Ко всем новостям

Опубликовано: 20.08.2025

Как мы строили безопасную микросервисную архитектуру с Service Mesh: взгляд изнутри

Публикации в СМИ Опубликовано на Хабре

Привет! Меня зовут Валентин Вертелецкий, я DevOps в СберТехе, занимаюсь развитием Platform V Kintsugi — это графическая консоль для сопровождения Postgres-like СУБД. Наш продукт построен на микросервисной архитектуре и сначала разрабатывался с использованием базовой функциональности Kubernetes — там нет встроенных механизмов аутентификации, авторизации, управления доступом и шифрования трафика. Когда же у нас стало больше сервисов, нам понадобилось повысить защиту и отказоустойчивость, добавить возможности управления доступом.

Мы опираемся на подход Zero Trust: ни одному элементу системы не доверяем по умолчанию. Каждый запрос проверяется, привилегии для администраторов минимальны, трафик валидируется и шифруется. Нам предстояло обеспечить надёжную аутентификацию и авторизацию, а также централизованный контроль и мониторинг запросов. В этом нам помогла технология Service Mesh.

Для управления микросервисами в Kubernetes мы используем Platform V Synapse Service Mesh от СберТеха — это решение на основе платформы Istio. Покажу, как всё работает у нас. Плюс, я подготовил демо-проект для тестирования кейсов (ссылка в конце статьи). Надеюсь, он будет полезен командам, работающим с микросервисами.

Service Mesh (сервисная сеть) — это технология для управления сервисами в распределённых системах и микросервисной архитектуре. Её ключевые компоненты:

1. Sidecar Proxy: запускается вместе с каждым сервисом в виде отдельного контейнера (sidecar). Этот контейнер обрабатывает весь входящий и исходящий трафик сервиса, выполняя задачи, связанные с управлением сетью и безопасностью;
2. Control Plane: управляет прокси-сервисами, координирует их работу и отвечает за распределение конфигураций, политику безопасности, управление трафиком;
3. Data Plane: включает в себя сами прокси-сервисы, которые непосредственно занимаются обработкой трафика между сервисами.

Наши условия. Как я уже упомянул, наш продукт базируется на Platform V Synapse Service Mesh. Это Service Mesh от СберТеха, разработанный на основе Istio. Istio — open source-платформа для управления трафиком и защиты микросервисов в распределённых системах. В качестве проксирующего компонента используется Envoy — высокопроизводительный прокси с открытым исходным кодом.

Назначение Istio:

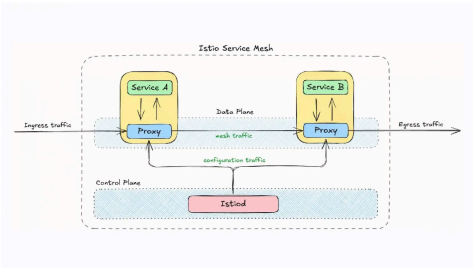
повышение безопасности: шифрование трафика, аутентификация, авторизация;

управление трафиком: маршрутизация запросов, балансировка нагрузки, отказоустойчивость;

наблюдаемость: мониторинг производительности сервисов, сбор метрик, трассировка, журналирование;

отказоустойчивость: механизм таймаутов, повторных попыток выполнения запросов.

Мы используем архитектуру Istio в режиме sidecar (sidecar mode), когда каждый сервис в облаке разворачивается вместе с контейнером экземпляра Envoy-прокси, который выступает посредником между приложением и внешним миром. Задача проксирующего узла — обработка входящего и исходящего сервисного трафика. Так в архитектуре внедряется дополнительный слой абстракции — он прозрачен для прикладного сервиса и позволяет решать инфраструктурные задачи, не вовлекая в этот процесс само приложение.



Для добавления sidecar-контейнера Envoy-прокси в под с приложением используется следующая инструкция в блоке `annotations`:

```
sidecar.istio.io/inject: 'true'
```

Основные типы ресурсов для конфигурирования Envoy, про которые расскажу:

- ServiceEntry: используется для регистрации сервисов в Istio service registry;
- VirtualService: управляет политиками определения маршрутов трафика к конкретному сервису; включает в себя правила маршрутизации, балансировки и другие политики обработки сервисного трафика;
- Gateway: определяет точку входа внешнего трафика в сервисную сеть и указывает порты и протоколы, которые будут использоваться для передачи трафика;
- DestinationRule: используется для определения подмножества сервисов (subsets), позволяет управлять политиками подключения, балансировки и настройки шифрования трафика для этих подмножеств;
- `EnvoyFilter`: ресурс для более тонкой настройки Envoy-прокси, позволяет создавать дополнительные фильтры для уже существующей конфигурации;
- PeerAuthentication: для определения политик аутентификации между сервисами;
- AuthorizationPolicy: определяет политики авторизации на уровне сервисов, позволяет контролировать доступ к ресурсам сервиса на основе различных критериев.

Примечание: примеры конфигураций, описанные в статье, протестированы в кластере Synapse Service Mesh 3.9 (Istio Service Mesh 1.17).

Итак, какие же возможности предлагает Istio для обеспечения безопасного транспорта и контроля над потоками трафика внутри и вне микросервисной архитектуры?

Концепция граничных шлюзов в Istio Service Mesh

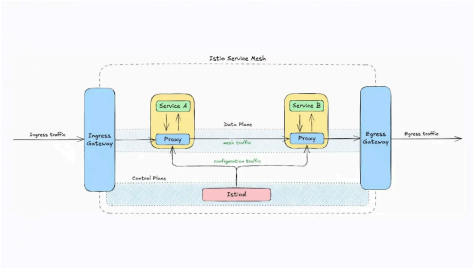
В Service Mesh для улучшения наблюдаемости и безопасности сервисов используется концепция граничных шлюзов. Что это даёт?

- Централизованное управление доступом. Граничные шлюзы служат единственной точкой входа и выхода для всех внешних запросов. Это упрощает внедрение политик безопасности и управление трафиком.
- Наблюдаемость. Через шлюзы собирается информация о входящих и исходящих запросах. Позволяет отслеживать производительность, выявлять узкие места и анализировать поведение системы.
- Безопасность. Шлюзы обеспечивают шифрование трафика, защиту от атак (например, DDoS) и возможность аутентификации пользователей и сервисов.
- Маршрутизация. Можно динамически направлять запросы к нужным сервисам, поддерживая различные стратегии балансировки нагрузки и распределения трафика.

Граничные шлюзы в Istio — это специализированные прокси-серверы, они делятся на два типа:

- Ingress Gateway — для обработки входящего трафика (от клиентов или других систем) в микросервисную сеть.
- Egress Gateway контролирует исходящий трафик — запросы из микросервисной сети наружу (например, внешние API, базы данных или другие сервисы).

Дополним нашу схему представленными компонентами и рассмотрим их назначение и подходы к конфигурированию.



Входящий граничный шлюз Ingress Gateway

`Ingress Gateway` — единая точка входа для внешнего трафика, поступающего в защищённый контур. Разворачивается как отдельный компонент в Kubernetes, работает как обратный прокси-сервер. Все входящие запросы проходят через него, прежде чем попасть к внутренним микросервисам. `Ingress Gateway` отвечает за маршрутизацию, первичную фильтрацию и контроль доступа.

Функциональное назначение:

1. единая точка управления: управление всеми входящими запросами через один узел позволяет централизованно настраивать правила маршрутизации, безопасность, аутентификацию и мониторинг;
2. упрощение администрирования и контроля трафика: вместо взаимодействия с каждым отдельным сервисом все поступающие из внешнего мира запросы обрабатываются на одном узле;
3. безопасность: граничный шлюз выступает в качестве первого рубежа обороны, снижая поверхность атаки и реализуя функции проверки подлинности, шифрования трафика, предотвращения атак и фильтрации нежелательных запросов;
4. оптимизация ресурсов: управление запросами через единый шлюз позволяет оптимизировать распределение нагрузки и лучше контролировать пропускную способность системы.

