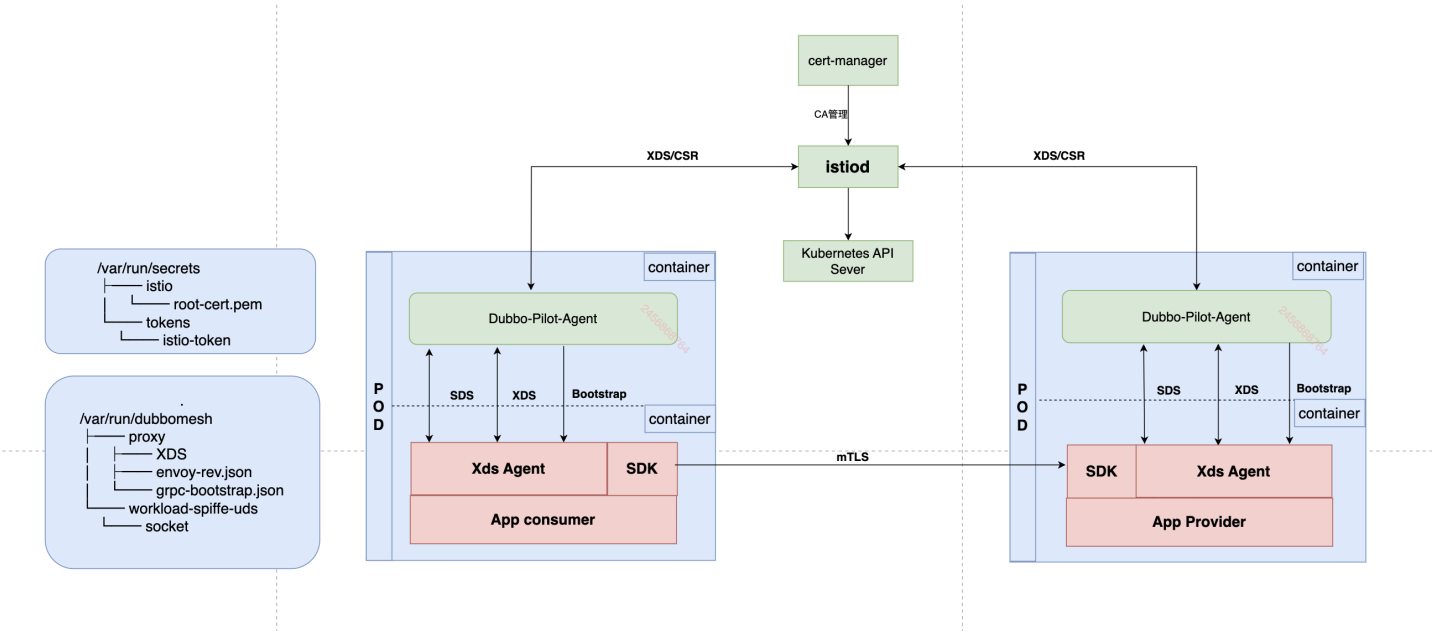


Dubbo零信任安全机制

一、整体架构

Dubbo 零信任安全机制整体框架是 istiod 作为控制面，istiod 控制面通常负责安全策略、证书等的管理，控制面负责与基础设施如 Kubernetes API Server 交互，将配置数据下发给 Dubbo 数据面组件。Dubbo 作为数据面适配 istiod 控制面 API。具体框架如下：



1. istiod 控制面:

- istiod 是 Istio 服务网格的控制面组件，负责管理安全策略、证书和其他配置。
- 它与基础设施组件（如 Kubernetes API Server）交互，将配置数据下发给 Dubbo 数据面组件。

2. Dubbo 数据面:

- 数据面包括 dubbo pilot agent 和集成了 Dubbo SDK 的业务应用程序，它们运行在同一个 Pod 中的不同容器里。
- 这两个容器通过共享卷 `/var/run/dubbomesh` 上的本地套接字进行通信。

3. Dubbo Pilot Agent:

- 在启动时创建私钥和证书签名请求（CSR），然后将 CSR 发送给 istiod 进行签名。
- 负责与 istiod 进行安全认证，并监控工作负载证书的过期和重新签发。
- 代理 XDS 服务和提供 SDS gRPC 服务。
- 生成 Dubbo SDK 的 bootstrap 配置。

4. Xds Agent:

- 解析由 dubbo pilot agent 生成的 Bootstrap 配置，并与 SDS 和 XDS 建立连接。
- 解析接收到的 LDS（监听器发现服务）、RDS（路由发现服务）、CDS（集群发现服务）、EDS（端点发现服务）和 SDS 资源。
- 为 Dubbo SDK 提供 TLS 管理、对等认证（Peer Authentication）、请求认证（Request Authentication）和授权（Authorization）能力。
- 为 Dubbo SDK 提供服务发现能力。

这个框架通过在 Dubbo 中集成 Istio 的控制面和数据面组件，实现了零信任安全原则，包括但不限于加密通信、身份验证和授权。这样，Dubbo 应用程序可以在服务网格环境中安全地通信和服务发现，同时保持了 Dubbo 的高性能和灵活性。

二、Istio 控制面 三大 CRD

1. 对等认证 PeerAuthentication

双向 TLS 认证策略

1. Mesh/Namespaced/Workload 三种策略目标
2. PERMISSIVE/STRICT/DISABLE 三种工作模式
3. 保存身份信息到 source.principal

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-peer-policy"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: reviews
  mtls:
    mode: STRICT
```

2. 请求认证 RequestAuthentication

基于请求令牌认证策略

1. Mesh/Namespace/Workload 三种策略目标
2. token 的位置
3. issuer 和 audiences
4. JSON Web Key Set (JWKS)
5. 保存身份信息到 request.auth.principal

