Ausgabe 04|2023
Deutschland € 15,90 Österreich € 16,90 Schweiz sfr 24,20
4 194164 115901 04

www.ITSpektrum.de

IT Spektrum

vormals OBJEKTspektrum

Digitaler Wandel & Software-Architektur für Profis

Navigieren im Brown Field:

Wie sich das Miteinander von alter und neuer IT effektiv gestalten lässt

Zero Trust:
Vertraue niemanden und verifiziere jeden

Netzwerk-Security
Kubernetes – aber sicher!

Architektur-Porträt
Der Instant-Messenger Threema

Navigieren im Brown Field: Wie sich das Miteinander von alter und neuer IT effektiv gestalten lässt

G 6540F

Netzwerk-Security mit Istio

Kubernetes – aber sicher!

In Zeiten von Cloud-Computing ist das Thema Netzwerksicherheit allgegenwärtig. Klassische Ansätze wie host-spezifische Traffic-Kontrolle lassen sich aber nicht immer leicht umsetzen. Wir demonstrieren, wie dieses Problem mit dem Istio Service Mesh gelöst werden kann und welche Herausforderungen es dabei gibt.

Microservice-Architekturen ermöglichen es, komplexe Anwendungen in kleinere, besser beherrschbare Teile zu zerlegen (**siehe Abbildung 1**). Eine gute Möglichkeit, solche Systeme zu deployen, besteht darin, die einzelnen Microservices mit allen ihren Abhängigkeiten in Container zu verpacken und diese Container miteinander zu verknüpfen. So etwas geschieht in einem Container-Orchestrierungssystem wie Kubernetes, in dem die gesamte Konfiguration des Systems in wenigen Config-Dateien festgelegt wird (nach dem Ansatz von Infrastructure-as-Code).

Kubernetes ist eine Open Source basierte Lösung, mit der Container auf Clustern orchestriert werden können. Dabei ist es möglich, diese Cluster selbst zu hosten. Aber gerade für kleinere Unternehmen oder Projekte ist es oft praktisch, betriebsfertige Cluster-Systeme als Software-as-a-Service von einem der vielen Anbieter zu mieten. Die verschiedenen Angebote sind weitgehend kompatibel, was die Gefahr eines Vendor-Locks minimiert.

Wer das Ziel hat, eine Applikation in einem Kubernetes-Cluster in der Cloud zu betreiben, muss sich früher oder später die Frage stellen, wie er oder sie in diesem Fall mit Sicherheitsrisiken umgehen möchte. Am besten nimmt man das Thema Sicherheit frühzeitig unter die Lupe, um es bei der Planung der Anwendung zu berücksichtigen und Risiken gegebenenfalls zu minimieren. Wir machten dabei in unseren Projekten die Erfahrung, dass insbesondere die Sicherung der Kommunikation der Cluster-Anwendung mit der Außenwelt herausfordernd war.

Wir stellen hier einige der dabei auftretenden Probleme und unsere Lösungen in diesem Bereich vor. Um das ganze anschaulicher zu machen, betrachten wir als Beispiel eine Anwendung, in der personenbezogene Daten erhoben werden und die deshalb einen erhöhten Schutzbedarf hat. Außerdem kommuniziert unsere Anwendung über verschiedene Schnittstellen mit gesicherten Kundensystemen, die in einer traditionellen Bare-Metal-Art betrieben werden.

Es treffen also zwei verschiedene Welten und Denkweisen aufeinander, die es zu vereinbaren gilt. Technologisch ist das sehr spannend, weil an dieser Stelle mitunter Kreativität und Improvisation gefragt sind. Konkret haben wir es mit zwei Anforderungen zu tun:

- Aus Sicherheitsgründen soll analog zu den traditionellen Rechenzentren der ein- beziehungsweise ausgehende Netzwerkverkehr kontrolliert werden (Traffic-Control). Das bildet eine zweite Verteidigungslinie für den Fall, dass der Application-Layer von einem Angreifer durchbrochen wird („Defence-in-Depth“).
- Zur Kommunikation mit dem Kundensystem werden HTTPS-Schnittstellen verwendet, die hinter einer Firewall liegen. Diese Firewall braucht feste IPs für die Requests für ein Whitelisting. Das heißt, für den ausgehenden Traffic sollte es möglichst ein Gateway mit einer festen IP geben. Analog soll es auch möglich sein, zum Beispiel E-Mail-Server zu betreiben.

