Microservices? Mit Sicherheit!

Claus Straube, Landeshauptstadt München

In jedem produktiven Software-System spielt Sicherheit eine zentrale Rolle. Warum sollte dies in einer Microservice-Architektur anders sein? Allerdings ist es im Vergleich zu einer monolithisch geprägten Architektur in einem verteilten System deutlich komplexer, die Anwendung adäquat abzusichern. Dieser Artikel diskutiert die Themen „Anwendungsdesign", „Authentication" und „Authorization" sowie die Angriffserkennung aus Architektur-Sicht.

Microservices finden immer mehr Verwendung in produktiven Systemen. Das liegt nicht nur an ihren Vorzügen, wie punktgenaue Skalierbarkeit oder – bedingt durch ihre Größe und lose Kopplung – bessere Wartbarkeit, sondern vor allem daran, dass in den letzten Monaten ein Ökosystem von Werkzeugen aus dem Boden geschossen ist, das die Nachteile einer verteilten Anwendungslandschaft zumindest abmildert. Eine dieser Herausforderungen, die produktive IT-Systeme immer mit sich bringen, ist Security. Daran hat sich auch mit dem Microservice-Hype nichts geändert. Nach wie vor will man als Systembetreiber bestimmen, wer was in der Anwendung tun darf.

Die Architektur

Der erste (und einfachste) Schritt in Richtung einer sicheren Microservice-Anwendung ist es, die Angriffsfläche so klein wie möglich zu halten. Im Vergleich zu einer monolithischen Architektur wächst diese bei einer verteilten Anwendungsarchitektur von Natur aus. Man hat viele unabhängige Services, von denen jeder ein API hat, und diese kleinen Anwendungen kommunizieren auch noch über das Netzwerk miteinander. Das erhöht logischerweise die Fläche. Glücklicherweise lässt sich die Angriffsfläche mit ein paar einfachen Mitteln recht gut verkleinern.

Grundsätzlich sollte die Anwendung in einer sicheren Umgebung ausgebracht sein. Der Zugriff von außerhalb ist also beispielsweise durch eine Firewall limitiert. Natürlich sollten sowohl auf der Firewall als auch auf den Containern beziehungsweise virtuellen Maschinen, auf denen die Services laufen, alle Ports geschlossen sein, die nicht unbedingt benötigt werden. Wenn man sich an die Regel hält, jeden Service in einem eigenen Container zu installieren, wird man normalerweise mit einem offenen Port auskommen. Die Kommunikation zwischen den Services beziehungsweise zwischen den Maschinen sollte über „https" verschlüsselt sein. Für den Entwickler ist das dank Werkzeugen wie Spring Boot oder Kubernetes kein großer Aufwand mehr. Die größte Herausforderung liegt tatsächlich in der Verwaltung der Zertifikate.