```
apiVersion: "security.istio.io/v1beta1"
kind: "RequestAuthentication"
metadata:
  name: "httpbin-demo1"
  namespace: foo
spec:
  jwtRules:
    - issuer: "https://dubbo.apache.org/"
      audiences:
        - dev
      fromHeaders:
        - name: "Authorization"
          prefix: "Bearer "
      jwks: |
        {
          "keys": [
            {
              ...
            }
          ]
        }
      }
```

3. 授权 AuthorizationPolicy

授权策略

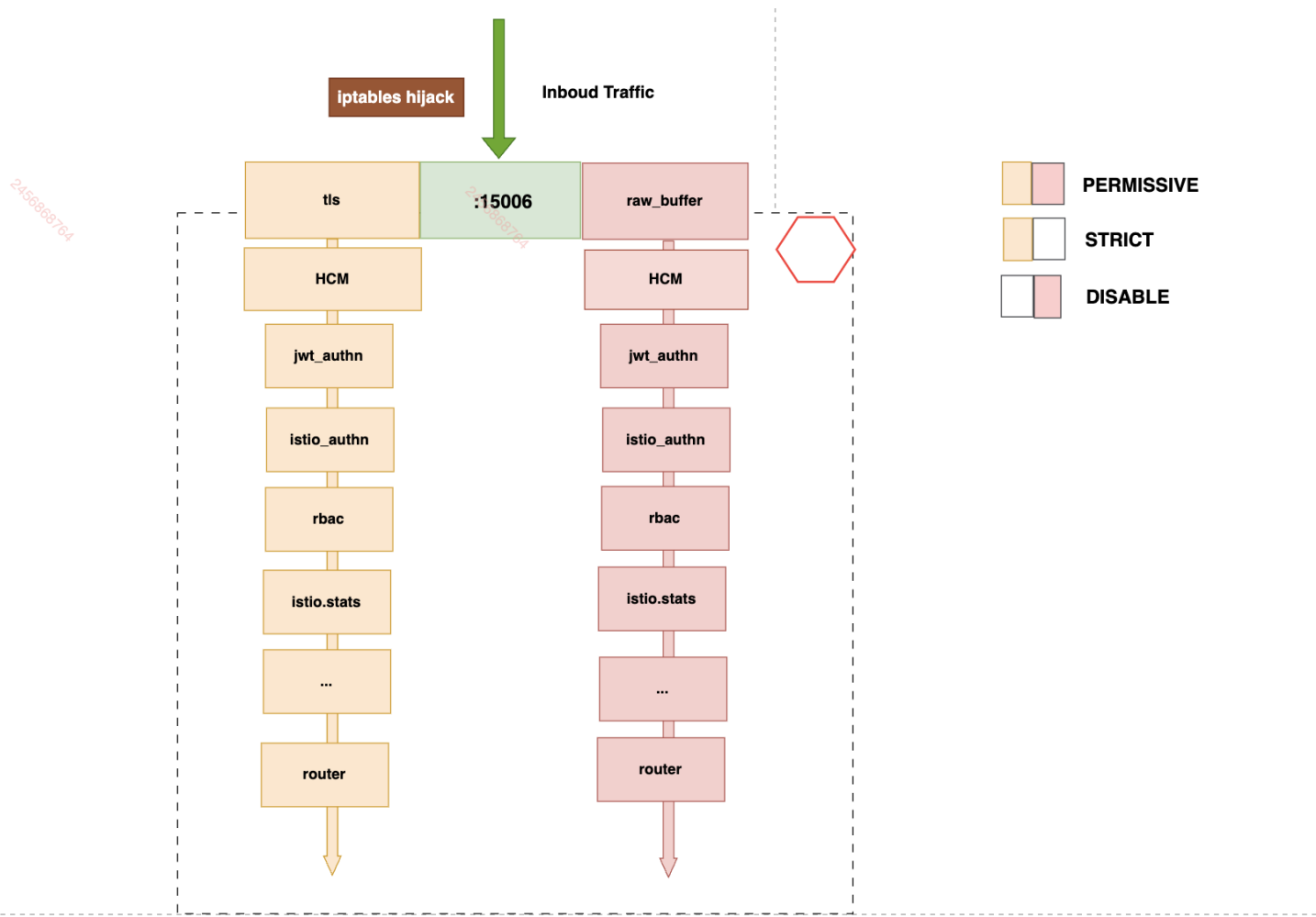
1. Mesh/Namespace/Workload 三种策略目标
2. action 指定ALLOW、DENY
3. rules 指定何时触发动作
 - from 指定请求的来源
 - to 指定请求的操作
 - when 指定应用规则所需的条件

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
  action: ALLOW
  rules:
    - from:
        - source:
            principals: ["cluster.local/ns/default/sa/sleep"]
      - source:
            namespaces: ["dev"]
      to:
        - operation:
            methods: ["GET"]
      when:
        - key: request.auth.claims[iss]
          values: ["https://accounts.google.com"]
```

istio文档: <https://istio.io/latest/zh/docs/concepts/security/#authorization>

三、Envoy 数据面实现原理

1. Downstream virtualInbound 配置



```
listener:
  name: virtualInbound
  address:
    socketAddress:
      address: 0.0.0.0
      portValue: 15006
  filterChains:
    - name: virtualInbound-catchall-http
      filterChainMatch:
        transportProtocol: tls
      transportSocket: {...}
      filters:
        - name: envoy.filters.network.http_connection_manager
          typedConfig:
            routeConfig: {}
            httpFilters:
              - name: envoy.filters.http.jwt_authn
              - name: istio_authn
              - name: envoy.filters.http.rbac
              - name: ...
              - name: istio.stats
              - name: envoy.filters.http.router
    - name: virtualInbound-catchall-http
      filterChainMatch:
        prefix_ranges: {...}
        transport_protocol: raw_buffer
      transportSocket: {...}
      filters:
        - name: envoy.filters.network.http_connection_manager
        ...
```

```
transport_socket:
  name: "envoy.transport_sockets.tls"
  typed_config:
    "@type":
      "type.googleapis.com/envoy.extensions.transport_sockets.tls.v3.DownstreamTlsContext"
    common_tls_context:
      tls_certificate_sds_secret_configs:
        - name: "default"
          sds_config:
            api_config_source:
              api_type: "GRPC"
            grpc_services:
              - envoy_grpc:
                  cluster_name: "sds-grpc"
      combined_validation_context:
        default_validation_context:
          match_subject_alt_names:
            - prefix: "spiffe://cluster.local/"
      validation_context_sds_secret_config:
        name: "ROOTCA"
        sds_config:
          api_config_source:
            api_type: "GRPC"
          grpc_services:
            - envoy_grpc:
                cluster_name: "sds-grpc"
      require_client_certificate: true
```