Kubernetes und seine Sicherheitsfeatures

Kubernetes erlaubt es, Container in Gruppen als sogenannte Pods zu deployen, deren Verhalten durch eine Ressourcen-Definition festgelegt wird, beispielsweise ein Deployment. Zur Kommunikation stellt Kubernetes seinen Pods eine eigene DNS und einen internen IP-Bereich zur Verfügung.

Einträge in diesem DNS-System werden üblicherweise über Services definiert. Ein Pod, der einen anderen Pod aufrufen will, erreicht ihn also über einen Service, der die Anfrage an einen passenden Pod weiterleitet. In der Basiskonfiguration werden dazu oft unverschlüsselte Verbindungen verwendet. Die Sicht- und Erreichbarkeit von Services und anderen Ressourcen lässt sich über Namespaces strukturieren und auch zu einem gewissen Teil kontrollieren. Dies soll hier der Einfachheit halber aber keine Rolle spielen. Die Ressourcen-Konfigurationen liegen in Form von yaml-Dateien vor und können über eine UI oder mittels kubectl über die Kommandozeile verändert werden.

Anfragen von außen erreichen die Pods entweder über spezielle Services oder häufiger über einen speziellen Proxy – den Ingress –, der die Anfragen auf verschiedene Services verteilt. Per Default ist die Kommunikation innerhalb des Clusters und vom Cluster nach außen vollständig offen. Zwar gibt es die Möglichkeit, per Network-Policies bestimmte Firewall-Regeln für den Cluster zu definieren. Leider hat dieser Mechanismus für sich genommen bisher aber einige Grenzen. Speziell uns fehlten hier die folgenden Features [kub]:

- Whitelisting nach Urls, bisher ist dies nur nach IPs von externen APIs möglich,
- Umleitung des ausgehenden Traffics über ein Ausgangsgateway.

Wir brauchen also noch eine andere Lösung.

Lösung für das Traffic-Control-Problem: Entscheidung für ein Istio-Service-Mesh

Mit regulären Kubernetes-Mitteln ist es relativ einfach, den eingehenden Netzwerkverkehr zu kontrollieren, da dieser ausschließlich über zu definierende Punkte in den Cluster gelangen kann. Solche Punkte können beispielsweise Load Balancer Services oder Ingresse sein. Um den ausgehenden Verkehr zu kontrollieren, könnte man zum Beispiel in jedem Pod eine Firewall installieren und alle Services im Cluster so behandeln, als ob sie offen über das Internet miteinander kommunizieren würden („Zero-Trust-Architecture“). Für unseren Anwendungsfall würden daraus jedoch diverse Nachteile resultieren. Insbesondere wäre der Konfigurations- und Wartungsaufwand sehr groß gewesen und die Komplexität hätte leicht zu sicherheitskritischen Fehlern geführt.

Wie oben beschrieben, bieten die von Kubernetes zur Verfügung gestellten Network-Policies leider keine ausreichenden Möglichkeiten, um die gewünschten Sicherheitsfeatures umzusetzen. Um dennoch mehr Kontrolle über die Kommu-