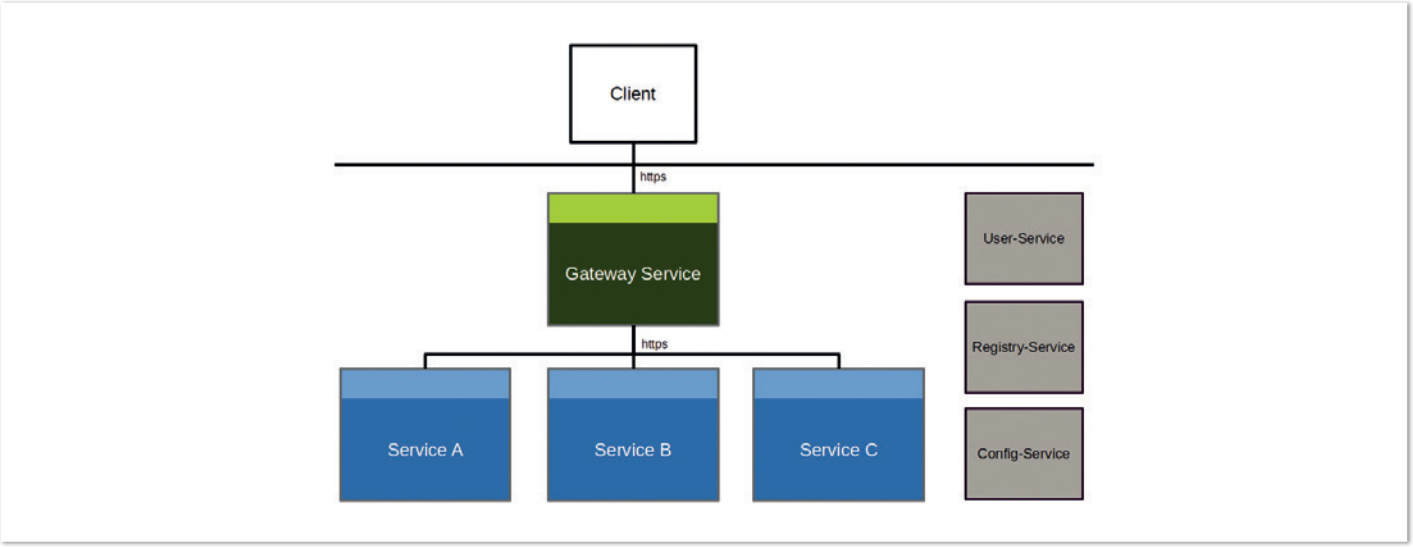

Abbildung 1: Microservice-Architektur mit API-Gateway

Ein weiterer wichtiger Aspekt ist die Verschattung der Schnittstellen. In der Regel wird die Gesamtheit aller Services mehr Schnittstellen zur Verfügung stellen, als von außen verwendet werden. Dies kann zum Beispiel daran liegen, dass bestimmte Operationen ausschließlich von anderen (internen) Services aufgerufen werden. Um auch hier die Angriffsfläche möglichst klein zu halten, ist es notwendig, die öffentlich verfügbaren Schnittstellen an einer zentralen Stelle zu bündeln.

Ein zusätzlicher Effekt durch diese Bündelung ist, dass von außen keine Rückschlüsse auf das Deployment gemacht werden können. Der Client muss nicht „n" Service-Adressen aufrufen, sondern genau eine. Um dieses API-Gateway-Pattern *(siehe „http://microservices.io/patterns/apigateway.html")* anzuwenden, gibt es bei Spring Cloud Netflix *(siehe „https://cloud.spring.io/spring-cloud-netflix")* den Zuul-Service. Dieser ist selbst ein Spring-Boot-Service und wird mit der Annotation „@EnableZuulProxy" in Verbindung mit den entsprechenden Abhängigkeiten im Klassenpfad erstellt.

Im einfachsten Fall kann der Zugriff auf die anderen Services durch URL-Pattern-basierte Routes konfiguriert werden. Wenn man den vollen Funktionsumfang von Zuul nutzen will, dann hat man auch die Möglichkeit, auf die Routen programmatisch zuzugreifen *(siehe Abbildung 1)*.

Bei sehr hohen Anforderungen an die Sicherheit können auch zwei API-Gateways zum Einsatz kommen. Das erste bündelt den Zugriff auf die öffentlichen Services, das zweite den auf die privaten Services. Bei dieser Architektur wird man aber gegebenenfalls beim fachlichen Design der Microservices Zugeständnisse an das Deployment machen müssen. In der ersten Zone werden also zwangsläufig vor allem Services zur Orchestrierung platziert werden. Von einem Schnitt nach der reinen Domain-Driven-Design-Lehre wird man in diesem Fall etwas abweichen müssen. Wichtig ist, dass in beiden Zonen jeweils auch eine separate Authentication verwendet wird. Das heißt, jede Zone hat ihre eigene, übergeordnete Security.

Der Security-Context

In einer monolithischen Anwendungs-Architektur besteht die Anwendung aus einem einzigen Sicherheitskontext. In diesem spielen sich praktisch alle sicherheitsrelevanten Vorgänge ab. Dort werden die Rollen und Rechte verwaltet (sofern man nicht mit einem Identity-Management-System arbeitet) und an jeder Stelle im Verfahren ist klar, wer eine Aktion gerade ausführt.