2. Upstream Cluster 配置

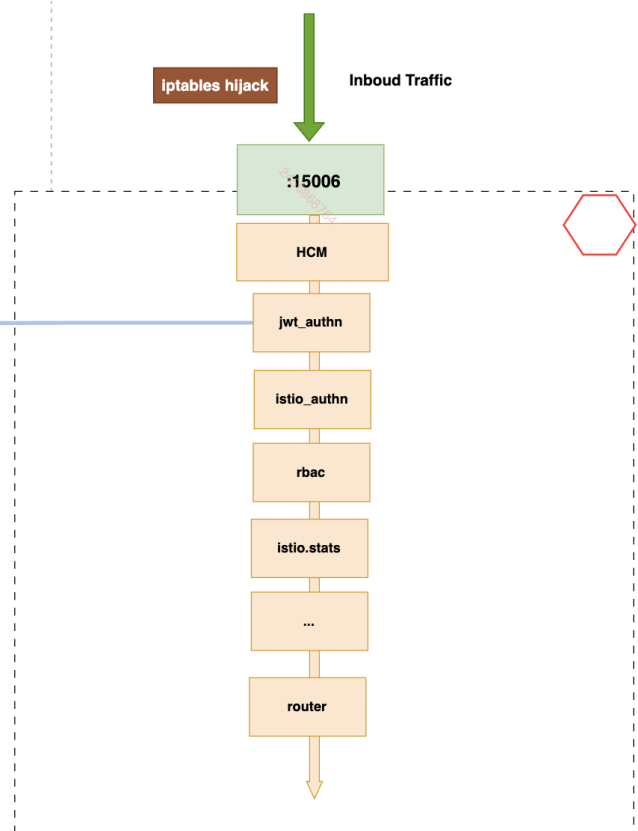
```
cluster:
  "@type": "type.googleapis.com/envoy.config.cluster.v3.Cluster"
  name: "outbound18000l1httpbin.foo.svc.cluster.local"
  type: EDS
  eds_cluster_config: {...}
  circuit_breakers: {...}
  ...
  transport_socket_matches:
    - name: "tlsMode-istio"
      match:
        tlsMode: "istio"
      transport_socket:
        name: "envoy.transport_sockets.tls"
        typed_config:
          "@type":
            "type.googleapis.com/envoy.extensions.transport_sockets.tls.v3.UpstreamTlsContext"
            common_tls_context: {...}
            - name: "tlsMode-disabled"
              match: {}
            transport_socket:
              name: "envoy.transport_sockets.raw_buffer"
              typed_config:
                "@type":
                  "type.googleapis.com/envoy.extensions.transport_sockets.raw_buffer.v3.RawBuffer"
```

```
common_tls_context:
  tls_certificate_sds_secret_configs:
    - name: "default"
      sds_config:
        api_config_source:
          api_type: "GRPC"
        grpc_services:
          - envoy_grpc:
              cluster_name: "sds-grpc"
  combined_validation_context:
  default_validation_context:
    match_subject_alt_names:
      - exact: "spiffe://cluster.local/ns/foo/sa/httpbin"
  validation_context_sds_secret_config:
    name: "ROOTCA"
    sds_config:
      api_config_source:
        api_type: "GRPC"
      grpc_services:
        - envoy_grpc:
            cluster_name: "sds-grpc"
```

3. envoy.filters.http.jwt_authn

```
name: envoy.filters.http.jwt_authn
providers:
  origins-0:
    issuer: http://dubbo.apache.org
    audiences:
      - dev
      - test
  fromHeaders:
    - name: "Authorization"
      prefix: "Bearer "
  localJwks:
    inlineString: "..."
rules:
  - match:
      prefix: "/"
    requires:
      requiresAny:
        requirements:
          - providerName: origins-0
          - allowMissing: {}
```

1. 允许 Token 为空的请求
2. 有Token 如果校验失败, 就拒绝请求
3. 在授权时候可以拒绝没有经过Token认证的请求



4. istio_authn

```

name: istio_authn
policy:
  peers:
    - mtls: {}
  origins:
    - jwt:
        issuer: http://dubbo.apache.org
        originsOptional: true
        principalBinding: USE_ORIGIN
        skipValidateTrustDomain: true

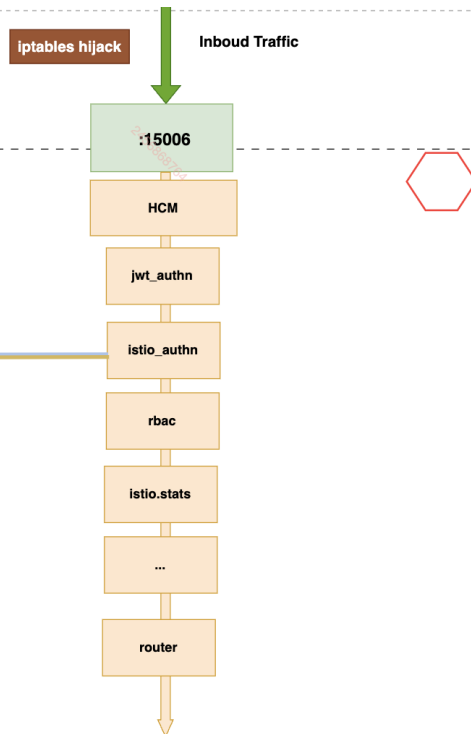
```

