터에 Citadel 하나만 있으므로 모든 Envoys에는 동일한 root-cert.pem이 있습니다.

Use the openssl tool to check if certificate is valid (current time should be in between Not Before and Not After)

openssl 도구를 사용하여 인증서가 유효한지 확인하세요 (현재 시간은 Not Before와 Not After 사이에 있어야 함.)

```
    $ kubectl exec $(kubectl get pod -l app=httpbin -o jsonpath=
{.items..metadata.name}) -c istio-proxy -- cat /etc/certs/cert-chain.pem | openssl
x509 -text -noout  | grep Validity -A 2
    Validity
            Not Before: May 17 23:02:11 2018 GMT
            Not After : Aug 15 23:02:11 2018 GMT
```

> PEM 인코딩된 인증서를 파싱해서 정보를 출력 $ openssl x509 -text -noout -in localhost.crt

클라이언트 인증서(client certificate)의 ID(identity)를 확인할 수도 있습니다.

```
    $ kubectl exec $(kubectl get pod -l app=httpbin -o jsonpath=
{.items..metadata.name}) -c istio-proxy -- cat /etc/certs/cert-chain.pem | openssl
x509 -text -noout  | grep 'Subject Alternative Name' -A 1
            X509v3 Subject Alternative Name:
                URI:spiffe://cluster.local/ns/default/sa/default
```

Istio에서 서비스 신원(service identity)에 대한 자세한 정보는 Istio 신분(Istio identity)을 확인하세요.

Verify mutual TLS configuration

istioctl 도구를 사용하여 상호 TLS(mutual TLS) 설정이 유효한지 확인하세요. 대상 규칙(destination rule)은 클라이언트 네임 스페이스에 따라 다르므로 istioctl 명령에는 클라이언트의 포드가 필요합니다. 대상 서비스를 제공하여 해당 서비스로만 상태를 필터링 할 수도 있습니다.

다음 명령은 httpbin.default.svc.cluster.local 서비스의 인증 정책(authentication policy)을 식별하고 sleep App의 동일한 Pod에서 본 서비스의 대상 규칙(destination rules)을 식별합니다.

```
    $ SLEEP_POD=$(kubectl get pod -l app=sleep -o jsonpath=
{.items..metadata.name})
    $ istioctl authn tls-check ${SLEEP_POD} httpbin.default.svc.cluster.local
```

다음 예제 출력에서 볼 수 있습니다 :

- 상호 TLS(Mutual TLS)는 포트 8000의 httpbin.default.svc.cluster.local에 대해 일관되게 설정됩니다.
- Istio는 메쉬 전체 기본 인증 정책(mesh-wide default authentication policy)을 사용합니다.
- Istio는 istio-system 네임 스페이스에 기본 대상 규칙(default destination rule)을가집니다.

```
    HOST:PORT                                  STATUS     SERVER     CLIENT
AUTHN POLICY        DESTINATION RULE
    httpbin.default.svc.cluster.local:8000     OK         mTLS       mTLS
default/            default/istio-system
```

The output shows:

- STATUS: 이경우에는 httpbin service 인 Server와 Client 또는 httpbin를 호출하는 client간의 TLS Setting 일관되는지 여부
- SERVER: 서버에서 사용되는 모드
- CLIENT: 클라이언트에서 사용되는 모드.
- AUTHN POLICY: the name and namespace of the authentication policy. If the policy is the mesh-wide policy, namespace is blank, as in this case: default/
- DESTINATION RULE: the name and namespace of the destination rule used.

충돌(conflicts)이 있는 경우를 설명하기 위해 잘못된 TLS 모드로 httpbin에 대한 서비스 별 대상 규칙(service-specific destination)을 추가하세요.

```
    $ cat <<EOF | kubectl apply -f -
    apiVersion: "networking.istio.io/v1alpha3"
    kind: "DestinationRule"
    metadata:
      name: "bad-rule"
      namespace: "default"
    spec:
      host: "httpbin.default.svc.cluster.local"
      trafficPolicy:
        tls:
          mode: DISABLE
    EOF
```

위와 같은 istioctl 명령을 실행하면 서버가 mTLS에 있는 동안 클라이언트가 HTTP 모드에 있으므로 CONFLICT 상태가 표시됩니다.

```
    $ i authn tls-check ${SLEEP_POD} httpbin.default.svc.cluster.local
    HOST:PORT                                 STATUS       SERVER     CLIENT
AUTHN POLICY        DESTINATION RULE
    httpbin.default.svc.cluster.local:8000     CONFLICT     mTLS       HTTP
default/            bad-rule/default
```