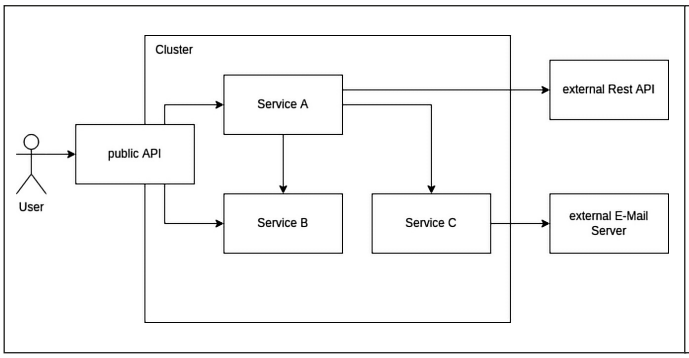


Abb. 1: Beispiel für ein kleines Microservice-System in einem Kubernetes-Cluster

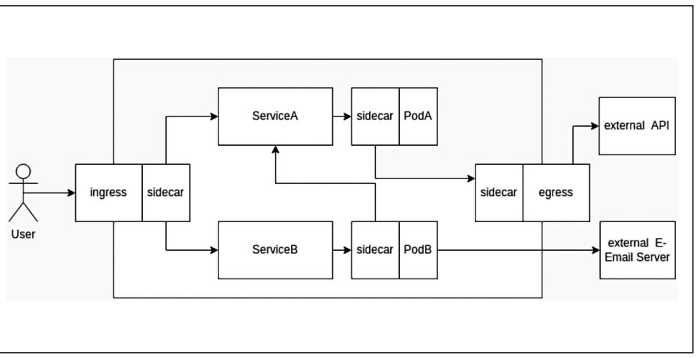


Abb. 2: Übersicht über ein Service-Mesh

nikation zwischen Services zu erlangen, bieten sich sogenannte Service-Meshes (**siehe Abbildung 2**) an. Eine gute Open-Source-Lösung für ein Service-Mesh ist Istio. Es ist eine Infrastruktursoftware mit Orchestrierungsfunktion, das sich nahtlos in Kubernetes einfügt.

In einem klassischen Service-Mesh wie Istio wird der Traffic in und von Containern über Seiten-Container („Sidecars") in ein abgesichertes Netzwerk geleitet. Über diesen Traffic hat man nun an zentraler Stelle sehr viel Kontrolle. Unter anderem ist es möglich, ausgehenden Traffic über einen sogenannten Egress, also ein Ausgangsgateway, auszuleiten und dort auch festzulegen, welche Ziele dieser Traffic haben darf, inklusive Whitelisting nach Urls. Ein Service-Mesh erfüllt damit genau die Anforderungen, die wir für Traffic-Control hatten. Als Bonus kann mit einem Service-Mesh auch die Kommunikation zwischen den einzelnen Services im Cluster verschlüsselt werden.

Istio startet und konfiguriert Envoy-Proxy-Server als Sidecars, Ein- und Ausgangsgateways (Ingresse oder Egresse). Außerdem wird die Kommunikation zwischen den Envoy-Proxies TLS verschlüsselt und Istio verteilt die Schlüssel dazu automatisch. Darüber hinaus loggt das Istio-Service-Mesh auch Metadaten über die ein- und ausgehende Kommunikation für jeden Kubernetes Pod.

Die Konfiguration läuft über von Istio definierte Kubernetes-Ressourcen, also zum Beispiel über yml-Dateien in einem von Istio definierten Format [istio-a].

Einrichtung von Istio

Für die grundlegende Einrichtung eines Istio-Service-Meshes auf einem Kubernetes Clusters gibt es zwei gute Möglichkeiten. Einerseits gibt es ein Kommandozeilentool von Istio – istioctl [istio-b]. Dieses Tool erlaubt nicht nur die Installation von Istio auf einem Kubernetes-Cluster, sondern hilft auch beim Debuggen und bei der Analyse des Service-Meshes. Genauere Informationen zur Konfiguration liefern Dokumentation und API-Referenz [istio-b] [istio-c] [istio-m].

Alternativ gibt es inzwischen Helm-Charts für die Istio-Installation [istio-e]. Dies ist zum Beispiel von Vorteil, wenn kein weiteres Tool auf dem Administrationsrechner installiert werden soll. Die Helm-Variante ist dabei neuer und unserer Erfahrung nach noch nicht ganz so ausgereift – beispielsweise mussten wir bei einer Testinstallation von Istio mit Egress-Gateway auf einem kleinen Cluster den Egress-Kubernetes-Service von Hand anpassen, damit sie wie gewünscht funktionierte. Die Konfiguration kann in beiden Fällen in versionierbaren Dateien abgelegt werden, über die auch Updates installiert werden können.

Einige Features von Istio müssen in dieser Konfiguration explizit hinzugefügt werden. Wenn, wie in diesem Text, der ausgehende Netzwerk-Traffic kontrolliert werden soll, ist dies insbesondere das Egress-Gateway.

Als letztes braucht der Kubernetes *Namespace*, in dem die Services und Pods laufen, das Label `istio-injection: enabled`, damit der Istio-Controller weiß, dass für die Pods Sidecars gestartet werden sollen.