Envoy dynamic metadata

```

istio_authn:
  request.auth.principal: "http://dubbo.apache.org"
  request.auth.claims:
    iss: "http://dubbo.apache.org"
  aud:
    - dev
    - test
  sub: "john"
  iat: 1516239022

```

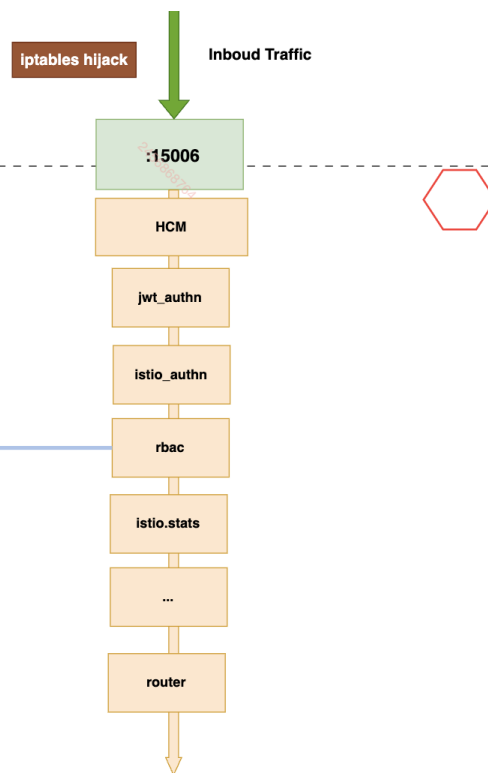


5. envoy.filters.http.rbac

```

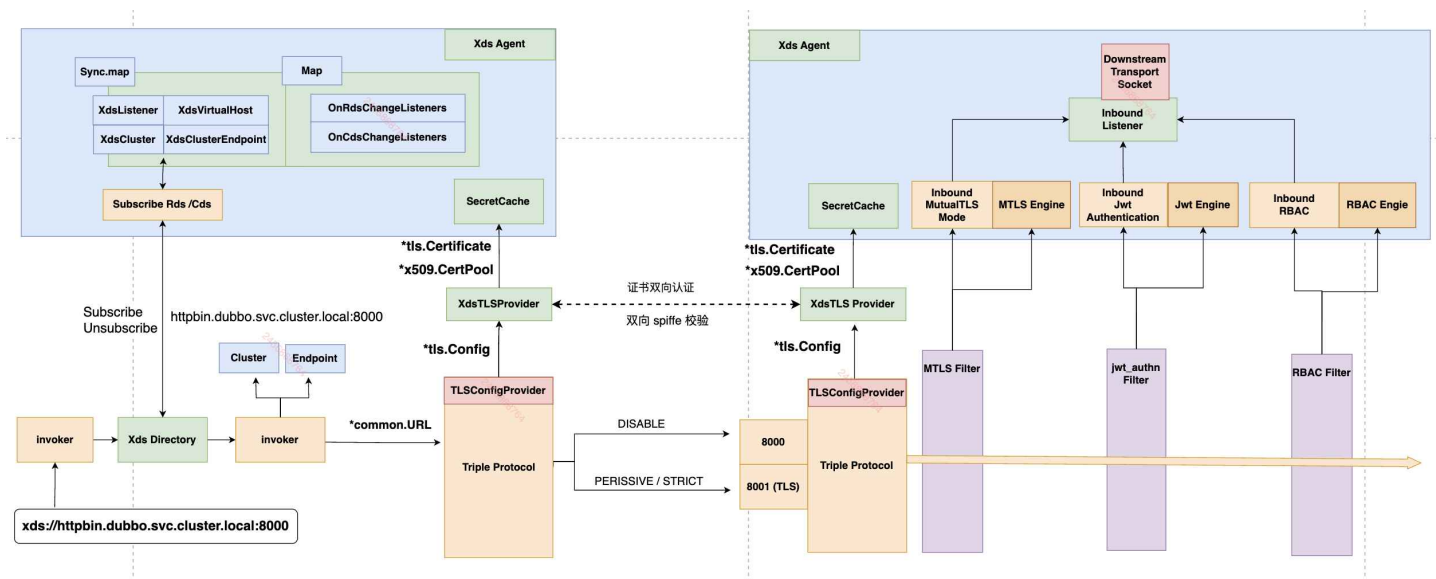
name: envoy.filters.http.rbac
rules:
  policies:
    ns[foo]-policy[httpbin-admin]-rule[0]:
      permissions:
        - andRules:
            rules:
              - orRules:
                  rules:
                    - header:
                        name: ":method"
                        exactMatch: GET
      principals:
        - andIds:
            ids:
              - orIds:
                  ids:
                    - metadata:
                        filter: istio_authn
                        path:
                          - key: request.auth.principal
                          value:
                            stringMatch:
                              exact: http://dubbo.apache.org

```



四、Dubbo SDK 数据面实现原理

整体框架如下：



1. 实现 mTLS 和 PeerAuthentication

具体步骤如下：

1. 实现 Tri Protocol TLS：

- 在 Tri Protocol Export 时，为其注入 TLS 配置提供者（TLS Config Provider）。
- 同时启动 HTTP 服务器和 HTTPS 服务器，其中 HTTPS 服务器监听的端口是 HTTP 服务器端口号加一。

2. Provider 端配置：

- 通过 Xds TLS Provider 中 `GetServerWorkLoadTLSConfig` 函数获取服务器工作负载的 `*tls.Config` 配置。
- 配置包括获取工作负载证书、CA 证书池，并设置客户端证书的验证逻辑。
- 在 `VerifyPeerCertByServer` 函数中，同时验证客户端证书是否在服务端的 ROOT CA 证书链中，以及客户端证书中的 SPIFFE URI 是否匹配预期的值。