Dies ist in einer Microservice-Architektur fundamental anders. Hier ist jeder Service autark. Er kümmert sich nach dem „Do One Thing and Do it Well"-Prinzip um seine Aufgabe. Natürlich sind (beziehungsweise sollten) die einzelnen Services abgesichert sein, aber eben jeder für sich. Ein großer gemeinsamer Security-Context wie in einer monolithischen Architektur kommt so nicht zustande. Er sollte vielleicht auch gar nicht so existieren, weil ein Service unter Umständen von mehr als einer Anwendungsdomäne genutzt werden könnte – also gar nicht, wie in der klassischen monolithischen Denkweise, zu einer einzelnen Anwendung gehört. Dieser dezentrale Ansatz macht Security in einer Microservice-Architektur zur Herausforderung.

Authorization

Wo werden die Nutzer, Rollen und Rechte verwaltet? Wenn man sich überlegt, welchen Aufwand es nach sich ziehen würde, die Zuordnungen „Nutzer zu Rolle" und „Rolle zu Recht" in vielleicht fünf unterschiedlichen Services zu pflegen, und dann den Gedanken weiter spinnt, wie das mit hundert Services aussehen würde, kommt man schnell zum Schluss, dass es hierfür eine zentrale Stelle geben muss: einen User-Service nach dem Motto „Do One Thing and Do it Well", der rein für das Mapping „Nutzer auf Rolle" und „Rolle auf Recht" zuständig ist.

Woher der User-Service die Nutzer- und Rollen-Daten bezieht, ist hierbei zweitrangig. Im einfachsten Fall sind die Daten im Speicher vorgehalten oder sie kommen aus einem Lightweight Directory Access Protocol (LDAP) oder Identity-Management. Selbst Mischlösungen sind vorstellbar. Das heißt, Nutzer werden in einem LDAP verwaltet, die Rollen und das Mapping in einer Datenbank des Service. Hier kann man sämtliche Freiheiten, die einem Frameworks wie Spring Security bieten, nutzen, um die Anforderungen bestmöglich abzudecken. Die Rechte kommen jedoch immer aus den einzelnen Microservices, denn das Recht ist dort mit einem Stück Code verbunden, der nur ausgeführt wird, wenn der Nutzer das Recht

besitzt. Damit ist klar, dass es wichtig zu wissen ist, welche Person (oder gegebenenfalls Maschine) hinter einem Aufruf steckt. Nur so können die Rechte im Service aufgelöst werden.

Oftmals wird ein Request nicht von genau einem Service abgearbeitet, sondern er löst eine Kaskade von Anfragen an unterschiedliche Services aus. Die Kommunikation hinter dem ersten Service könnte natürlich durch technische User durchgeführt werden. Ein detailliertes Berechtigungssystem auf allen Service-Ebenen ist dann allerdings nicht mehr möglich. Besser wäre es, wenn die Identität des aufrufenden Nutzers bei jedem Aufruf mitgegeben würde.

Authentication

Hier stellt sich die Frage, wie man die Rechte an den Aufrufer binden kann und – mindestens genauso spannend – wie diese Daten von Service zu Service transportiert werden. Eine Möglichkeit, dies einfach in einer verteilten Service-Wolke einzubauen, bietet hier OAuth2. Zwar ist OAuth2 („the industry-standard protocol for authorization", *siehe „https://oauth.net/2"*) ein Authorization- und kein Authentication-Protokoll – aber es erzeugt für einen Benutzer einen eindeutigen Token. Diesen bekommt der Nutzer nur, wenn er sich vorher auch authentifiziert hat. Der Token kann bei „http"-Aufrufen im Nachrichten-Header weitergegeben werden. Dadurch lässt sich in einer verteilten Anwendung sehr einfach ein Authentication-Mechanismus aufbauen, der die zuvor festgelegten Anforderungen perfekt erfüllt.

Wer sich im Spring-Ökosystem bewegt, kann sich über die Unterstützung freuen, die Spring Cloud Security *(siehe „https://cloud.spring.io/spring-cloud-security")* von Hause aus mitbringt. Mit wenigen Annotationen ist hier ein OAuth-Server (dieser kann natürlich auch von jedem Nicht-Spring-Service genutzt werden) auf Basis einer Spring-Boot-Anwendung erstellt.