Traffic-Control mit Istio

Um zu verstehen, wie sich mit Istio der ein- und ausgehende Netzwerk-Traffic kontrollieren lässt, ist es hilfreich, sich zunächst konzeptionell eine grobe Übersicht über die Funktionsweise von Istio zu verschaffen [istio-f].

Istio startet für jeden Pod einen Envoy-Sidecar-Container in entsprechend gekennzeichneten Namespaces des Clusters. Zusammen mit speziellen weiteren Envoy-Proxies für beispielsweise das Egress-Gateway bilden die Sidecar-Container das Service-Mesh-Netzwerk und agieren als Proxies, über die die Netzwerkkommunikation läuft.

Wenn nun innerhalb eines Clusters ein Microservice A mit einem anderen Microservice B kommunizieren möchte, wird eine Verbindung über seinen Sidecar zum Sidecar des anderen Microservice hergestellt (siehe **Abbildung 3**, erste Spalte). Diese Verbindung muss vorher über einen *ServiceEntry* erlaubt worden sein. Für Services innerhalb des Clusters wird ein solcher Eintrag automatisch hinzugefügt.

Steuerung externer Kommunikation mit Istio

Für die externe Kommunikation bietet Istio verschiedene Mechanismen. In der Grundeinstellung erlaubt Istio den Pods, mit unbekannten externen Services direkt zu kommunizieren und die Istio-Sidecars zu ignorieren.

Um die ausgehende Kommunikation besser zu kontrollieren, ist es möglich, dieses Verhalten so zu ändern, dass die Istio-Sidecars nur diejenige Kommunikation an externe Services zulassen, die vorher explizit erlaubt wurde [istio-g]. In der Grundkonfiguration des Istio-Service-Mesh-Netzwerks kann man festlegen, dass nur Kommunikation, die in zuvor definierten *ServiceEntries* definiert wurde, Istio-Sidecars verlassen darf.

Wenn ausgehende Kommunikation für spezifische externe Ziele, zum Beispiel Google, erlaubt werden soll, muss manuell ein eigener ServiceEntry hinzugefügt werden. Das muss ein ServiceEntry sein, der den Traffic direkt zu der gewünschten Adresse durchlässt. Dazu benötigt der ServiceEntry die Markierung `location: MESH_EXTERNAL` in seiner Ressource (siehe **Abbildung 3**, zweite Spalte). Mit diesem Ansatz wird zwar der Austritt von Information zu unerlaubten Zielen erschwert. Allerdings wird über diesen Mechanismus allein noch keine sichere Egress-Traffic-Kontrolle ermöglicht [istio-g].

Eine Alternative oder Ausbaustufe ist es, den Traffic, der an einen externen Service