3. Consumer 端配置：

- 在 Xds directory 中，为每个 `XdsCluster` 中的每个 `XdsClusterEndpoint` 生成对应 Dubbo 通用 URL（common.URL）。
- 根据 `XdsCluster` 的 mTLS 模式设置端口号：如果是 `DISABLE`，则使用端点的实际端口号；如果是 `STRICT` 或 `PERMISSIVE`，则端口号是端点对应端口号加一。
- 将 XDS Cluster Upstream transport socket 中的 SPIFFE 认证方式添加到 URL 中，并传递到 Tri Protocol 的 refer 方法里。
- 通过 Xds TLS Provider 中 `GetClientWorkLoadTLSConfig` 函数获取客户端工作负载的 `*tls.Config` 配置。
- 配置包括获取工作负载证书、CA 证书池，并设置服务器证书的验证逻辑。

- 在 `VerifyPeerCertByClient` 函数中，同时验证服务端证书是否在客户端的 ROOT CA 证书链中，以及服务端证书中的 SPIFFE URI 是否匹配预期的值。

4. MTLS Filter：根据当前mTLS model 来判断是否允许请求HTTP流量和 HTTPS流量。

- STRICT模式：只允许 HTTPS 流量
- DISABLE模式：只允许 HTTP 流量
- PERMISSIVE模式：允许 HTTP 流量或者 HTTPS流量

5. Xds TLS Provider 中获取 `tls.Config` 方法如下：

```
1 func (x *xdsTLSProvider) GetServerWorkLoadTLSConfig(url *common.URL)
   (*tls.Config, error) {
2     cfg := &tls.Config{
3         GetCertificate: x.GetWorkloadCertificate,
4         // ClientAuth:      tls.VerifyClientCertIfGiven, // for test only
5         ClientAuth: tls.RequireAndVerifyClientCert, // for prod
6         ClientCAs: x.GetCACertPool(),
7         VerifyPeerCertificate: func(rawCerts [][]byte, verifiedChains [][]
   []*x509.Certificate) error {
8             err := x.VerifyPeerCertByServer(rawCerts, verifiedChains)
9             if err != nil {
10                 logger.Errorf("Could not verify client certificate: %v", err)
11             }
12             return err
13         },
14         MinVersion:          tls.VersionTLS12,
15         CipherSuites:         tlsprovider.PreferredDefaultCipherSuites(),
16         NextProtos:           []string{"h2", "http/1.1"},
17         PreferServerCipherSuites: true,
18     }
19
20     return cfg, nil
21 }
22
23 func (x *xdsTLSProvider) GetClientWorkLoadTLSConfig(url *common.URL)
   (*tls.Config, error) {
24
25     verifyMap := make(map[string]string, 0)
26     verifyMap["SubjectAltNamesMatch"] =
   url.GetParam(constant.TLSSubjectAltNamesMatchKey, "")
27     verifyMap["SubjectAltNamesValue"] =
   url.GetParam(constant.TLSSubjectAltNamesValueKey, "")
28
29     cfg := &tls.Config{
30         GetCertificate:      x.GetWorkloadCertificate,
```



```

31     InsecureSkipVerify: true,
32     RootCAs:           x.GetCACertPool(),
33     VerifyPeerCertificate: func(rawCerts [][]byte, verifiedChains [][]*x509.Certificate) error {
34         certVerifyMap := verifyMap
35         err := x.VerifyPeerCertByClient(rawCerts, verifiedChains,
36             certVerifyMap)
37         if err != nil {
38             logger.Errorf("Could not verify server certificate: %v", err)
39         }
40         return err
41     },
42     MinVersion:          tls.VersionTLS12,
43     CipherSuites:        tlsprovider.PreferredDefaultCipherSuites(),
44     NextProtos:          []string{"h2", "http/1.1"},
45     PreferServerCipherSuites: true,
46 }
47 return cfg, nil
48 }
49 }

```

2. 实现 RequestAuthentication

基于 Dubbo Filter 机制构建 jwt_authn Filter。具体步骤和示例代码：

1. **获取 JwtAuthentication 配置**：从控制面获取下发的 `JwtAuthentication` 配置，该配置包含了 JWT 的发行者、受众、密钥集等信息。
2. **构建 HTTP 请求头**：从 Dubbo 的 `Invocation` Attachments 中提取构建 HTTP 请求头。
3. **构建 Jwt Authn Engine**：根据 HTTP 请求头和 `JwtAuthentication` 配置，构建一个 `JwtAuthnEngine` 实例，该实例负责处理 JWT 的验证和解析。
4. **调用 Filter 方法**：调用 `JwtAuthnEngine` 的 `Filter` 方法来判断请求是否合法。如果请求非法，则拒绝请求并返回错误。
5. **处理合法请求**：如果请求合法，将 JWT 的声明（Claims）信息写入到 `Invocation` 的附件中，以便后续的 RBAC Filter 使用。继续执行后续的调用 `invoker.Invoke(ctx, invocation)`。

示例代码如下：

```

1 func (f *jwtAuthnFilter) Invoke(ctx context.Context, invoker protocol.Invoker,
  invocation protocol.Invocation) protocol.Result {
2     ...
3     // 获取控制面下发的 Jwt Authentication 配置
4     jwtAuthentication := f.pilotAgent.GetHostInboundJwtAuthentication()
5     if jwtAuthentication == nil {
6         logger.Info("[jwt authn filter] skip jwt authn because there is no jwt
  authentication found.")
7         return invoker.Invoke(ctx, invocation)
8     }
9     // 从 Invocation Attachments 构建 Http 请求头
10    headers := buildRequestHeadersFromCtx(ctx, invoker, invocation)
11    // 构建 Jwt Authn Engine
12    jwtAuthnFilterEngine :=
  istioengine.NewJwtAuthnFilterEngine(jwtAuthentication)
13    // Jwt Authn Engine 判断请求是否拒绝
14    jwtAuthnResult, err := jwtAuthnFilterEngine.Filter(headers)
15    if err != nil {
16        result := &protocol.RPCResult{}
17        result.SetResult(nil)
18        result.SetError(err)
19        return result
20    }
21
22    jwtVerifyStatus := jwtAuthnResult.JwtVerfiyStatus
23    // 如果请求非法, 就拒绝请求返回
24    if jwtVerifyStatus == istioengine.JwtVerfiyStatusMissing {
25        result := &protocol.RPCResult{}
26        result.SetResult(nil)
27        result.SetError(errors.New("jwt token is missing"))
28        return result
29    }
30
31    if jwtVerifyStatus == istioengine.JwtVerfiyStatusFailed {
32        result := &protocol.RPCResult{}
33        result.SetResult(nil)
34        result.SetError(errors.New("jwt token verify fail"))
35        return result
36    }
37
38    // 把 jwt Token claims 写入到 invocation attachments 中
39    findProviderName := jwtAuthnResult.FindProviderName
40    findHeaderName := jwtAuthnResult.FindHeaderName
41    jwtToken := jwtAuthnResult.JwtToken
42    providers := jwtAuthentication.Providers
43    if len(findProviderName) > 0 && jwtToken != nil {
44        ...