Создадим конфигурацию сервиса, которая позволит перенаправлять входящии трафик на порт нашего контеинера с приложением, запущенным в поде с sidecar-прокси:

```
---
apiVersion: v1
kind: Service
metadata:
name: curator
spec:
selector:
  app: curator
ports:
  - name: tcp-curator
    protocol: TCP
    port: 8080
    targetPort: 8080
```

Сервис с именем `curator` принимает подключения через порт `8080 (TCP)` и перенаправляет трафик на порт нашего приложения.

Далее опишем конфигурацию сервиса для граничного прокси, который будет принимать все приходящие снаружи запросы:

```
---
kind: Service
apiVersion: v1
metadata:
 name: ingressgateway
 labels:
   istio: ingressgateway
spec:
 ports:
   - name: https
     protocol: TCP
     port: 443
     targetPort: 8443
 selector:
   istio: ingressgateway
```

Все подключения к сервису `ingressgateway` через порт `443 (TCP)` будут перенаправляться на порт `8443 (TCP)`, где Envoy-прокси в поде Ingress-шлюза слушает входящие подключения.

Теперь, когда у нас обеспечена сетевая доступность наших ключевых сервисов, приступим к настройке Istio. Определим точку входа для внешнего трафика с использованием `Gateway`:

```
---
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
 labels:
   app: curator
 name: curator-gw
spec:
 selector:
   istio: ingressgateway
 servers:
   - hosts:
       - curator-da-dp01-db-kintsugi-master.solution.test
     port:
       name: https
       number: 8443
       protocol: HTTPS
     tls:
       caCertificates: /secrets/istio/ingressgateway-ca-certs/ca.crt
       mode: SIMPLE
```

Приведённая конфигурация создаёт Gateway, который принимает входящий HTTPS-трафик на порт 8443 для хоста curator-da-dp01-db-kintsugi-master.solution.test. Трафик обрабатывается с использованием TLS-шифрования. Сертификаты и ключи берутся из секретов, примонтированных в контейнер с Envoy-прокси.

Для описания маршрутизации трафика на уровне сервиса воспользуемся ресурсом VirtualService:

```
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
 name: curator-vs
 labels:
   app: curator
spec:
 exportTo:
   - .
 gateways:
   - curator-gw
 hosts:
   - curator-da-dp01-db-kintsugi-master.solution.test
 http:
   - route:
       - destination:
           host: curator
           port:
             number: 8080
```

Конфигурация VirtualService определяет маршрутизацию HTTP-трафика, поступающего через Gateway для хоста curator-da-dp01-db-kintsugi-master.solution.test. Весь трафик прозрачно направляется на физический сервис, принимающий подключения через порт 8080. Параметр exportTo указывает, что правило применится только для текущего пространства имён. Рекомендуется использовать именно такую конфигурацию, чтобы при отладке не повлиять на работоспособность других проектов, подключённых к контрольной панели Istio.

Напоследок применим конфигурацию, которая создаёт внешний маршрут к сервису curator через единую точку входа:

```
---
kind: Ingress
apiVersion: networking.k8s.io/v1
metadata:
 name: curator
 labels:
   istio: ingressgateway
spec:
 ingressClassName: nginx
 tls:
   - hosts:
       - curator-da-dp01-db-kintsugi-master.solution.test
     secretName: istio-ingressgateway-certs
 rules:
   - host: curator-da-dp01-db-kintsugi-master.solution.test
     http:
       paths:
         - path: /
           pathType: Prefix
           backend:
             service:
               name: ingressgateway
               port:
                 number: 443
```

Трафик, поступающий на хост curator-da-dp01-db-kintsugi-master.solution.test и порт 443, будет перенаправлен на созданный нами сервис ingressgateway, где к нему применятся все политики, описанные в наших ресурсах.

Выполним запрос к ресурсу readiness нашего сервиса и посмотрим лог граничного прокси:

```
date: Fri, 16 May 2025 20:28:10 GMT
server: istio-envoy
content-length: 0
```

Лог `Ingress Gateway`:

```
[2025-05-16T20:28:10.323Z] "GET /readiness HTTP/1.1" 200 - via_upstream - "-" 0 0 23 23 "172.21.9.138" "cur
```

Значение	Интерпретация
`[2025-05-16T20:28:10.323Z]`	Время запроса (UTC)
`"GET /readiness HTTP/1.1"`	Метод, путь запроса и протокол
`200`	HTTP-код ответа (успешно)
`-`	Размер ответа (тело запроса отсутствует)
`via_upstream`	Запрос обработан upstream-сервером
`-`	Upstream service time (не указано)
`"-"`	Referrer (отсутствует)
`0 0`	Размер тела запроса и ответа (тело запроса отсутствует)
`23 23`	23 мс — время между приёмом первого байта запроса и отправкой последнего байта ответа сервером;23 мс — общее время выполнения запроса.
`"172.21.9.138"`	IP-адрес клиента, инициировавшего запрос
`"curl/7.61.1"`	User Agent клиента
`"f15176bd-fa4e-41b9-8a09-142021d9cbaf"`	Request ID — уникальный идентификатор запроса
`"curator-da-dp01-db-kintsugi-master.solution.test"`	Запрашиваемый хост
`"172.21.13.146:8080"`	Адрес конечного сервиса, куда был спроксирован запрос
`outbound|8080||curator.da-dp01-db-kintsugi-master.svc.cluster.local`	Описание направления запроса внутри кластера:`outbound` — исходящий трафик (от шлюза к сервису);`8080` — целевой порт;`curator.da-dp01-db-kintsugi-master.svc.cluster.local` — полное имя сервиса внутри кластера.
`172.21.14.182:55290`	Внутренний интерфейс контейнера, обрабатывающего запрос
`172.21.14.182:8443`	Публичный интерфейс контейнера `Ingress Gateway`, принимающий запрос
`172.21.9.138:10662`	Внешний интерфейс клиента, отправившего запрос через curl
`curator-da-dp01-db-kintsugi-master.solution.test`	Внешний домен приложения, видимый снаружи кластера
`-`	Дополнительные метаданные запроса (данные отсутствуют)
`3363`	Размер лог-записи в байтах

Согласно данным из лога, клиент отправил GET-запрос к ресурсу `readiness` приложения `curator-da-dp01-db-kintsugi-master`. Время выполнения запроса составило 23 мс, операция завершилась успехом.

Теперь все запросы к нашему прикладному сервису фиксируются на граничном шлюзе. Это позволяет нам детально отслеживать и контролировать внешний трафик, поступающий в защищённый контур. Можем быстро выявлять аномалии, анализировать нагрузку и принимать меры для оптимизации безопасности и производительности. Profit!

Граничный исходящий шлюз Egress Gateway

него все запросы, исходящие от внутренних сервисов, направляются во внешние системы или Интернет.

Функциональное назначение:

1. контроль и управление: можно централизованно контролировать то, куда и как отправляются все запросы;
2. повышение безопасности: внешние ресурсы — потенциальный источник угроз, и централизованный контроль исходящего трафика помогает защитить внутреннюю сеть от несанкционированного доступа или утечек данных;
3. оптимизация: управление трафиком через единую точку выхода позволяет эффективнее использовать сетевую инфраструктуру и ресурсы системы.

Перед тем как перейти к разбору проксирования трафика через `Egress Gateway`, коротко расскажу о политиках обработки исходящего трафика. Для управления доступом к внешним сервисам в Istio Service Mesh на глобальном уровне используется ресурс `IstioOperator`. Это один из возможных способов конфигурирования политик безопасности. Он позволяет управлять различными аспектами конфигурирования, включая настройку сетевых политик для исходящего трафика. В частности, параметр `spec.meshConfig.outboundTrafficPolicy.mode` контролирует то, как сервисы в кластере взаимодействуют с внешними сервисами за пределами сервисной сети. Здесь предлагается два режима:

ALLOW_ANY: разрешает исходящие запросы ко всем внешним сервисам без ограничений. Может применяться в архитектуре, где политики Istio используются для внутренних сервисов, но взаимодействие с внешними источниками остаётся полностью открытым.