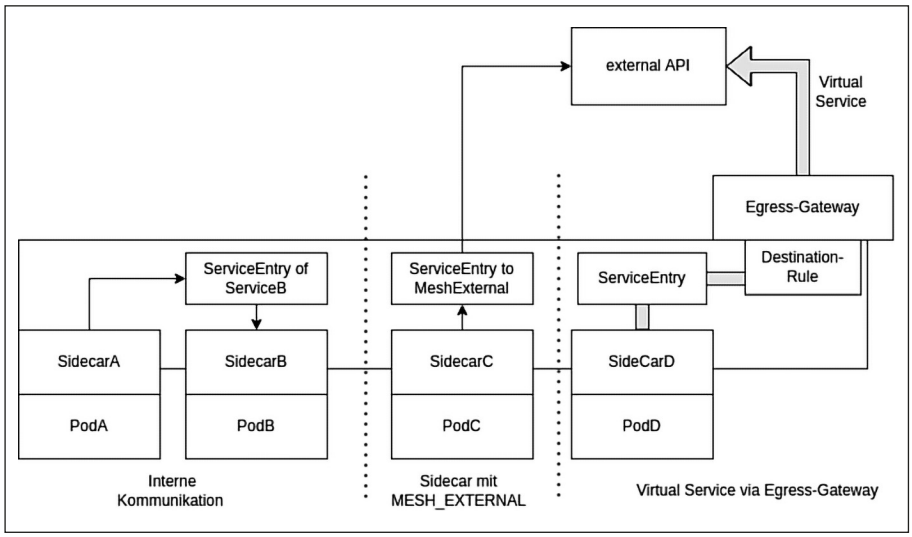


Abb. 3: Kommunikationsmöglichkeiten mit Istio

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: external-services-https
  namespace: default
spec:
  hosts:
  - google.com
  ports:
  - number: 443
    name: tls
    protocol: TLS
  resolution: DNS
```

Listing 1: ServiceEntry

gerichtet ist, nicht direkt vom Sidecar nach draußen, sondern über ein Istio-Egress-Gateway[1] zu leiten. Das ist beispielsweise dann interessant, wenn es zu den Sicherheitsanforderungen gehört, allen Traffic aus einem Service-Mesh heraus über ein bestimmtes Set von Nodes laufen zu lassen.

Die Einrichtung einer Datenverbindung über ein Egress-Gateway läuft über sogenannte VirtualServices, die den Traffic zunächst vom Sidecar zum Egress-Gateway und dann von dort weiter ins Internet leiten (siehe **Abbildung 3**, dritte Spalte). Eine solche Verbindung besteht aus vier Komponenten: Service-Entry, Destination-Rule, Egress-Gateway, VirtualService, auf deren Einrichtung wir im Folgenden kurz eingehen.

Als erstes muss in einem ServiceEntry – mit `resolution: DNS` – das Ziel des Traffics in dem Pod definiert werden (**siehe Listing 1**). Dieser ServiceEntry beschreibt, dass ausgehender Traffic mit dem Host „google.com“ über die Istio interne DNS weitergeleitet wird.

Außerdem muss der Gateway als internes Ziel im Istio-Service-Mesh registriert werden. Dafür benötigten wir noch die *Destination-Rule*-Ressource aus **Listing 2**.

Das Egress-Gateway muss darüber informiert werden, dass der Traffic für google.com:443 von ihm durchgelassen werden soll (**siehe Listing 3**). Dieses Gateway stellt das interne Ziel der Umleitung dar.

Als letztes wird die Umleitung des Netzwerk-Traffics in einem *VirtualService* festgelegt (**siehe Listing 4**).

Für den Pod, der einen mit Istio-Sidecar bestückten Container enthält, sieht der Kommunikationskanal aus wie ein regulärer Kubernetes-Service zum externen Ziel. Diese Verbindung wird in zwei Schritten aufgebaut: Im ersten Schritt wird Traffic aus dem Mesh auf Port 443 an das interne Ziel – das Egress-Gateway – geschickt. In der Konfiguration von Istio wird dann für das interne Ziel festgelegt, wie sich der Traffic am Ziel verhalten soll. Der einfachste Mechanismus, diese Definition abzubilden, ist das sogenannte *Subset*.

Im zweiten Schritt wird der Traffic, wenn er aus dem Egress-Gateway kommt, direkt an den Host im Internet weitergeleitet. Für weitere Informationen dazu siehe [istio-h].

Weitere Sicherheitsüberlegungen

Istio ergreift keine besonderen Sicherheitsmaßnahmen für die Nodes, auf denen der Egress-Service läuft. Außerdem können Pods auch ohne Sidecar-Container gestartet werden, das heißt, es ist für potenzielle Angreifer möglich, das Service-Mesh im Cluster komplett zu umgehen. Es kann also nicht garantiert werden, dass sämtlicher Traffic über das Gateway nach außen gelangt. [istio-h]

Innerhalb des Service-Meshes gibt es zwar Einstellungen, dass Istio jede Kommunikation blockiert, die nicht über einen expliziten ServiceEntry läuft (wie oben beschrieben). Diese Einstellung kann aber zu Problemen beim Starten bestimmter Services führen, zum Beispiel eines Nginx-Ingresses, eines Grafana Monitoring-Systems oder einiger Kubernetes-Operatoren. Zur Startprozedur dieser Services gehört ausgehende Kommunikation, die von Istio nicht blockiert werden darf. Das Istio-Sidecar, über das die Kommunikation gehen soll, startet aber normalerweise erst, wenn ein Service bereits läuft. Dadurch käme es zu einer Blockade, weil der Service nicht starten könnte.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: egressgateway-for-external-services-https
  namespace: default
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
    - name: external-services-https
```