```

```

45      // 转换 jwt Token 到 jwt claims json
46      if jwtJsonClaims, err :=
resources.ConvertJwtTokenToJwtClaimsJson(jwtToken); err != nil {
47          logger.Errorf("[jwt authn filter] can not convert from jwt token to
jwt claims, err:%v", err)
48      } else {
49          ...
50          // add new attachment named :x-auth
51          logger.Infof("[jwt authn filter] add attachment k: %s, v: %s",
constant.HttpHeaderXJwtClaimsName, jwtJsonClaims)
52          invocation.SetAttachment(constant.HttpHeaderXJwtClaimsName,
[]string{jwtJsonClaims})
53      }
54  }
55
56  return invoker.Invoke(ctx, invocation)
57 }

```

3. 实现 AuthorizationPolicy

基于 Dubbo Filter 机制构建 RBAC Filter。具体步骤和示例代码：

1. **获取 RBAC 配置：** 从控制面获取下发的 `AuthorizationPolicy` 配置，该配置包含了访问控制策略和角色定义。
2. **构建 HTTP 请求头：** 从 Dubbo 的 `Invocation` Attachment 中提取构建 HTTP 请求头。
3. **构建 RBAC Engine：** 根据 HTTP 请求头和 `AuthorizationPolicy` 配置，构建一个 `RBACEngine` 实例，该实例负责处理访问控制的决策。
4. **调用 Filter 方法：** 调用 `RBACEngine` 的 `Filter` 方法来判断请求是否合法。如果请求非法，则拒绝请求并返回错误。
5. **处理合法请求：** 如果请求合法，继续执行后续的调用 `invoker.Invoke(ctx, invocation)`。

示例代码如下：

```

1 func (f *rbacFilter) Invoke(ctx context.Context, invoker protocol.Invoker,
invocation protocol.Invocation) protocol.Result {
2     ...
3     // 获取控制面下发 RBAC 配置
4     v3RBAC := f.pilotAgent.GetHostInboundRBAC()
5     if v3RBAC == nil {

```

```

6      // there is no jwt authn filter
7      logger.Info("[rbac filter] skip rbac filter because there is no rbac
configuration found.")
8      return invoker.Invoke(ctx, invocation)
9  }
10
11  // 从 Invocation Attachments 构建 Http 请求头
12  headers := buildRequestHeadersFromCtx(ctx, invoker, invocation)
13  // 构建 RBAC Engine
14  rbacFilterEngine := istioengine.NewRBACFilterEngine(v3RBAC)
15  // RBAC Engine 判断请求是否拒绝
16  rbacResult, err := rbacFilterEngine.Filter(headers)
17  // 如果请求非法, 就拒绝请求返回
18  if err != nil {
19      result := &protocol.RPCResult{}
20      result.SetResult(nil)
21      result.SetError(err)
22      return result
23  }
24  // 继续执行后续 invoke
25  logger.Infof("[rbac filter] rbac result: %s",
utils.ConvertJsonString(rbacResult))
26  return invoker.Invoke(ctx, invocation)
27 }

```

五、Xds Agent 工作原理

Xds Agent 提供对外接口能力如下:

```

1
2 // PilotAgentType represents the type of Pilot agent, either server workload
or client workload.
3 type PilotAgentType int32
4 const (
5     PilotAgentTypeServerWorkload PilotAgentType = iota
6     PilotAgentTypeClientWorkload
7 )
8
9 // OnRdsChangeListener defines the signature for RDS change listeners.
10 type OnRdsChangeListener func(serviceName string, xdsVirtualHost
resources.XdsVirtualHost) error
11 // OnEdsChangeListener defines the signature for EDS change listeners.
12 type OnEdsChangeListener func(clusterName string, xdsCluster
resources.XdsCluster, xdsClusterEndpoint resources.XdsClusterEndpoint) error
13