You can also confirm that requests from sleep to httpbin are now failing: sleep에서 httpbin으로의 요청이 실패했음을 확인할 수도 있습니다 :

```
    $ kubectl exec $(kubectl get pod -l app=sleep -o jsonpath=
{.items..metadata.name}) -c sleep -- curl httpbin:8000/headers -o /dev/null -s -w
'%{http_code}\n'
    503
```

계속하기 전에 잘못된 대상 규칙(destination)을 제거하여 다음 명령을 사용하여 상호 TLS(mutual TLS)를 다시 작동 시키세요.

```
    $ kubectl delete destinationrule --ignore-not-found=true bad-rule
```

Verify requests

이 작업은 상호 TLS(mutual TLS)를 사용하는 서버가 다음과 같은 요청에 대한 응답을 활성화하는 방법을 보여줍니다.

- In plain-text
- With TLS but without client certificate
- With TLS with a client certificate

이 작업을 수행하려면 client proxy를 by-pass해야합니다. 가장 간단한 방법은 istio-proxy 컨테이너에서 요청을 보내는 것입니다.

1. 다음 명령을 사용하여 httpbin과 통신하려면 TLS가 필요하므로 일반 텍스트(plain-text) 요청이 실패했는지 확인하세요.

    ```
        $ kubectl exec $(kubectl get pod -l app=sleep -o jsonpath=
    ```

```
{.items..metadata.name}) -c istio-proxy -- curl http://httpbin:8000/headers
-o /dev/null -s -w '%{http_code}\n'
    000
    command terminated with exit code 56
```

> 종료 코드는 56입니다.이 코드는 네트워크 데이터를 수신하지 못하는 것으로 해석됩니다.

> curl 응답코드 네트워크 데이터 수신 실패. 네트워크를 통해 데이터를 수신하는 것은 대부분의 curl 조작에서 중요한 부분이며 curl이 가장 낮은 네트워킹 계층에서 데이터 수신에 실패한 오류가 발생하면이 종료 상태가 리턴됩니다. 왜 이런 일이 발생했는지 정확하게 알기 위해 보통 심각한 파기가 필요합니다. 자세한 모드를 활성화하고, 추적하고 가능하다면 Wireshark 또는 이와 유사한 도구를 사용하여 네트워크 트래픽을 확인하세요.

2. 클라이언트 인증서가없는 TLS 요청도 실패하는지 확인합니다.

```
    $ kubectl exec $(kubectl get pod -l app=sleep -o jsonpath=
{.items..metadata.name}) -c istio-proxy -- curl https://httpbin:8000/headers
-o /dev/null -s -w '%{http_code}\n' -k
    000
    command terminated with exit code 35
```

> 이번에는 종료 코드가 35이며 SSL/TLS handshake 어딘가에서 발생하는 문제에 해당합니다.

> curl 응답코드 TLS / SSL 연결 오류입니다. SSL 핸드 셰이크가 실패했습니다. 여러 가지 이유로 SSL 핸드 셰이크가 실패 할 수 있으므로 오류 메시지가 몇 가지 추가적인 단서를 제공 할 수 있습니다. 당사자들이 SSL / TLS 버전, 쾌적한 암호 모음 또는 이와 유사한 것에 동의하지 않았을 수 있습니다.

3. Confirm TLS request with client certificate succeed:

```
    $ kubectl exec $(kubectl get pod -l app=sleep -o jsonpath=
{.items..metadata.name}) -c istio-proxy -- curl https://httpbin:8000/headers
-o /dev/null -s -w '%{http_code}\n' --key /etc/certs/key.pem --cert
/etc/certs/cert-chain.pem --cacert /etc/certs/root-cert.pem -k
    200
```

> Istio는 서비스 이름보다 Kubernetes 서비스 계정(service accounts)을 사용하여 서비스 이름(service name)보다 강력한 보안을 제공합니다 (자세한 내용은 Istio ID 참조). 따라서 Istio에서 사용하는 인증서에는 서비스 이름이 없습니다.이 서비스 이름은 curl이 서버 ID(server identity)를 확인하는 데 필요한 정보입니다. curl 클라이언트가 중단되지 않도록 curl을 -k 옵션과 함께 사용합니다. 이 옵션을 사용하면 클라이언트가 서버에서 제공하는 인증서의 서버 이름 (예 : httpbin.default.svc.cluster.local)을 확인 및 찾지 못합니다.

Cleanup

```
$ kubectl delete --ignore-not-found=true -f samples/httpbin/httpbin.yaml
$ kubectl delete --ignore-not-found=true -f samples/sleep/sleep.yaml
```

Plugging in External CA Key and Certificate

Citadel Health Checking

Provisioning Identity through SDS

Mutual TLS Migration

Mutual TLS over HTTPS