Genauso schnell sind die von den Microservices angebotenen Endpunkte als OAuth-Ressourcen mit „@EnableResourceServer" gekennzeichnet. Dadurch wird der gesamte Service automatisch abgesichert. Um in den einzelnen Services nun automatisch bei Aufruf einen Security-Context aufbauen zu können, muss dem OAuth-Server noch ein „User Info Endpoint" hinzugefügt werden. Auf Seiten der Microservices ist dieser User-Info-Endpoint noch mit „security.oauth2.user-info-uri" zu konfigurieren, damit er gefunden werden kann. Wie bei Spring üblich, funktionieren die Dinge mit einer Basiskonfiguration einfach. So auch hier: Bei kaskadierenden Service-Aufrufen sorgt Spring dafür, dass der Token über ein „OAuth2Rest"-Template automatisch weiter propagiert wird *(siehe Abbildung 2)*.

Ein Nachteil dieser Methode ist allerdings, dass jeder Service-Aufruf gleichzeitig einen Request gegen den User- beziehungsweise OAuth-Service erzeugt. Das ist aus zwei Gründen kritisch:

- Die vielen Requests sind natürlich eine Belastung für Netzwerk und User-Service. Je mehr Aufrufe erfolgen und je mehr Services von diesen betroffen sind, desto höher wird die Belastung.

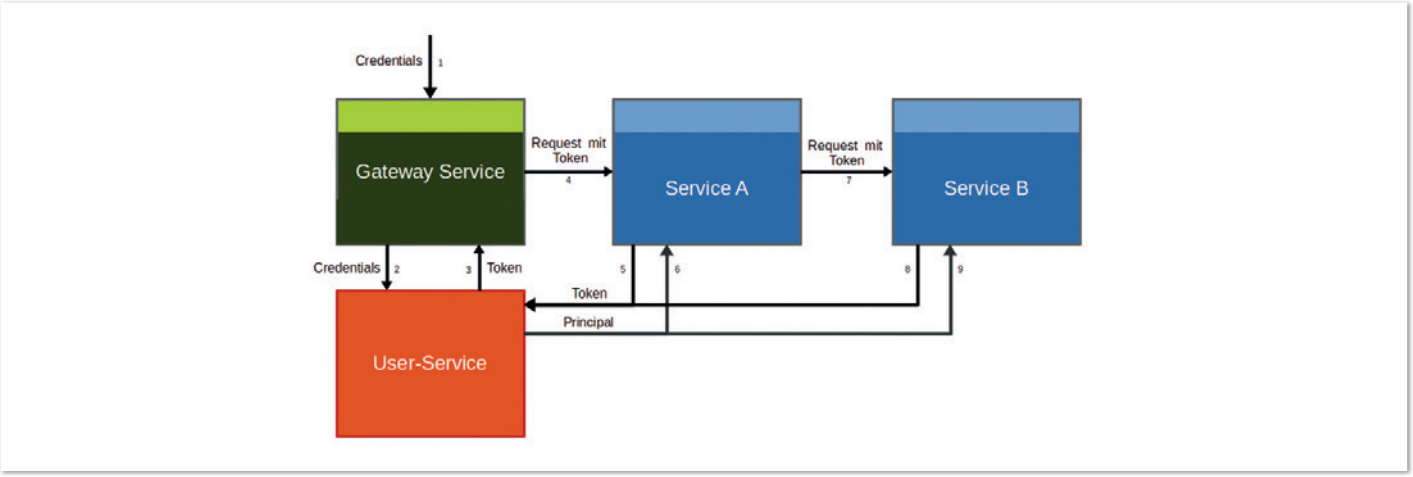

Abbildung 2: Kommunikation mit User-Endpoint pro Request

- Der User-Service ist eine absolut kritische Ressource; ist diese nicht vorhanden, wird potenziell nichts funktionieren.