REGISTRY_ONLY: разрешает исходящие запросы только внутри сервисной сети. Все обращения к внешним ресурсам, не зарегистрированным в Istio, блокируются.

```
---
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: istio-default
...
 spec:
  meshConfig:
   outboundTrafficPolicy:
     mode: REGISTRY_ONLY
...
```

Ставим режим, ограничивающий взаимодействие компонентов, входящих в нашу сервисную сеть, с внешними сервисами, не зарегистрированными в Istio. Так мы обеспечим строгий контроль трафика, исходящего из сервисной сети.

Переходим к реализации сценария взаимодействия компонентов сервисной сети с внешними сервисами, используя `Egress Gateway` в качестве прямого прокси.

Классический пример взаимодействия Platform V Kintsugi с внешними сервисами — это интеграция с экземплярами СУБД снаружи кластера. Данные о наблюдаемых кластерах и объектах мониторинга Kintsugi хранит в репозитории метаданных на основе PostgreSQL. Воспользуемся ресурсом `ServiceEntry` для регистрации внешнего сервиса СУБД в Istio и создадим разрешающее правило для передачи трафика из сервисной сети наружу:

```
---
apiVersion: networking.istio.io/v1beta1
kind: ServiceEntry
metadata:
 name: repository-se
spec:
 addresses:
   - 10.40.20.50
 exportTo:
   - .
 hosts:
   - repository.solution.test
 location: MESH_EXTERNAL
 ports:
   - name: tcp-5433
     number: 5433
     protocol: tcp
 resolution: STATIC
```

Основные параметры конфигурации ресурса `ServiceEntry`:

location: указывает местоположение сервиса:

MESH_INTERNAL: сервис находится внутри сети Istio;

MESH_EXTERNAL: сервис находится снаружи сети Istio;

resolution: параметр определяет способ разрешения адреса внешнего сервиса;

addresses: используется для статического определения конечных точек (адресов) внешних сервисов.

В приведённой конфигурации создаётся запись в реестре Istio для внешнего сервиса PostgreSQL — он зарегистрирован под статически указанным доменным именем `repository.solution.test` (IP: `10.40.20.50`) и принимает подключения через порт `5433 (TCP)`. С этого момента трафик к этому сервису будет отслеживаться и управляться Istio.

Для реализации проксирования трафика через `Egress Gateway` от нашего сервиса к СУБД создадим точку проксирования трафика в граничном прокси:

```
---
kind: Service
apiVersion: v1
metadata:
 name: egress-repository-service
 labels:
   istio: egressgateway
spec:
 ports:
   - name: tcp-passthrough
     protocol: TCP
     port: 5000
     targetPort: 5000
 selector:
   istio: egressgateway
```

В конфигурации определим порт `5000 (TCP)`, через который в `Egress Gateway` будут обрабатываться подключения наших внутренних сервисов к СУБД.

Как и в случае с входящим граничным прокси, с помощью `Gateway` и `VirtualService` добавим политики маршрутизации исходящего трафика:

```
---
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
 name: repository-gw
spec:
 selector:
   istio: egressgateway
 servers:
   - hosts:
       - repository.solution.test
     port:
       name: tcp-passthrough
       number: 5000
       protocol: TCP
```

Приведённая конфигурация создаёт ресурс `Gateway`, который принимает входящий TCP-трафик через порт `5000` для хоста `repository.solution.test` (IP: `10.40.20.50`). Трафик обрабатывается в режиме passthrough: между микросервисом и СУБД настраивается защищённый канал передачи данных со сквозным шифрованием.

```
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
 name: repository-vs
spec:
 exportTo:
   - .
 gateways:
   - repository-gw
```

```
  tcp:
    - match:
        - gateways:
            - mesh
          port: 5433
      route:
        - destination:
            host: egress-repository-service
            port:
              number: 5000
            subset: repository-internal
    - match:
        - gateways:
            - repository-gw
          port: 5000
      route:
        - destination:
            host: repository.solution.test
            port:
              number: 5433
            subset: tcp-passthrough-repository
```

Созданный `VirtualService` управляет маршрутизацией TCP-трафика: все запросы, адресованные хосту СУБД, зарегистрированному в Istio под именем repository.solution.test на порт `5433 (TCP)`, через сервисную сеть перенаправляются на внутренний порт `5000 (TCP)` граничного прокси. После чего трафик, поступающий на порт `5000 (TCP)` граничного прокси, маршрутизируется на адрес хоста `repository.solution.test` (IP: `10.40.20.50`) и порт `5433 (TCP)`. В правилах маршрутизации дифференцируем трафик по порту назначения и ресурсу `Gateway`, где:

- repository-gw: шлюз, созданный на предыдущем шаге и принимающий подключения к Egress Gateway;
- mesh: специальное зарезервированное значение в Istio. Используется для обозначения внутреннего трафика между микросервисами в сервисной сети. Указывает, что соответствующее правило маршрутизации должно применяться к трафику, циркулирующему напрямую между сервисами без проксирования через граничные шлюзы Ingress и Egress.

Для удобства мониторинга и отладки перенаправим исследуемые потоки трафика в выделенные subsets.

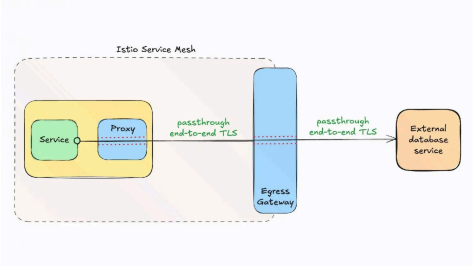
В конце определим ресурсы правила `DestinationRule` для каждого потока трафика:

1. internal: трафик от микросервиса к Egress Gateway (subset repository-internal);
2. external: трафик от Egress Gateway к сервису СУБД (subset tcp-passthrough-repository).

```
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
 name: repository-internal-dr
spec:
 exportTo:
   - .
 host: egress-repository-service
 subsets:
   - name: repository-internal
```

```
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
 name: repository-external-dr
spec:
 exportTo:
   - .
 host: repository.solution.test
 subsets:
   - name: tcp-passthrough-repository
 workloadSelector:
```

Так мы получили схему проксирования трафика через граничный шлюз `Egress Gateway`.



Воспользуемся вспомогательным инструментом и отправим запрос из сервисной сети к внешнему сервису:

```
sh-4.4$ curl -v repository.solution.test:5433
* Rebuilt URL to: repository.solution.test:5433/
*   Trying 10.40.20.50...
* TCP_NODELAY set
* Connected to repository.solution.test (10.40.20.50) port 5433 (#0)
```

```
[2025-05-16T20:35:53.303Z] "- - -" 0 - - - "-" 94 0 7 - "-" "-" "-" "-" "10.40.20.50:5433" outbound|5433|tc
```

Примечание: в логе отсутствуют данные, специфичные для HTTP-трафика.