```

```

14 // XdsAgent is the interface for managing xDS agents.
15 type XdsAgent interface {
16     // Run starts the xDS agent with the specified Pilot agent type.
17     Run(pilotAgentType PilotAgentType) error
18     // GetWorkloadCertificateProvider returns the provider for workload
    certificates.
19     GetWorkloadCertificateProvider() WorkloadCertificateProvider
20     // SubscribeRds subscribes to changes in the Route Discovery Service (RDS)
    for the given service name.
21     SubscribeRds(serviceName, listenerName string, listener OnRdsChangeListener)
22     // UnsubscribeRds unsubscribes the listener from changes in the Route
    Discovery Service (RDS) for the given service name.
23     UnsubscribeRds(serviceName, listenerName string)
24     // SubscribeCds subscribes to changes in the Endpoint Discovery Service
    (EDS) for the given cluster name.
25     SubscribeCds(clusterName, listenerName string, listener OnEdsChangeListener)
26     // UnsubscribeCds unsubscribes the listener from changes in the Endpoint
    Discovery Service (EDS) for the given cluster name.
27     UnsubscribeCds(clusterName, listenerName string)
28     // GetHostInboundListener returns the configuration for the host inbound
    listener.
29     GetHostInboundListener() *resources.XdsHostInboundListener
30     // GetHostInboundMutualTLSMode returns the mutual TLS mode for host inbound
    connections.
31     GetHostInboundMutualTLSMode() resources.MutualTLSMode
32     // GetHostInboundJwtAuthentication returns the JWT authentication
    configuration for host inbound connections.
33     GetHostInboundJwtAuthentication() *resources.JwtAuthentication
34     // GetHostInboundRBAC returns the RBAC (Role-Based Access Control)
    configuration for host inbound connections.
35     GetHostInboundRBAC() *rbacv3.RBAC
36     // Stop stops the xDS agent.
37     Stop()
38 }
39
40 // WorkloadCertificateProvider is the interface for providing workload
    certificates.
41 type WorkloadCertificateProvider interface {
42     // GetWorkloadCertificate returns the TLS certificate for the given
    ClientHelloInfo.
43     // This method is responsible for providing certificates based on the
    information
44     // available in the ClientHelloInfo, such as SNI (Server Name Indication).
45     GetWorkloadCertificate(*tls.ClientHelloInfo) (*tls.Certificate, error)
46     // GetCACertPool returns the root CA certificate pool.
47     // This method is responsible for providing the root CA certificate pool
48     // which is used for verifying the authenticity of certificates presented

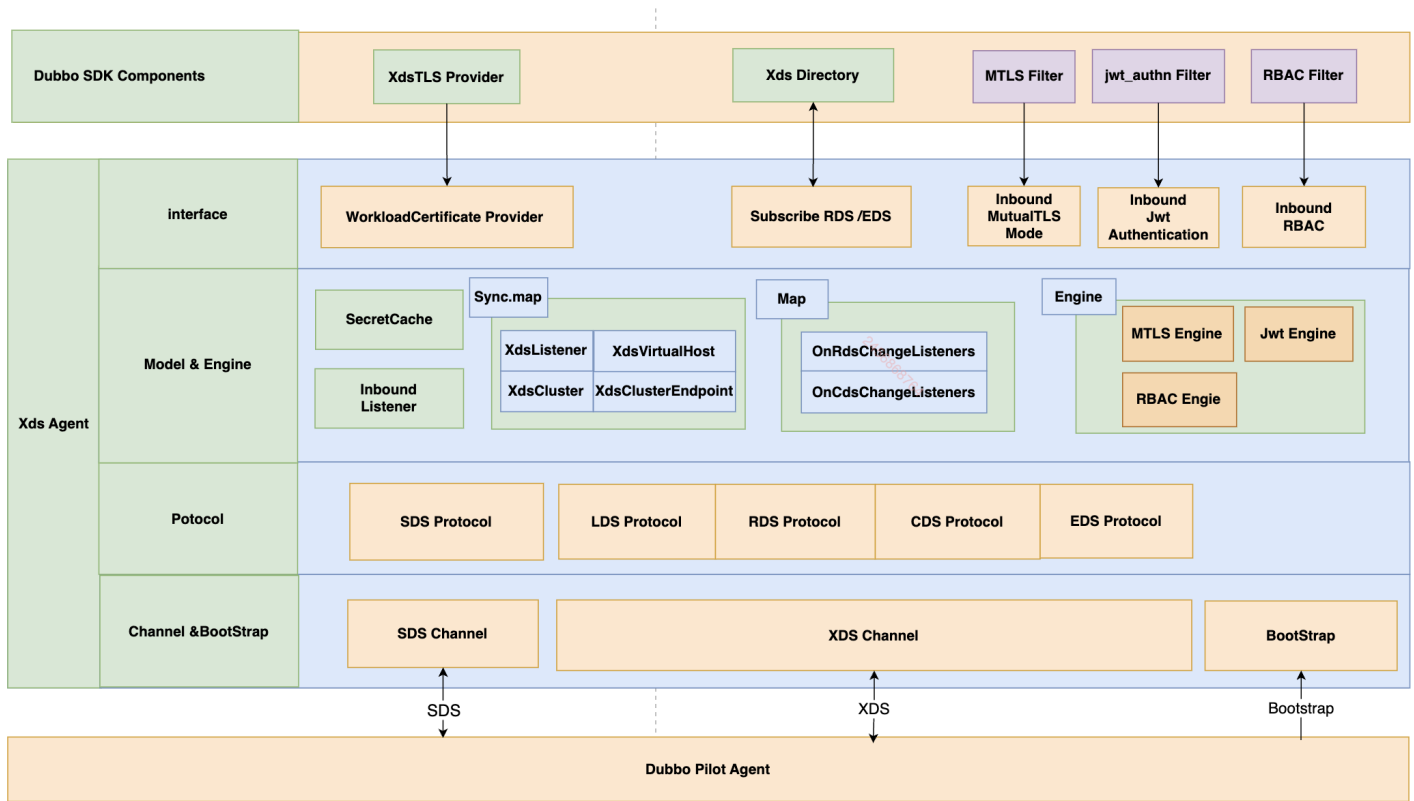
```

```

49 // by peers during mutual TLS handshake.
50 GetCACertPool() (*x509.CertPool, error)
51 }

```

具体Xds Agent 组件如下：



1. Channel & Bootstrap 层

- **Bootstrap:** 负责解析bootstrap配置，包含节点信息、服务发现服务（SDS）和扩展服务发现（XDS）套接字地址。
- **Channel:** 负责与 SDS 和 XDS 套接字建立连接，并在连接失败时进行重试。它还负责发送和接收请求，并处理协议的消息订阅，一旦接收到新数据，就会调用相应的订阅回调函数

2. Protocol 层

- 该层接收来自 Channel层的 订阅回调，并解析 Envoy 的监听器（Listener）、路由（Route）、集群（Cluster）和端点（Endpoint）配置。解析完成后，它通过 go channel 将这些数据发送给 Xds Agent 的模型层。