Die Probleme lassen sich natürlich durch die eine oder andere Maßnahme, wie beispielsweise ein Cache auf Seiten der Ressourcen-Server, abmildern. Ganz eliminieren kann man sie allerdings nicht. Es gibt noch eine weitere Möglichkeit, Autorisierung mit OAuth2 umzusetzen. Die Spezifikation lässt dem Nutzer recht viele Freiheiten bei der Wahl des Tokens. Entsprechend viele Geschmacksrichtungen gibt es hier auch. Die beschriebene Methode funktioniert mit einem einfachen Bearer-Token *(siehe „https://tools.ietf.org/html/rfc6750").* Dieser besteht aus einer zufälligen Zeichenfolge und hat sonst keine semantische Bedeutung.

Es gibt allerdings auch sogenannte „strukturierte Token" wie den JSON-Web-Token (JWT, *siehe „https://tools.ietf.org/html/rfc7519").* Er besteht immer aus drei Teilen, einem Header, der Payload und der Signatur, wobei die drei Teile jeweils durch einen Punkt getrennt sind. Wird der JWT verschickt, dann werden Header und Payload per „Base64" encodiert und der gesamte Token per „HMAC" oder „RSA" signiert. Die Signatur stellt Integrität und Authentizität des Tokens sicher, ohne nochmals beim Server anfragen zu müssen. Interessant am JWT ist, dass man beliebige Dinge in der Payload mitschicken kann, beispielsweise auch Informationen über den Nutzer.

Genau das passiert, wenn man die JWT-Funktionalität von Spring Cloud Security nutzt. Im User-Service wird automatisch ein JWT mit allen notwendigen Informationen zum eingeloggten Nutzer erzeugt. Neben dem Namen des Users werden dort beispielsweise alle „Authorities" gespeichert, also die Rechte, die in der Methoden- oder Klassen-Security verwendet werden, die dem Nutzer zugeordnet werden können. Verwendet man Spring auch auf Seite der OAuth-Ressourcen, dann wird der lokale Security-Context des Service automatisch, auf Basis der Informationen, die im JWT gespeichert wurden, erzeugt. Ein Request Richtung User-Service ist an dieser Stelle nicht mehr notwendig.

Es gibt allerdings eine kleine Einschränkung: In einem JWT können zwar prinzipiell beliebig viele Daten gespeichert werden – da der Token aber im „http"-Header transportiert wird, muss man sich hier den Rahmenbedingungen des empfangenden Servers beugen. Aktuell können die meisten Server standardmäßig um die sieben bis acht KB verarbeiten. Ob das reicht, hängt sehr stark von der Größe der Anwendung und noch viel mehr vom Rollen- und Rechte-Konzept ab.

Grundsätzlich sollte man in der IT versuchen, volatile Dinge möglichst aus dem Code herauszuhalten. Die Zuordnung von Rollen auf Rechte oder von Nutzern auf Rollen ist so etwas Flüchtiges. Um trotzdem möglichst viel Handlungsfreiheit zu haben, kann man an der Granularität der Rechte ansetzen. Diese können beispielsweise auf Klassen- oder sogar auf Methoden-Ebene vergeben werden. Je weniger Code von einem Recht abgesichert wird, desto größer sind später die Freiheitsgrade bei der Konfiguration der Rollen.

Bei Spring heißen diese Rechte „Authorities". Das sind genau die Daten, die über den JWT geliefert werden müssen, um den Security-Context zu erzeugen. Hier zeigt sich das Problem. Hat man in der Anwendung Rechte mit sprechenden Namen wie „contractservice__calculate_insurance_risk" vergeben, was ja durchaus hilfreich bei der Zuordnung zu Rollen ist, dann kann man sich leicht ausrechnen, dass bei 150 bis 250 Rechten je Nutzer ein Problem auftritt. Diese Zahl ist bei kaskadierenden Service-Aufrufen in Verbindung mit einem feingranularen Rechte-Konzept schnell erreicht.Lösen lässt sich dieses Problem, indem man statt Rechten Rollen überträgt. Ein Nutzer wird in der Regel nur wenige Rollen haben. Die Wahrscheinlichkeit, dass man mit dieser Methode auf technische Limitationen stößt, ist relativ gering. Allerdings verlässt man damit bis zu einem gewissen Grade die Infrastruktur, die einem Spring per Konvention bietet. Die Rollen in den Token zu bekommen, ist dabei noch nicht das Problem. Das lässt sich auf Seite des User-Service durch eine einfache Query (sofern eine Datenbank genutzt wird) in einer „AuthenticationConfigurer"-Klasse lösen, die Rollen statt Rechte ausliest.