Значение	Интерпретация
`[2025-05-16T20:35:53.303Z]`	Время запроса (UTC)
`"- - -"`	Метод, путь запроса и протокол отсутствуют
`0`	HTTP-код ответа отсутствует, значение по умолчанию 0
`-`	Размер ответа, тело запроса отсутствует
`-`	Не используется
`-`	Upstream service time не указано
`"-"`	Referrer отсутствует
`94 0`	Размер тела запроса и ответа (тело запроса отсутствует)
`7 -`	7 мс — время между приёмом первого байта запроса и отправкой последнего байта ответа сервером.Данные о продолжительности запроса отсутствуют.
`"-"`	Адрес downstream отсутствует или не определён
`"-"`	User Agent клиента отсутствует
`"-"`	Request ID отсутствует
`"-"`	Запрашиваемый хост отсутствует
`"10.40.20.50:5433"`	Адрес конечного upstream-сервиса, куда был спроксирован запрос
`outbound|5433|tcp-passthrough-repository|repository.solution.test`	Описание направления запроса внутри кластера:`outbound` — исходящий трафик (от шлюза к внешнему сервису);`5433` — целевой порт;`tcp-passthrough-repository` — имя внутреннего маршрута проксируемого трафика;`repository.solution.test` — полное имя внешнего сервиса.
`172.21.28.154:39058`	Внутренний интерфейс контейнера `Egress Gateway`, принимающего запрос
`172.21.28.154:5000`	Публичный интерфейс контейнера `Egress Gateway`, принимающего запрос

	Не используется
–	Дополнительные метаданные запроса (данные отсутствуют)
246	Размер лог-записи в байтах

Согласно данным из лога, изнутри кластера успешно установлено TCP-соединение с сервисом `repository.solution.test`.

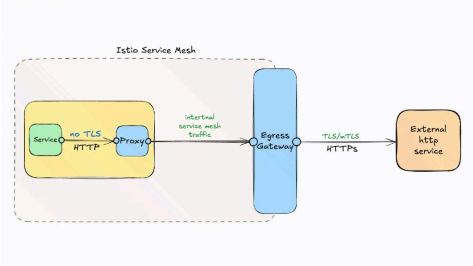
В результате исходящие запросы от нашего приложения ко внешнему сервису СУБД фиксируются на граничном шлюзе `Egress Gateway`. Благодаря чему у нас появляется возможность контролировать сетевое взаимодействие в egress-сегменте нашей сервисной сети.

Разделяем и делегируем

Теперь мы можем контролировать исходящий трафик на уровне `Egress Gateway`. Но что нам это даёт помимо улучшенного мониторинга? И можно ли как-то использовать это при взаимодействии с другими сервисами?

Как и sidecar-прокси, расположенный в поде с контейнером приложения, `Egress Gateway` служит посредником при коммуникации компонентов сервисной сети с внешними сервисами. В предыдущем примере мы рассмотрели взаимодействие нашего компонента с внешним сервисом СУБД, когда клиент использует сквозное защищённое подключение к БД в режиме passthrough и весь трафик прозрачно проксируется через граничный шлюз. Теперь другой пример. Есть задача по доставке логов приложений во внешнее хранилище данных. Для обеспечения защищённого соединения клиент использует сертификат, приватный ключ и цепочку сертификатов доверенных удостоверяющих центров. В сервисной сети с небольшим количеством компонентов задача решается тривиально, но когда они начинают исчисляться десятками или сотнями, конфигурирование и управление сервисами усложняется. Что можно предпринять в таком случае?

Отдадим реализацию безопасного канала связи с внешним сервисом самому граничному шлюзу, так мы освободим прикладные сервисы от инфраструктурных задач.



На схеме представлено взаимодействие прикладного сервиса с внешним ресурсом с использованием защищённого транспорта. Немного позже вернёмся к вопросу шифрования трафика внутри контура сервисной сети. А пока перейдём к настройке граничного шлюза в части его взаимодействия с внешним сервисом Elasticsearch, которое возьмём в качестве примера для хранения логов приложения.

Для простоты положим, что все необходимые артефакты — сертификат клиента, приватный ключ и цепочка сертификатов доверенных удостоверяющих центров — смонтированы в файловую систему `Egress Gateway`, и мы сразу же можем перейти к конфигурированию Istio. В качестве отправной точки на граничном исходящем шлюзе выполним запрос к внешнему сервису. Убедимся в том, что он доступен и у нас есть всё необходимое для реализации защищённого транспорта. В качестве примера используем запрос проверки состояния кластера Elasticsearch:

```
sh-4.4$ curl -X GET --cacert /secrets/istio/egressgateway-ca-certs/ca.crt --http1.1 -I https://elastic.solu
HTTP/1.1 200 OK
Server: nginx/1.27.2
Content-Type: application/json
Content-Length: 390
Connection: keep-alive
X-elastic-product: Elasticsearch
Strict-Transport-Security: max-age=31536000
```

Поскольку граничный шлюз взаимодействует с внешними сервисами напрямую, здесь нам не требуется дополнительного конфигурирования проксирования. Достаточно воспользоваться уже имеющимся набором данных для безопасного подключения к внешнему сервису Elasticsearch.

Проверочный запрос выполнен успешно, можно переходить к настройке проксирования трафика из сервисной сети.

Выполним очередной запрос, но уже из контейнера прикладного сервиса `curator`, и убедимся, что у нас действительно отсутствует сетевая связность с хранилищем логов.

Получаем отказ в установлении соединения от удалённого сервиса. Поэтому переходим к настройке правил обработки трафика в Isito.

Первым делом зарегистрируем внешний сервис в сервисной сети:

```
---
apiVersion: networking.istio.io/v1beta1
kind: ServiceEntry
metadata:
 name: elastic-se
spec:
 exportTo:
   - .
 hosts:
   - elastic.solution.test
 location: MESH_EXTERNAL
 ports:
   - name: http-elastic
     number: 8080
     protocol: HTTP
   - name: tls-elastic
     number: 9200
     protocol: TLS
 resolution: DNS
```

Как ранее упоминалось, эта задача решается с помощью `ServiceEntry`, где мы декларируем наш внешний ресурс с адресом `elastic.solution.test`, принимающим запросы через порт `9200 (HTTPS)`. В этот раз Istio будет самостоятельно разрешать доменное имя узла с помощью DNS и выполнять маршрутизацию трафика на основе правил, указанных нами для этого хоста. Уточню: в нашей архитектуре прикладной сервис не отвечает за обеспечение безопасности, задача шифрования трафика возлагается на инфраструктуру — для этого в сервисной сети мы определим порт `8080 (HTTP)` для обработки таких запросов.

Создадим сервис для приёма трафика граничным шлюзом от узлов сервисной сети:

```
---
kind: Service
apiVersion: v1
metadata:
 name: egress-elastic-service
 labels:
   istio: egressgateway
spec:
 ports:
   - name: tcp-elasticsearch
     protocol: TCP
     port: 4000
     targetPort: 4000
 selector:
  istio: egressgateway
```

Теперь исходящий граничный шлюз готов принимать от нас подключения к порту `4000` через выделенный для нашей задачи сервис `egress-elastic-service`.

Когда определены все базовые сущности для транспорта трафика, пришло время описать правила маршрутизации запросов, адресованных нашему внешнему сервису. Для этого определим в ресурсе `Gateway` адрес и порт внешнего сервиса и с помощью `VirtualService` свяжем его с правилом маршрутизации трафика:

```
---
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
 name: elastic-gw
spec:
 selector:
   istio: egressgateway
```

```
      port:
        name: http-elasticsearch
        number: 4000
        protocol: HTTP
```

Конфигурация создаёт шлюз с именем `elastic-gw`, его задача — перенаправлять HTTP-трафик, адресованный хосту `elastic.solution.test` на граничный шлюз `Egress Gateway`, который принимает подключения через порт `4000`.

```
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
 name: elastic-vs
spec:
 exportTo:
   - .
 gateways:
   - elastic-gw
   - mesh
 hosts:
   - elastic.solution.test
 http:
   - match:
       - gateways:
           - mesh
         port: 8080
     route:
       - destination:
           host: egress-elastic-service
           port:
             number: 4000
           subset: elastic-internal
   - match:
       - gateways:
           - elastic-gw
         port: 4000
     route:
       - destination:
           host: elastic.solution.test
           port:
             number: 9200
           subset: tls-origination-elastic
```

В конфигурации `VirtualService` определим два правила маршрутизации:

1. Запросы, приходящие на внутренний шлюз mesh и порт 8080 перенаправляются на порт 4000 сервиса egress-elastic-service. Для полноты контроля определим для этого потока трафика subset elastic-internal.
2. Запросы, поступающие на шлюз elastic-gw и порт 4000 перенаправляются на хост elastic.solution.test и порт 9200. Трафик, соответствующий заданному набору критериев, выделим в subset tls-origination-elastic.