Listing 2: Destination-Rule-Ressource

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-egressgateway
  namespace: default
spec:
  selector:
    istio: egressgateway
  servers:
    - port:
        number: 443
        name: tls
        protocol: TLS
      hosts:
        - google.com
      tls:
        mode: PASSTHROUGH
```

Listing 3: Traffic für google.com:443 durchlassen

[1] Um das Egress-Gateway zu aktivieren, muss es in der Istio-Konfiguration angegeben werden.

Deshalb kann es sinnvoll sein, dass Istio bei uns die Kommunikation erst einmal durchlässt und wir sie in zweiter Instanz über Netzwerkregeln blockieren oder selektiv für einzelne Services durchlassen (insbesondere für die genannten Services und den Istio-Egress-Gateway). Ein ähnlicher Ansatz wird auch von den Istio-Entwicklern empfohlen [istio-i].

Network-Policies als zweite Firewallschicht

Die Kubernetes *Network-Policies* konfigurieren die interne Kubernetes-Firewall. Wie oben beschrieben ist dieser Mechanismus zwar zu grob gestrickt, um alle Sicherheitsanforderungen an den Netzwerkverkehr zu erfüllen, aber als zusätzliche zweite Schicht kann er die noch existierenden Lücken im Istio-Service-Mesh schließen.
Wir blockieren nun mit *Network-Policies* alle ausgehende Kommunikation, mit Ausnahme folgender Verbindungen, die benötigt werden, damit unser System weiter funktioniert:

- Alle ausgehende Kommunikation aus Pods, die zu anderen Pods geht. Wichtig waren die Ports 53 für TCP und UDP, die in Richtung des Kubernetes-Systems-Namespaces gehen, damit die Kubernetes interne DNS funktioniert.
- Außerdem sollte Kommunikation an alle internen Pods und Services (insbesondere die, bei denen es Istio-Sidecars gibt) erlaubt sein. Istio verwendet dort eine Reihe von Ports im Bereich 15000 bis 15090. Die IP-Bereiche (CIDR), an die diese Kommunikation möglich sein sollte, entsprechen dabei einerseits dem Service-Subnet, das zum Beispiel in manchen Kubernetes-Clustern in kubeadm-config *Configmap* gefunden werden kann, und andererseits dem IP-Pool für die Pods zum Beispiel im Calico-System.
- Bestimmte ausgehende Kommunikation von Infrastruktur-Pods wurde während deren Startphase benötigt. Die Netzwerkregeln dafür wurden dann speziell für diese einzelnen *Deployments* oder *Statefulsets* angelegt.
- Eventuell noch Ports für interne Pod-to-Pod-Kommunikation (siehe unten).

Außerdem erlauben wir über eine zweite *Network-Policy* alle eingehenden Verbindungen. Denn eingehende Kommunikation aus dem Internet kann sowieso nur über unseren Ingress in das Kubernetes-System kommen, dessen Konfiguration für unsere Anwendung hinreichenden Schutz bietet.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: direct-google-through-egress-gateway
  namespace: default
spec:
  hosts:
    - google.com
  gateways:
    - mesh
    - istio-egressgateway
  tls:
    - match:
        - gateways:
            - mesh
          port: 443
          sniHosts:
            - google.com
      route:
        - destination:
            host: istio-egressgateway.istio-system.svc.cluster.local
            subset: external-services-https
            port:
              number: 443
    - match:
        - gateways:
            - istio-egressgateway
          port: 443
          sniHosts:
            - google.com
      route:
        - destination:
            host: google.com
            port:
              number: 443
          weight: 100
```

Listing 4: VirtualService

Interne Pod-zu-Pod-Kommunikation wurde als ausreichend sicher betrachtet.

Fallstricke

Die Umsetzung dieser Netzwerkregeln erwies sich als komplex und oft auch fragil. In der Praxis muss eine ganze Reihe von Ports für die interne Kommunikation freigegeben werden, die man in der Regel nicht sofort auf dem Schirm hat, denn dabei handelt es sich nicht um Datenbank- oder Applikationsports, die man selbst definiert hat. Bei uns mussten zudem mehrere Cluster-interne CIDR erlaubt werden (z. B. in unserem Falle für Calico und den Kube-System-Namespace), um die Cluster-interne Kommunikation nicht zu behindern. Diese internen IPs und Ports findet man im Zweifelsfall nur mittels systematischer Suche über das Ausschlussprinzip beziehungsweise über Trial-and-Error-Verfahren oder im Austausch mit dem Cloud-Provider, da sie meist spezifisch für den jeweiligen Anbieter sind. An dieser Stelle muss man daher mit einigem Zeitaufwand rechnen, der einmalig für die Einrichtung zu investieren ist.