Im User-Service muss zusätzlich ein Endpunkt geschaffen werden, über den das „Rollen zu Rechte"-Mapping abgerufen werden kann. Auf Service-Seite sind allerdings mehr Dinge anzupassen. Einmal müssen die „Rollen zu Rechte"-Mappings vom User-Service in die Services übertragen und dann im Speicher vorgehalten werden. Dies kann per „Push" immer dann passieren, wenn Änderungen am Mapping vorgenommen werden. Alternativ könnten die Services die Informationen periodisch per „Pull" vom User-Service abrufen. Zudem ist in der Security-Pipeline dafür zu sorgen, dass die im Token mitgelieferten Rollen beim Aufbau des Security-Context in die richtigen Rechte umgewandelt werden *(siehe Abbildung 3).* Hat man diese Bedingun-

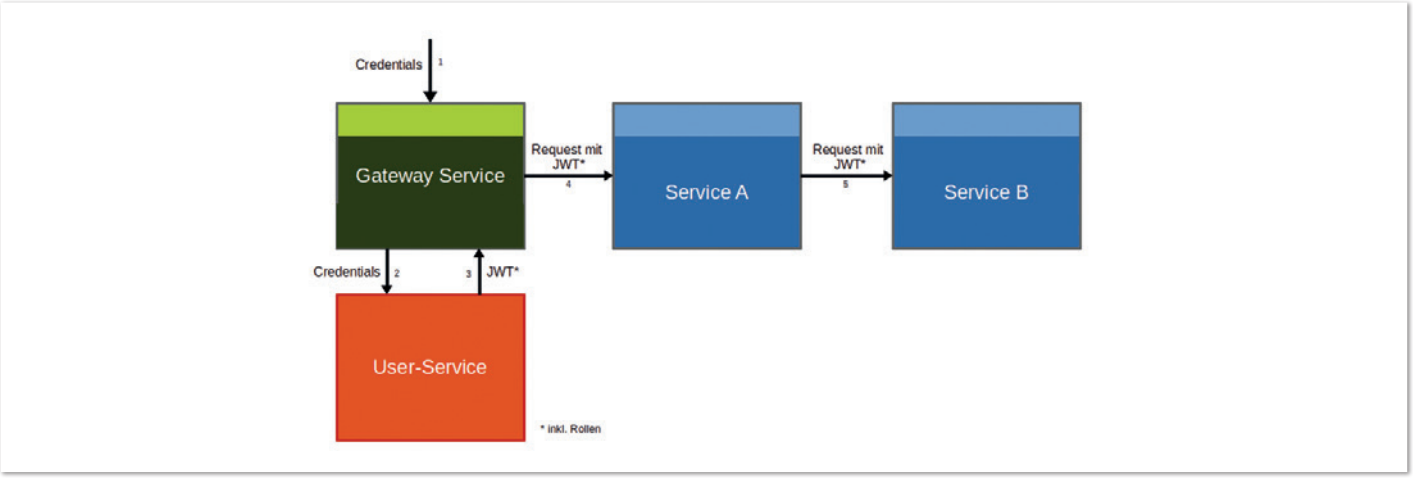

Abbildung 3: Kommunikation mit JWT pro Request

gen erfüllt, kann sich der Nutzer nach der ersten Anmeldung frei im System bewegen, ohne auf einen User-Service angewiesen zu sein.