3. Model & Engine 层

- 这一层通过 LDS（监听器发现服务）、RDS（路由发现服务）、CDS（集群发现服务）和 EDS（端点发现服务）协议解析数据，并将这些数据保存在同步映射（Sync.map）中。
- SDS 解析后的 workload 证书和根 CA 被保存在 SecretCache 中。

- 如果通过 LDS 解析发现是 Envoy 虚拟入口监听器（VirtualInboundListener），则同时将其保存到入口监听器（InboundListener）中。
- Engine 层包含三个引擎：MTLS 引擎、JWT 引擎和 RBAC 引擎。这些引擎根据控制面下发的配置信息（PeerAuthentication、RequestAuthentication、AuthorizationPolicy）和当前请求的 HTTP 头信息来判断请求是否合法。

4. Interface 层

- 这一层实现了 Xds Agent 对外提供的接口，包括获取证书、订阅和取消订阅 RDS/EDS、获取当前由控制面下发的 MTLS 模式、JWT 和 RBAC 配置等。

六、最终如何使用

1. Provider

```
package main

import ...

type GreetTripleServer struct { 2 usages
}

func (srv *GreetTripleServer) Greet(ctx context.Context, req *greet.GreetRequest) (*greet.GreetResponse, error) {
    resp := &greet.GreetResponse{Greeting: req.Name}
    return resp, nil
}

func main() {
    srv, err := server.NewServer(
        server.WithServerProtocol(
            protocol.WithPort(port: 8000),
            protocol.WithTriple(),
            protocol.WithTlsProvider(tlsProvider: "xds-provider"),
        ),
        server.WithServerXds(),
    )
    if err != nil : err *

    if err := greet.RegisterGreetServiceHandler(srv, &GreetTripleServer{}); err != nil : err *

    if err := srv.Serve(); err != nil {
        logger.Error(err)
    }
}
```

2. Consumer

```

package main

import ...

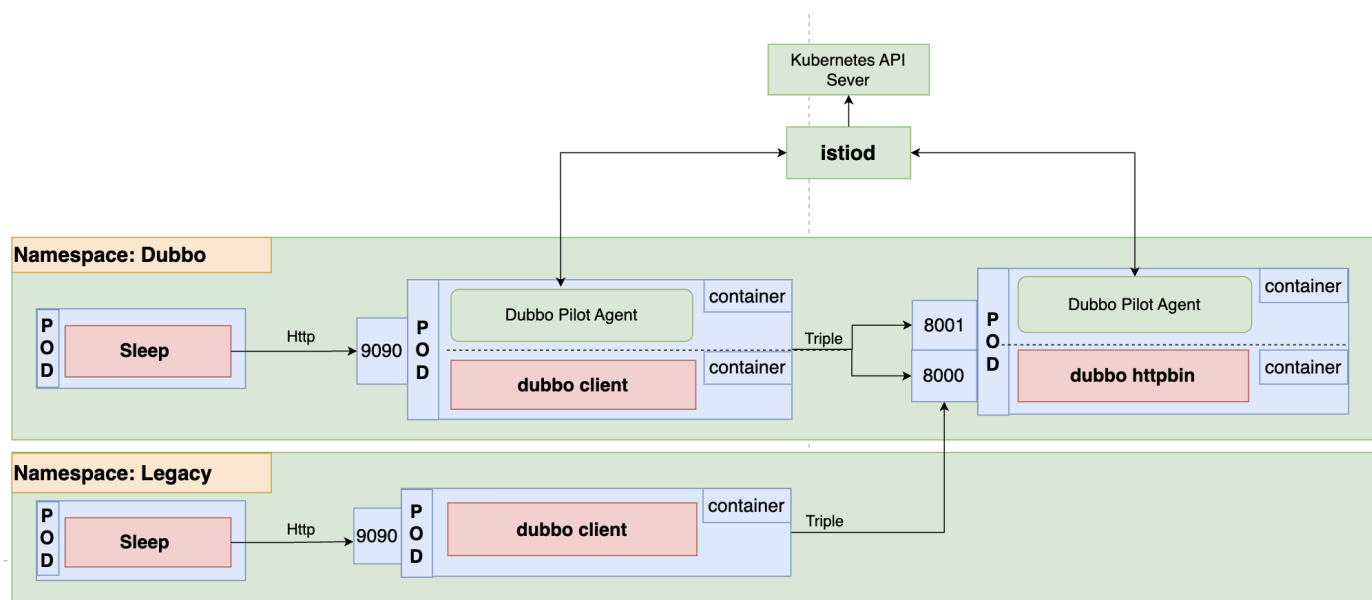
func main() {
    cli, err := client.NewClient(
        client.WithClientURL(url: "xds://httpbin.dubbo.svc.cluster.local:8000"),
    )
    if err != nil { err }

    svc, err := greet.NewGreetService(cli)
    if err != nil { err }

    resp, err := svc.Greet(context.Background(), &greet.GreetRequest{Name: "hello world"})
    if err != nil {
        logger.Error(err)
    }
    logger.Infof(fmt: "Greet response: %s", resp.Greeting)
}

```

3. 测试环境



测试环境说明: <https://github.com/2456868764/dubbo-mesh/blob/main/deploy/httpbin/README.md>

七、项目代码和进度

1. 项目代码

- dubbo-go-sdk PR: <https://github.com/apache/dubbo-go/pull/2643>
- dubbo-pilot-agent 和 测试代码: <https://github.com/2456868764/dubbo-mesh>
- 测试环境: <https://github.com/2456868764/dubbo-mesh/blob/main/deploy/httpbin/README.md>

2. 项目进度

- Dubbo-pliot-agent : 完成
- Dubbo-sdk
 - xds-agent: 完成
 - mtls: 完成
 - mtls filter: 完成
 - jwt_authn filter: 完成
 - RBAC filter: 开发中
 - RBAC Model 定义: 开发中
 - RBAC pb 解析
 - RBAC engine
 - Isito 部署测试功能 samples: 完成
 - 技术文档和PPT: 完成