Sobald man mit Netzwerkregeln in Kubernetes arbeitet, muss man natürlich darüber hinaus zum Beispiel auch den ausgehenden Traffic freischalten, der von einem Nginx Ingress-Controller nach außen geht. Ansonsten ist die Webapplikation nicht mehr von außen erreichbar.
Für Traffic, der nicht über die Protokolle HTTPS oder TLS mit Standard-Ports 80 oder 443 läuft (z. B. für E-Mail oder SFTP-Verbindungen), ist zudem ein Transport durch das Istio-Egress-Gateway derzeit nicht möglich. In diesem Fall bleibt nur die Option, sie als Mesh-External Services zu definieren und den Traffic direkt vom Sidecar zu verschicken. Der ausgehende Traffic kann trotzdem noch über Netzwerkregeln eingeschränkt werden.

Lösungsansätze für den Bedarf an festen IPs

Damit eine Cloud-Anwendung mit Services aus der alten Bare-Metal-Welt kommunizieren kann, benötigt sie manchmal eine feste IP-Adresse für den ausgehenden Traffic. Beispielsweise gibt es in bestimmten Unternehmen Rest-APIs, die hinter einer Firewall liegen, die alle Anfragen

Literatur & Links

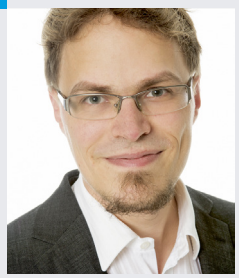
[istio-a] Istio/The Istio Service Mesh, siehe: **https://istio.io/latest/about/service-mesh/#what-is-istio**
[istio-b] Istio/Operator Installation, siehe: **https://istio.io/latest/docs/setup/install/operator/**
[istio-c] Istio/Customizing, siehe: **https://istio.io/latest/docs/setup/additional-setup/customize-installation/**
[istio-d] Istio/Operator, siehe: **https://istio.io/latest/docs/reference/config/istio.operator.v1alpha1/**
[istio-e] Istio/Helm, siehe: **https://istio.io/latest/docs/setup/additional-setup/customize-installation/#customize-istio-settings-using-the-helm-api**
[istio-f] Istio/How it works, siehe: **https://istio.io/latest/about/service-mesh/#how-it-works**
[istio-g] Istio/Accessing External Services, siehe: **https://istio.io/latest/docs/tasks/traffic-management/egress/egress-control/#security-note**
[istio-h] Istio/Egress Gateways, siehe: **https://istio.io/latest/docs/tasks/traffic-management/egress/egress-gateway/**
[istio-i] Istio/Kubernetes Network Policies, siehe:
https://istio.io/latest/docs/tasks/traffic-management/egress/egress-gateway/#apply-kubernetes-network-policies
[istio-j] Istio/http-proxy, siehe: **https://istio.io/latest/docs/tasks/traffic-management/egress/http-proxy/**
[istio-k] Istio/Security, siehe: **https://istio.io/latest/docs/concepts/security/**
[istio-l] Istio Blog - Egress Mongo, siehe: **https://istio.io/latest/blog/2018/egress-mongo/?_ga=2.46966375.1142339246.1649398695-1194130097.1645809715#direct-tcp-egress-traffic-through-an-egress-gateway**
[istio-m] Istio/Accessing External Services, siehe: **https://istio.io/latest/docs/tasks/traffic-management/egress/egress-control/**
[kub] Istio/Network Policies, siehe: **https://kubernetes.io/docs/concepts/services-networking/network-policies/**