Angriffserkennung

Neben den beschriebenen, passiven Vorkehrungen wäre es natürlich auch interessant, Angriffe erkennen und aktiv Gegenmaßnahmen ergreifen zu können. Im einfachsten Fall könnte man beispielsweise einen verdächtigen User sperren. Doch wie erkennt man, dass sich ein Nutzer verdächtig verhält? Dafür gibt es „Intrusion Detection"-Systeme wie AppSensor *(siehe „http://appsensor.org")* vom Open Web Application Security Project (OWASP, *siehe „https://www.owasp.org")*. Diese Systeme sammeln aus unterschiedlichsten Quellen Informationen und fügen sie zu einem „Lagebild" zusammen. Auf dessen Basis müssen dann die entsprechenden Maßnahmen zur Abwehr von Angreifern abgeleitet werden.

AppSensor bietet hier nicht nur die Software, sondern auch ein konzeptuelles Basis-Framework mit einer Beschreibung typischer „Detection Points". Das sind Ereignisse, die in einer Web-Anwendung auftreten können und dabei helfen, die Lage einzuschätzen. Auf Entwurfsebene unterscheiden sich die Detection Points einer monolithischen und einer verteilten Anwendung wenig. Auch die Frage, wie man an die gewünschte Information kommt, ist relativ ähnlich. Verwendet man in beiden Architektur-Varianten beispielsweise das Spring-Framework, so wird man viele Daten aus den zahlreichen Spring-Events ziehen können. Der größte Unterschied liegt wohl im Transport der Nachrichten von dem Punkt, am dem ein Ereignis aufgetreten ist, zum zentralen Sensor-Service. In einem monolithischen System können die Ereignisse in der Anwendung gesammelt und zum ID-System geleitet werden. Bewegt man sich in einer verteilten Anwendung, so sind die sicherheitsrelevanten Events in voneinander unabhängigen Klein-Anwendungen verteilt. Jeder dieser Services muss also selbst dafür sorgen, dass sicherheitsrelevante Nachrichten beim Sensor-Service ankommen.

Wie so oft gibt es natürlich für dieses Problem mehrere Lösungen. Verwendet man ein „Security Information and Event Management"-System (SIEM), so hat man hier oft die Möglichkeit, Logfiles aus unterschiedlichen Quellen einlesen und zentral auswerten zu lassen. Einen leichtgewichtigen Ansatz bietet Spring Cloud Stream *(siehe „https://cloud.spring.io/spring-cloud-stream")* – vorausgesetzt, man bewegt sich im Spring-Ökosystem.

Die Idee hinter Spring-Cloud-Stream ist, auf Basis einer Messaging Middleware (etwa RabbitMQ, *siehe „https://www.rabbitmq.com")* Annotationen anzubieten, mit denen Event-Nachrichten in einer verteilten Anwendung versendet werden können, ohne dass Sender und Empfänger voneinander wissen müssen. Der Sender versendet also ein Event und ein an den Bus angeschlossener Empfänger nimmt sich die Nachrichten, die interessieren.

Im Falle von ID würden die Services ihre Detection-Point-Events verschicken, der Sensor-Service würde sie aufnehmen. Der große Vorteil einer Bus- gegenüber einer Logfile-Lösung ist, dass sich die Ergebnisse praktisch in Echtzeit auswerten lassen. Damit kann auch ohne zeitliche Verzögerung automatisiert auf einen Angriff reagiert werden.

Fazit

Wie an den drei Beispielen gezeigt, hat der Microservice-Trend nichts an den grundsätzlichen Security-Design-Prinzipien geändert – sehr wohl aber daran, wie die Dinge umgesetzt werden müssen. Das muss allerdings nicht immer auch viel aufwändiger sein; mit den richtigen Werkzeugen lassen sich hier sehr smarte und alltagstaugliche Lösungen erzielen.

Claus Straube
claus.straube@gmail.com

Claus Straube ist seit dem Jahr 2011 als IT-Architekt für Java-Architektur und Anwendungs-Integration beim internen IT Dienstleister (itM) der Landeshauptstadt München (LHM) angestellt. Zusätzlich berät er freiberuflich Organisationen in Architekturfragen – insbesondere bei Integrationsthemen. Vor seiner Zeit bei der LHM war er als Entwickler und Architekt bei Oracle, Kabel Deutschland und einem Startup beschäftigt.