Таким образом, запросы от внутренних сервисов к хосту с именем elastic.solution.test через порт 8080 (HTTP) маршрутизируются на шлюз Egress Gateway и далее перенаправляются на целевой хост и порт внешнего сервиса.

Всё, что нам осталось — определить политики обработки для двух потоков трафика:

1. internal: трафик к Egress Gateway от внутреннего сервиса (subset elastic-internal);
2. external: трафик от Egress Gateway к хранилищу данных Elasticsearch (subset tls-origination-elastic).

Воспользуемся ресурсом `DestinationRule` и опишем политику для каждого из потоков:

```
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
 name: egress-elastic-internal-dr
spec:
```

```
  subsets:
    - name: elastic-internal
  workloadSelector:
    matchLabels:
      app: curator
```

Конфигурация описывает политику обработки трафика, передаваемого от внутреннего сервиса `curator` к исходящему граничному шлюзу. Здесь создаётся subset `elastic-internal`, в котором трафик передаётся прозрачно без применения каких-либо дополнительных политик обработки. Применяем правило для ресурсов с меткой прикладного сервиса `app: curator`.

На финальном шаге конфигурирования определим политику для потока трафика от граничного шлюза к внешнему сервису:

```
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
 name: egress-elastic-external-dr
spec:
 exportTo:
   - .
 host: elastic.solution.test
 subsets:
   - name: tls-origination-elastic
     trafficPolicy:
       loadBalancer:
         simple: ROUND_ROBIN
       portLevelSettings:
         - port:
             number: 9200
           tls:
             caCertificates: /secrets/istio/egressgateway-ca-certs/ca.crt
             clientCertificate: /secrets/istio/egressgateway-certs/tls.crt
             mode: SIMPLE
             privateKey: /secrets/istio/egressgateway-certs/tls.key
             sni: elastic.solution.test
 workloadSelector:
   matchLabels:
     istio: egressgateway
```

Политика определяет subset `tls-origination-elastic` и позволяет установить защищённое подключения в режиме `SIMPLE` с проверкой подлинности сертификата сервера. Так, все запросы, адресованные хосту `elastic.solution.test` на порт `9200`, будут шифроваться на граничном шлюзе и отправляться внешнему сервису.

Теперь, когда конфигурация готова, проверим тестовым запросом её работу. Отправим запрос из сервисной сети к внешнему хранилищу Elasticsearch:

```
sh-4.4$ curl -X GET -u $USER:$PASSWORD --http1.1 -I http://elastic.solution.test:8080/_cluster/health
HTTP/1.1 200 OK
server: envoy
date: Fri, 16 May 2025 20:42:47 GMT
content-type: application/json
content-length: 390
x-elastic-product: Elasticsearch
strict-transport-security: max-age=31536000
x-envoy-upstream-service-time: 7
```

Лог `Egress Gateway`:

```
[2025-05-16T20:42:47.162Z] "GET /_cluster/health HTTP/1.1" 200 - via_upstream - "-" 0 390 63 62 "172.21.13.
```

Запрос выполнен успешно, это значит, что граничный шлюз обеспечивает прозрачное для прикладного сервиса защищённое взаимодействие с внешним ресурсом.

Аутентификация

Istio обеспечивает проверку подлинности клиентов, отправляющих запросы к сервисам внутри сервисной сети. Поддерживает два основных механизма аутентификации:

- сетевая аутентификация (mTLS-аутентификация);
- JWT-аутентификация.

Рассмотрим каждый из них подробно.

Шифрование данных при межсервисном взаимодействии и сетевая аутентификация

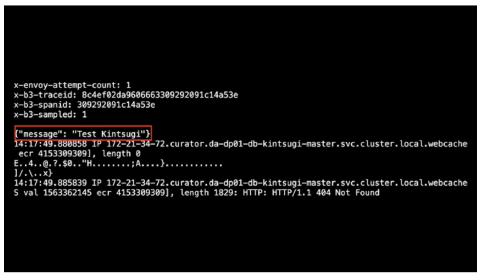
Mutual TLS (mTLS) — расширение стандартного протокола шифрования TLS, где не только сервер аутентифицирует клиента, но и клиент проверяет подлинность сервера. Это повышает безопасность взаимодействия между сервисами.

Рассмотрим работу механизма в действии. Но сперва проверим, в каком виде данные передаются внутри сервисной сети без включённой политики сетевой аутентификации. Отправим тестовый запрос из сервисной сети к нашему прикладному сервису `curator` и выполним дамп сетевого трафика на внешнем интерфейсе пода:

```
sh-4.4$ curl -i -X POST curator:8080 -d '{"message": "Test Kintsugi"}'
HTTP/1.1 200 OK
server: envoy
content-length: 22
content-type: application/json
```

С помощью curl отправим «тайное» послание сервису изнутри сети Istio и параллельно посмотрим, что об этом думает `tcpdump`:

```
sh-4.4$ tcpdump -A -i eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```



Как мы видим, трафик передаётся в незашифрованном виде, поэтому перейдём к настройке шифрования и взаимной аутентификации в нашей сервисной сети. Обратимся к ресурсу `PeerAuthentication`, который управляет настройками аутентификации на уровне сервиса. Он определяет требования к аутентификации между клиентом и сервером в рамках одного пространства имён или всего кластера. Рассмотрим базовый вариант его конфигурации:

```
---
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
 name: default
spec:
 mtls:
   mode: STRICT
```

`spec.mtls.mode` — ключевой параметр ресурса, который определяет требования к режиму работы mTLS для входящих запросов и может принимать следующие значения:

- UNSET: наследуется режим от родительской политики, например, глобальной настройки сервисной сети;
- DISABLE: mTLS отключен, трафик передаётся в открытом виде (plain-text);
- PERMISSIVE: разрешает использовать как plain-text, так и mTLS-соединение;
- STRICT: допускает использование только mTLS-соединения.

В нашем примере будем строго требовать возможность использовать режим mTLS для всех входящих запросов. Это правило применяется ко входящим соединениям и гарантирует, что любой клиент, обращающийся к сервису, пройдёт через взаимную

```
sh-4.4$ curl -i -X POST curator:8080 -d '{"message": "Test Kintsugi"}'

HTTP/1.1 503 Service Unavailable
content-length: 95
content-type: text/plain
server: envoy
upstream connect error or disconnect/reset before headers. reset reason: connection termination
```

В ответе на запрос получаем ошибку с HTTP-кодом 503 Service Unavailable, свидетельствующую о недоступности сервиса. Почему так произошло? Всё дело в том, что теперь в сервисной сети действует правило, требующее от сервиса-клиента включённого режима mTLS при установлении соединения. Устраним данный недочёт. В этом нам поможет ресурс DestinationRule, определяющий правила исходящего соединения:

```
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
 name: enable-mtls-dr
spec:
 exportTo:
   - .
 host: '*.svc.cluster.local'
 trafficPolicy:
   tls:
     mode: MUTUAL_TLS
```

`DestinationRule` предоставляет широкий спектр возможностей для управления исходящим трафиком. Остановимся на наиболее важных для нас параметрах: `spec.host` — адрес хоста, к которому будут применены политики, описанные в ресурсе, и `spec.trafficPolicy.tls.mode` — режим TLS соединения. Istio предполагает использование следующих режимов:

- DISABLE: отключение TLS, все подключения между сервисами устанавливаются в незащищённом режиме, нет шифрования (данные передаются в открытом виде);
- SIMPLE: используется, когда нужна защита передаваемых данных, но проверка подлинности клиента не требуется;
- MUTUAL: режим активирует взаимную аутентификацию, проверяется подлинность обоих респондентов — это наиболее безопасный вариант установления соединения;
- ISTIO_MUTUAL: функционально соответствует режиму MUTUAL, при его использовании не требуется указывать данные сертификатов и ключа клиента — всю работу по генерации и управлению сертификатами берёт на себя Istio.

В нашем примере мы создали ресурс `DestinationRule`, в котором политики применяются ко всем сервисам в домене `*.svc.cluster.local` и устанавливают режим работы `ISTIO_MUTUAL`.

Если в глобальной конфигурации сервисной сети установлен флаг `enableAutoMtls: true`, то явное указание параметров `trafficPolicy.tls` в ресурсе `DestinationRule` становится необязательным: Istio автоматически применяет mTLS для взаимодействия между сервисами. Однако определение `trafficPolicy.tls` в `DestinationRule` остаётся важным инструментом для полного контроля над шифрованием трафика и политиками безопасности, независимо от глобальных настроек сервисной сети.

После конфигурирования повторим запрос к испытуемому сервису и выполним дамп сетевого трафика:

```
sh-4.4$ tcpdump -A -i eth0
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes


sh-4.4$ curl -i -X POST curator:8080 -d '{"message": "Test Kintsugi"}'
HTTP/1.1 200 OK
server: envoy
content-length: 22
content-type: application/json
```

Запрос выполнен успешно с ожидаемым результатом. Взглянем на дамп трафика:

На скриншоте видим запрос от сервиса-клиента к прикладному сервису `curator` и передаваемые в канале данные в зашифрованном виде. Всё работает!

Ресурсы `PeerAuthentication` и `DestinationRule` позволяют гибко конфигурировать политики аутентификации в среде Istio. Однако важно учитывать, что неправильная настройка политик может приводить к рассогласованию настроек транспорта и, как следствие, прерыванию связи между сервисами и дестабилизации работы вашего приложения.

Мы рассмотрели сценарий межсервисной аутентификации внутри сервисной сети. Но как обеспечить безопасность приложения в случае, если сервис или пользователь находится за её пределами, и механизму Istio уже неподвластен полный контроль и управление объектами аутентификации? Здесь нам на помощь приходят специальные возможности для идентификации и проверки подлинности клиента.

Аутентификация запросов с использованием jwt

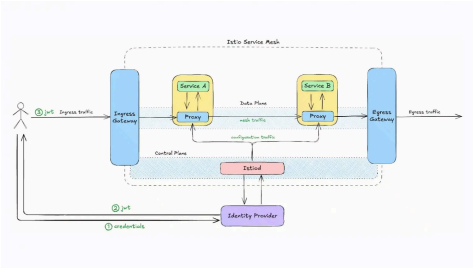
Аутентификация JSON Web Token (JWT) — это один из популярных методов, который широко применяется в современных веб-приложениях и API. Основные компоненты JWT-аутентификации:

1. Issuer: служба, выпускающая токены;
2. JWKS (JSON Web Key Set): набор ключей, используемых для подписи и проверки подписей токена;
3. Token: сам токен, который содержит информацию о пользователе и данные, используемые для проверки подлинности.

Процесс аутентификации состоит из следующих этапов:

1. Выпуск токена: пользователь предоставляет свои учётные данные (имя пользователя, пароль) серверу аутентификации. Если проверка прошла успешно, то сервер возвращает JWT-токен.
2. Передача токена: токен отправляется вместе с каждым HTTP-запросом в заголовке Authorization с префиксом Bearer. Например: curl -H “Authorization: Bearer <token>” https://kintsugi.solution.test.
3. Проверка токена: когда запрос поступает на ресурс, защищённый Istio, происходит проверка токена. Она включает в себя следующие шаги:
   1) проверка подписи токена с использованием JWKS;
   2) проверка срока действия токена;
   3) проверка соответствия требуемым данным: идентификатор пользователя, роли и другие атрибуты.

Перед тем как перейти к настраиванию инфраструктуры дополним нашу архитектурную схему компонентом IdP (Indentity Provider). Это сервис управления идентификацией пользователя, используемый для аутентификации, авторизации пользователей в прикладном сервисе. Также он отвечает за выпуск JWT-токенов.



Для контроля изменений поведения в процессе дальнейшего конфигурирования выполним исходный запрос к сервису `curator` снаружи сервисной сети с использованием ранее настроенного ресурса `Ingress`:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt https://curator-da-dp01-db-kintsugi-master.solut
```

В нашем запросе к API-ресурсу `readiness` компонента `curator` используем метод GET. Работаем по протоколу `HTTP 1.1`, обрабатываем все редиректы и обеспечиваем клиента сертификатами доверенных удостоверяющих центров, необходимых для проверки сертификата сервера при установлении защищённого соединения. В результате получаем успешный ответ с кодом `200 OK`, что сигнализирует нам о беспрепятственном доступе к сервису извне:

```
HTTP/1.1 200 OK
server: istio-envoy
content-length: 0
```

Для настройки политик аутентификации в Isito воспользуемся ресурсом `RequestAuthentication`, где определим правила аутентификации для входящего трафика, адресованного прикладному сервису. Это позволит нам проконтролировать то, какие запросы будут разрешены, а какие — отклонены на основе предъявляемого JWT-токена:

```
kind: RequestAuthentication
metadata:
 name: kintsugi
spec:
 selector:
   matchLabels:
     app: curator
 jwtRules:
   - forwardOriginalToken: true
     issuer: https://idp.solution.test/auth/realms/da-dp01-db-kintsugi-master
     jwksUri: https://idp.solution.test/auth/realms/da-dp01-db-kintsugi-master/protocol/openid-connect/cert
```

Ключевые параметры ресурса:

- spec.jwtRules.issuer: идентификатор службы, выпустившей токен;
- spec.jwtRules.jwksUri: ресурс для получения публичных ключей для проверки подписи JWT-токенов;
- spec.selector.matchLabels: метка, определяющая, к каким объектам будет применена политика аутентификации;
- spec.jwtRules.forwardOriginalToken: определяет, будет ли перенаправлен токен в запросе приложению.

Ресурс устанавливает правило аутентификации входящего трафика для сервиса с меткой `app: curator`. Когда запрос поступает в сервисную сеть, Istio проверяет наличие валидного JWT-токена, подписанного издателем `issuer` и проверенного через указанный JWKS URI.

Проверим, как отработает запрос, содержащий невалидный токен в заголовке `Authorization`:

```
sh-4.4$ INVALID_TOKEN="This is invalid token"
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt -H "Authorization: Bearer $INVALID_TOKEN" https:
HTTP/1.1 401 Unauthorized
www-authenticate: Bearer realm="https://curator-da-dp01-db-kintsugi-master.solution.test/readiness", error=
content-length: 79
content-type: text/plain
date: Fri, 16 May 2025 21:04:58 GMT
server: istio-envoy
```

Получаем ответ с HTTP-кодом `401 Unauthorized`, и лог `Ingress Gateway` подтверждает это:

```
[2025-05-16T21:04:58.978Z] "GET /readiness HTTP/1.1" 401 - jwt_authn_access_denied{Jwt_is_not_in_the_form_o
```

Повторим наш запрос. На этот раз передадим токен в корректном формате, но он будет выпущен сторонней службой, недоверенной в нашем домене Istio. Пример распарсенного токена, который будем использовать в запросе:

```
{
  "alg": "RS256",
  "typ": "JWT",
}
{
  "exp": 1747419301,
  "iat": 1747419001,
  "jti": "f3fe8ea9-075b-4cb5-b729-20842373fd60",
  "iss": "https://invalid.solution.test/auth/realms/da-dp01-db-kintsugi-master",
  "aud": "kintsugi",
  "sub": "d9ea00d0-0803-42b3-a1ed-fb277159308b",
  "name": "user",
  "preferred_username": "kintsugi"
}
```

Здесь обратим внимание, что значение ключа `iss` указывает на ложный провайдер аутентификации и отличается от указанного в политике аутентификации `RequestAuthentication` в качестве значения ключа `issuer`.