blockieren, die nicht von einer bekannten IP kommen, oder für einen E-Mail-Server muss ein SPF-Eintrag in der DNS mit festen IPs hinterlegt werden.
Für dieses Problem gibt es mehrere Lösungsansätze. Aller Traffic kann über einen externen Proxy geleitet werden, der eine feste IP hat [istio-j]. Ein solcher extra Server verursacht natürlich zusätzlichen Wartungsaufwand.
Wenn es möglich ist, die IP einer Cluster-Node festzulegen, kann man die Deployment-Konfiguration des Istio-Egress-Gateways so ändern, dass er immer auf dieser Node gestartet wird. Damit käme automatisch aller Traffic, der dann über das Gateway geleitet wird, von dieser IP.
Je nach Clusteranbieter kann ein Festlegen einer Node-IP unmöglich sein. Für kleinere Cluster gibt es aber eventuell noch die Möglichkeit, eine Liste an IPs für die Cluster-Nodes festzulegen. Viele Anwendungen, zum Beispiel der SPF-Record, erlauben auch, mehrere IPs anzugeben und auf diese Weise das Zielproblem zu lösen. Für größere Systeme wäre dieses Verfahren jedoch nicht praktikabel, weil sehr viele feste IPs reserviert werden müssten, der SPF-Eintrag zu lang werden würde und das Whitelisting der IPs in den Firewalls der Partner zu Problemen führen würde.

Fazit

Service-Meshes wie Istio erlauben eine feingranulare Absicherung der Kommunikation innerhalb von Kubernetes-Clustern. Damit lassen sich über Istio grundlegende Sicherheitsanforderungen – wie Gateways, wie sie aus der „alten“ Server-Welt bekannt sind – an ein Kubernetes-Cluster mit vertretbarem Aufwand und relativ kostengünstig mit Open-Source-Mitteln umsetzen. Als Bonus bekommt man dabei auch eine Verschlüsselung der Kommunikation zwischen den einzelnen Services. Wie bei allen Netzwerkeinstellungen entsteht mit der zusätzlichen Sicherheit allerdings auch zusätzliche Komplexität.
Wir hätten uns gerne noch mehr damit beschäftigt, wie man am besten automatisiert und systematisch testen kann, ob die auf diese Weise definierte Netzwerk-Konfiguration den vorgegebenen Sicherheitsanforderungen genügt oder ob die Konfiguration noch Lücken aufweist. Ein Ansatz dafür, den wir bei uns im Team diskutiert, aber aus Zeitgründen nicht ausprobiert haben, sind eventuell automatisiert ausführbare API-Tests, die in regelmäßigen Abständen aus dem Cluster selbst heraus die prinzipielle Erreichbarkeit der benötigten externen Services abprüfen und die Blockierung anderer als der freigeschalteten externen Services testen. Die Tests könnten in einem leichtgewichtigen Docker-Container ausgeführt werden, der der gleichen Cluster-Konfiguration unterläge wie die tatsächlichen Applikations-Container. Im Falle, dass Tests fehlschlagen, könnte man gegebenenfalls auch über ein Monitoring-Tool wie Grafana/Loki automatisch Warnungen verschicken.
Es ist auch möglich, die komplette Authentifizierungs- und Autorisierungslogik von den Microservices in das Istio-Netzwerk auszulagern [istio-h]. Dies könnte gerade für größere Systeme sehr interessant sein, um den Wartungsaufwand zu reduzieren.
Service-Meshes haben noch viele weitere Anwendungsbereiche, zum Beispiel komplexeres Load-Balancing, die gesteuerte Umleitung eines Teils des Netzwerk-Traffics an andere Services oder sogar komplizierte Chaos-Monkey-Testverfahren, um die Stabilität von verteilten Anwendungen zu testen.
Wer in seinem Projekt derartige Anforderungen hat, für den lohnt sich ebenfalls ein Blick auf diese Technologie. Istio bietet für solche komplexeren Konfigurationen sogar die grafische Benutzerschnittstelle Kiali und erlaubt auch eine einfache Anbindung üblicher Cluster-Tools wie Grafana oder Jaeger. ||

Die Autoren



Dr. Thorsten Jahrsetz
(thorsten.jahrsetz@andrena.de)
arbeitet seit sechs Jahren als agiler Softwareentwickler bei der andrena objects ag. In verschiedenen Kundenprojekten beschäftigt er sich besonders mit dem Thema sichere Softwareentwicklung und sicherer Betrieb im agilen Umfeld.

Martin Nitsch
(martin.nitsch@andrena.de)
arbeitet seit vier Jahren als agiler Software-Engineer bei der andrena objects ag. Als solcher setzt er sich für Clean Code, TDD und CI/CD ein.