Запрос:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt -H "Authorization: Bearer $INVALID_ISSUER_TOKEN"
HTTP/1.1 401 Unauthorized
www-authenticate: Bearer realm="https://curator-da-dp01-db-kintsugi-master.solution.test/readiness", error=
```

```
server: istio-envoy
```

В логе `Ingress Gateway` видим специфичную ошибку, сигнализирующую о том, что указанный в токене `issuer` не сконфигурирован в нашей политике и, как следствие, не является доверенным. В результате чего получаем отказ в аутентификации запроса:

```
[2025-05-16T21:10:48.189Z] "GET /readiness HTTP/1.1" 401 - jwt_authn_access_denied{Jwt_issuer_is_not_config
```

Наконец, выполним проверку аутентификации с корректным токеном, выпущенным доверенным провайдером идентификации, и убедимся, что аутентификация пройдёт успешно. Пример распарсенного токена, который будем использовать в запросе:

```
{
  "alg": "RS256",
  "typ": "JWT",
}
{
  "exp": 1747419665,
  "iat": 1747419365,
  "jti": "94360a03-96cb-4ed6-a483-dee8defeb89e",
  "iss": "https://idp.solution.test/auth/realms/da-dp01-db-kintsugi-master",
  "aud": "kintsugi",
  "sub": "daf933df-11b5-40b9-9d23-8ee034595123",
  "name": "user",
  "preferred_username": "kintsugi"
}
```

Приведённый токен действителен и соответствует требованиям правил `jwtRules`, определённым в политике аутентификации `RequestAuthentication`. Поэтому перейдём к запросу:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt -H "Authorization: Bearer $VALID_TOKEN" https://
HTTP/1.1 200 OK
date: Fri, 16 May 2025 21:16:28 GMT
server: istio-envoy
content-length: 0
```

Запрос успешно прошёл проверку токена, по результатам чего мы получили ответ от нашего прикладного сервиса, об этом свидетельствует лог `Ingress Gateway`:

```
[2025-05-16T21:16:21.703Z] "GET /readiness HTTP/1.1" 200 - via_upstream - "-" 0 0 37 25 "172.21.9.138" "cur
```

Итак, мы увидели, как работает механизм аутентификации в Istio. Чуть позже посмотрим, что можно было бы здесь улучшить. А пока вернёмся к вопросу взаимной сетевой аутентификации с использованием протокола mTLS и немного подзакрутим гайки в части взаимодействия внешнего пользователя или сервиса с нашим приложением.

Ранее с помощью Gateway мы настроили взаимодействие с прокси с использованием режима установления защищённого соединения `mode: SIMPLE`, когда клиент проверяет подлинность сервера, в роли которого выступает наш прикладной сервис `curator`. Почему бы дополнительно не убедиться в подлинности клиента, будь то пользователь или внешний клиентский сервис?

Доработаем конфигурацию TLS-ресурса Gateway для включения режима взаимной аутентификации. Установим значение параметра `spec.servers.tls.mode` в режим `MUTUAL`:

```
---
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
 labels:
   app: curator
 name: curator-gw
spec:
 selector:
   istio: ingressgateway
 servers:
```

```
      name: https
      number: 8443
      protocol: HTTPS
    tls:
      caCertificates: /secrets/istio/ingressgateway-ca-certs/ca.crt
      mode: MUTUAL
      privateKey: /secrets/istio/ingressgateway-certs/tls.key
      serverCertificate: /secrets/istio/ingressgateway-certs/tls.crt
```

Проверим, как изменилось поведение при установлении защищённого соединения с использованием взаимной аутентификации mTLS:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt -H "Authorization: Bearer $VALID_TOKEN" https://
curl: (56) OpenSSL SSL_read: error:1409445C:SSL routines:ssl3_read_bytes:tlsv13 alert certificate required,
```

При попытке выполнить запрос получаем ошибку. Нам указывают на отсутствие в запросе информации о клиентском сертификате, по этой причине получаем отказ.

Исправим ситуацию и повторим запрос уже с предъявлением выпущенного сертификата и ключа клиента:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt --cert kintsugi.crt --key kintsugi.key -H "Autho
HTTP/1.1 200 OK
server: istio-envoy
content-length: 0
```

Запрос завершился успешно, по результатам взаимной аутентификации всё работает. Мы получили полноценное работающее решение с использованием взаимной аутентификации mTLS клиента и сервера.

Цель достигнута. Кажется, что процесс аутентификации доведён до совершенства. На этом можно было бы остановиться, но...

Смоделируем ситуацию, когда нелегитимный клиент пытается обратиться к нашему сервису, и у него есть действующий сертификат пользователя. Гипотетически такая ситуация может возникнуть, например, при блокировке пользователя и несвоевременном отзыве его персонального сертификата.

Предварительно сгенерировав сертификат и ключ, выполним тестовый запрос к сервису:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt --cert illegitimate.crt --key illegitimate.key -
HTTP/1.1 200 OK
server: istio-envoy
content-length: 0
```

Как мы видим, запрос завершился успехом, потому что в данном случае по-другому не могло и быть. Самое время задуматься о механизме фильтрации подключений по особому признаку. Например, по атрибутам сертификата клиента. Ранее мы упоминали о ресурсе `EnvoyFilter`, его основное предназначение в Istio — тонкое конфигурирование Envoy-прокси. Как раз-таки здесь он нам и поможет. Сконфигурируем фильтр так, чтобы при обработке запроса производился анализ заголовка, содержащего сертификат клиента, и обработка выполнялась только для определённого значения атрибута Common Name (CN):

```
---
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: ingressgateway-cert-filter
  labels:
    app.kubernetes.io/managed-by: Helm
spec:
  configPatches:
    - applyTo: HTTP_FILTER
      match:
        context: GATEWAY
        listener:
          filterChain:
            filter:
              name: envoy.filters.network.http_connection_manager
          portNumber: 8443
```

```
          name: envoy.filters.http.rbac
          typed_config:
            '@type': type.googleapis.com/envoy.extensions.filters.http.rbac.v3.RBAC
            rules:
              action: ALLOW
              policies:
                auth-by-cert-policy:
                  permissions:
                    - any: true
                  principals:
                    - or_ids:
                        ids:
                          - header:
                              name: x-forwarded-client-cert
                              safe_regex_match:
                                google_re2:
                                  max_program_size: 10000
                                regex: .*Subject="CN=kintsugi".*
  workloadSelector:
    labels:
      istio: ingressgateway
```

Определяем патч, который будет применён к конфигурации Envoy. Патч применяется в контексте `Gateway` для порта 8443 и выполняет разрешающее действие по результатам проверки содержимого заголовка `x-forwarded-client-cert`, в котором передаются данные клиентского сертификата. Фильтр применяется к компоненту `ingressgateway`. Пример фильтра работает по принципу белого списка и допускает обращение к ресурсам прикладного сервиса клиента с сертификатом, содержащим атрибут CN со строго определённым значением: `CN=kintsugi`

Проверим исправность работы нашего механизма фильтрации запросов. Для этого выполним запрос с клиентским сертификатом, где `CN=illegitimate`:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt --cert illegitimate.crt --key illegitimate.key -
HTTP/1.1 403 Forbidden
content-length: 19
content-type: text/plain
server: istio-envoy
```

Повторим запрос для клиента с сертификатом, где `CN=kintsugi`:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt --cert kintsugi.crt --key kintsugi.key -H "Autho
HTTP/1.1 200 OK
server: istio-envoy
content-length: 0
```

Приведённые примеры демонстрируют простейший кейс использования механизма фильтрации запросов. Применяя всю мощь регулярных выражений, Istio позволяет сформировать гибкую политику обработки трафика, которую можно использовать в своём проекте.

Итак, мы рассмотрели возможные сценарии использования механизма аутентификации в Istio. Разобрали примеры конфигурации и, наконец, подошли к не менее важному механизму обеспечения безопасности в нашем защищённом контуре — авторизации.

Авторизация

Авторизация — механизм определения прав доступа для уже аутентифицированного пользователя или сервиса. Базовое управление политиками авторизации в Istio выполняется с использованием ресурса `AuthorizationPolicy`.

Рассмотрим процесс авторизации в действии. Вернёмся к примеру запроса с использованием JWT-токена, с ним мы тестировали механизм аутентификации:

```
sh-4.4$ INVALID_TOKEN="This is invalid token"
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt -H "Authorization: Bearer $INVALID_TOKEN" https:
HTTP/1.1 401 Unauthorized
```

1. Заблаговременно обогатим наш запрос данными о клиентском сертификате и ключе, чтобы избежать проблемы с проверкой подлинности клиента.
2. В исследовательских целях исключим из запроса заголовок авторизации: Authorization: Bearer $INVALID_TOKEN

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt --cert kintsugi.crt --key kintsugi.key https://c
HTTP/1.1 200 OK
server: istio-envoy
content-length: 0
```

В результате получаем ответ с HTTP-кодом `200 OK`, что означает успешно выполненный запрос в обход аутентификации клиента. Это явно не входило в наши планы. Теперь наша задача — обеспечить контроль подобных запросов. И в качестве первого рубежа обороны нам надо реализовать проверку наличия токена в запросе. Для этого воспользуемся возможностью политики авторизации и реализуем правило проверки нашего запроса:

```
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: authz
spec:
 action: ALLOW
 rules:
   - from:
       - source:
           requestPrincipals:
             - '*'
 selector:
   matchLabels:
     app: curator
```

Ключевые параметры конфигурации `AuthorizationPolicy`:

spec.action: действие, которое должно применяться в случае выполнения правил, описанных в политике:
ALLOW: разрешить доступ;
DENY: отказать в доступе.

spec.rules: набор правил, определяющих условия, при которых действие выполняется; Каждый элемент массива rules представляет собой отдельное правило.

spec.rules.from.source.requestPrincipals: перечень источников запросов. Это может быть список пользователей или групп.

spec.selector: селектор для выбора сервисов, к которым будет применяться политика.

Конфигурация ресурса создаёт политику авторизации с именем `authz`, которая будет применена для прикладного сервиса `curator`. Политика разрешает запросы, в которых предоставляется информация о токене, и не накладывает ограничений на идентификатор источника запроса.

Применив политику, повторим запрос к нашему сервису без передачи заголовка авторизации с токеном:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt --cert kintsugi.crt --key kintsugi.key https://c
HTTP/1.1 403 Forbidden
content-length: 19
content-type: text/plain
server: istio-envoy
```

Получаем ответ с HTTP-кодом `403 Forbidden` — в нашем случае ожидаемый результат. Дополнительно проверим нашу политику на предмет корректной работы и отправим запрос с валидным токеном, выпущенным доверенным провайдером идентификации:

```
curl --http1.1 -I -X GET -L --cacert ca_bundle.crt --cert kintsugi.crt --key kintsugi.key -H "Authorization
HTTP/1.1 200 OK
server: istio-envoy
content-length: 0
```

Таким образом, мы проверили правильность работы сконфигурированной политики и реализовали проверку обязательного наличия JWT-токена в запросе.

Перейдем к следующему кейсу и переработаем созданное правило. Сделаем его более строгим. На примере запроса к сервису Kintsugi реализуем политику авторизации. Её задача будет заключаться в анализе содержимого объекта `roles` токена, в котором определены роли, назначенные клиенту.

Для демонстрации работы правила допустим, что у нас есть роль `kintsugi-readiness-role`, позволяющая обращаться к ресурсу `readiness` приложения, и определять его готовность к обработке запросов.

```
{
  "alg": "RS256",
  "typ": "JWT",
}
{
  "exp": 1747420820,
  "iat": 1747420520,
  "jti": "89858c3b-f955-4b9a-8430-e76444e68d71",
  "iss": "https://idp.solution.test/auth/realms/da-dp01-db-kintsugi-master",
  "aud": "kintsugi",
  "sub": "daf933df-11b5-40b9-9d23-8ee034595123",
  "name": "user",
  "realm_access": {
    "roles": [
      "readiness-role"
    ]
  },
  "preferred_username": "kintsugi"
}
```

С помощью политики авторизации определим политику доступа к заданному API-ресурсу приложения и посмотрим, как это работает:

```
---
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: authz
spec:
 action: ALLOW
 rules:
   - to:
       - operation:
           methods:
             - GET
           paths:
             - /readiness
     from:
       - source:
           requestPrincipals:
             - '*'
     when:
       - key: 'request.auth.claims[realm_access][roles]'
         values:
           - kintsugi-readiness-role
 selector:
   matchLabels:
     app: curator
```

Политика авторизации применяет разрешающее правило для выполнения API-запроса к ресурсу `readiness` компонента `curator` при наличии у объекта аутентификации необходимой роли `kintsugi-readiness-role`. Вместе с этим установленное разрешающее правило ставит неявный запрет на обращение к ресурсам API, не определённым в политике.

Выполним запрос к ресурсу `liveness`:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt --cert kintsugi.crt --key kintsugi.key -H "Autho
HTTP/1.1 403 Forbidden
```

Ресурс `liveness` не определён в политике, поэтому мы получили отказ в доступе.

Убедимся в том, что для явно объявленного правила для ресурса `readiness` запрос выполняется корректно:

```
sh-4.4$ curl --http1.1 -I -X GET -L --cacert ca_bundle.crt --cert kintsugi.crt --key kintsugi.key -H "Autho
HTTP/1.1 200 OK
server: istio-envoy
content-length: 0
```

Результаты выполнения запросов ожидаемы — политика работает корректно. Мы добавили возможность контроля доступа, но теперь уже на уровне ролевой модели приложения. Этот подход позволяет определять политики доступа пользователей к ресурсам приложения, исходя из содержимого JWT-токена, предлагая широкие возможности настраивания правил авторизации в среде Istio.

Финально взглянем, как работает ограничение используемого метода при выполнении запроса. Переквалифицируем наш запрос в `POST`-запрос:

```
sh-4.4$ curl --http1.1 -I -X POST -L --cacert ca_bundle.crt --cert kintsugi.crt --key kintsugi.key -H "Auth
HTTP/1.1 403 Forbidden
content-length: 19
content-type: text/plain
server: istio-envoy
x-envoy-upstream-service-time: 4
```

Well done!

Политики авторизации Istio позволяют ограничить доступ к ресурсам приложения на основе анализа пользовательских атрибутов и проверке соответствующих правил, заданных в ресурсе `AuthorizationPolicy`. Авторизация и аутентификация тесно взаимосвязаны и должны настраиваться комплексно. В примерах выше мы рассмотрели авторизацию на основе анализа содержимого JWT-токенов, но это лишь малая часть того, что умеет Istio.

Вместо заключения

Очень надеюсь, что приведённые мной изыскания окажутся вам полезными и пригодятся в проектировании и реализации микросервисных решений.

На основе примеров в статье я подготовил демо-проект, который можно использовать для тестирования кейсов и дальнейшего развития идеи построения защищённого микросервисного приложения: https://gitverse.ru/spbvalentine/istio-demo.

Другие новости

Все новости

Все новости

Kintsugi

Графическая консоль управления реляционными СУБД

Решение для надежной оркестрации микросервисов в контейнерной среде

PLATFORM V SBER

platformv@sbertech.ru

+7 495 547 99 80

Стать партнером:
partners.platformv@sbertech.ru

Подпишитесь на рассылку Platform V

Email

Я даю АО «СберТех» согласие на получение предложений, информационных и рекламных материалов, а также обработку моих персональных данных (адреса электронной почты) в целях осуществления информационных и маркетинговых коммуникаций

Enterprise-система – корпоративная, бизнес-система Open source — ПО с открытым исходным кодом. High load — высокие нагрузки, высоконагруженные приложения. CTO — технический директор. CISO — директор по информационной безопасности. Low-code — разработка ПО без создания большого количества кода.