Microservices und Sicherheit

Aufsatz zum Seminar Komponenten, Agenten und Workflows in verteilten Systemen

Felix Ortmann und Konstantin Möllers
{0ortmann,1kmoelle}@informatik.uni-hamburg.de

16. Januar 2016

Zusammenfassung

Mit der stetig zunehmenden Diversifikation von Informationssystemen im Internet ist ein Bedarf für eine verteilte und skalierbare Architektur aufgekommen. So entstand etwa der neue Trend zur Microservice-Infrastruktur. Allerdings birgt diese Risiken in Bezug auf Sicherheit, da sowohl Architektur als auch Kommunikation eines Microservice-Netzwerks Sicherheitsprobleme aufwerfen können. In diesem Aufsatz gehen wir daher auf diese Probleme ein und zeigen für drei Schutzziele Lösungsmöglichkeiten auf. Es wird gezeigt, dass es bereits Möglichkeiten und neuartige Technologie gibt um sich vor Angriffen zu schützen.

1 Einleitung

Die Verschiedenartigkeit von Informationssystemen im Internet nimmt stetig zu. Waren es früher noch einfache Homepages um Inhalte auszuliefern, verlagern sich heutzutage komplexe Anwendungen zunehmend ins Web. Die Vorteile liegen dabei auf der Hand: Das Ausrollen neuer Software wird zum Kinderspiel und auch das Warten wird leichter. Allerdings wird das zentrale Softwaresystem auch mehr zum *Single Point of Failure*, welcher der erhöhten Belastung nicht standhält. Konsequenterweise ist der Bedarf für eine verteilte und skalierbare Architektur aufgekommen. So entstand etwa der neue Trend zur Microservice-Infrastruktur.

Mit diesem neuen Konzept kamen allerdings auch neue Risiken in Bezug auf Sicherheit auf. Denn sowohl die Architektur eines Microservices als auch die Kommunikation untereinander ermöglicht neue Sicherheitsprobleme. In der vorliegende Arbeit werden diese erläutert und diskutiert, um Lösungswege für neuartige Microservice-Netzwerke aufzuzeigen.

Dazu strukturiert sich die Arbeit wie folgt. Zunächst wird in Abschnitt 2 das Konzept der Microservices erläutert und dessen Sicherheitsperspektive motiviert. Weitergehend geht Abschnitt 3 auf Anwendungsarchitekturen von Microservices ein und erläutert dessen Sicherheitsaspekte. Abschnitt 4 überträgt dieses Vorgehen auf die Kommunikation zwischen Microservice-Instanzen. In Abschnitt 5 werden die drei primären Schutzziele diskutiert und Zuletzt fasst Abschnitt 6 diese Arbeit zusammen.

2 Einführung

Zunächst führen wir allgemein in das Prinzip der Microservices ein. In diesem Abschnitt werden wir in Unterabschnitt 2.1 darauf eingehen, wie die Funktionsweise von Microservices beschrieben werden kann. Daraufhin grenzt Unterabschnitt 2.2 ab, wie sich diese von anderen Modellen unterscheiden. Andererseits motivieren wir den Gegenstand dieser Arbeit, indem wir in Unterabschnitt 2.3 bestimmte Eigenschaften von Microservices hinsichtlich der Sicherheit untersuchen.

2.1 Eigenschaften von Microservices

Um die Funktionsweise von Microservices zu erläutern, betrachten wir die Definition nach (Newman, 2015), nach der „Microservices kleine, autonome Dienste sind, welche Zusammen Aufgaben absolvieren“. Wir analysieren im Folgenden diese drei Eigenschaften auf ihre Kernaussagen.

Zunächst handelt es sich um „kleine“ Dienste. Die Grundeigenschaft, die damit ausgedrückt werden soll, ist eine ausgeprägte **Kohäsion** der Software des Microservices. Damit wird Robert Martins *Single Responsibility Principle* ausgedrückt, welcher beschreibt, dass Software, welche aus der gleichen Intention agiert, zusammengefügt werden soll und dass andere Teile der Software voneinander zu trennen sind. Das Resultat sind kleine bzw. kleinstmögliche Dienste, die einem wohlbestimmten Zweck dienen.

Die zweite geforderte Eigenschaft, die der **Autonomie**, besagt dass Microservices selbständig sind. Dies betrifft zum einen die Ausführung und Installation des Services, da er unabhängig von anderen Diensten gewartet, installiert und ausgeführt werden kann, ohne das gesamte Informationssystem zu brechen. Zum anderen betrifft es aber auch die Art und Weise, wie der Service entwickelt wird. Denn nach (Fowler & Lewis, 2014) wird ein gesamtes Entwicklungsteam entsprechend der Aufgabe aus verschiedenen Experten für Bedienoberfläche, Datenbank, DevOps o.ä. zusammengesetzt. Dementsprechend kann der Microservice auf alle erforderlichen Ressourcen zurückgreifen ohne von anderen Services abhängig zu sein.

Die Autonomie der Microservices eröffnet auch weitere Möglichkeiten für die Entwicklung. Etwa kann jeder Microservice in einer anderen Programmiersprache geschrieben werden, wenn diese dem Anwendungsfall besser dient. So kann für rechenintensive Aufgaben eine nebenläufige Sprache wie *Clojure*[1] verwendet werden oder es kann für einfache Aufgaben auf leichte Skriptsprachen wie etwa *PHP*[2] zurückgegriffen werden.

Zuletzt erfordert Newmans Definition die Eigenschaft der **Kooperation**. Dies ist bedingt durch die harten Grenzen, welche ein Microservice aufwirft, die eine Kommunikation der Dienste untereinander erzwingt. Um ein gemeinsames Ziel zu erreichen, beispielsweise einen Aufgabenablauf zu erfüllen, der heterogene Teilaufgaben vorsieht, müssen die Dienste zusammen arbeiten. Die Kommunikation unter den Diensten erfolgt in festgelegten Protokollen, wie etwa *Message Queueing* oder *RESTful APIs*. Abschnitt 4 expliziert, welche Protokolle dafür in Betracht gezogen werden können.

Die Verwendung abgesprochener Kommunikationsmittel indes ermöglicht das replizieren verschiedener Dienste. Denn die übertragenen Nachrichten können von einer Lastverteilung ausgelesen und auf verschiedene Knoten übertragen werden. Dadurch erreichen

[1] http://clojure.org/
[2] http://php.net/

Microservices eine sehr hohe horizontale Skalierbarkeit, da sie quasi beliebig oft instantiiert werden können.

Nachdem wir die drei wichtigsten Eigenschaften von Microservices erläutert haben, werden wir im nächsten Abschnitt deren Prinzip von weiteren Architekturmodellen abgrenzen und motivieren.

2.2 Abgrenzung zu anderen Modellen

Wenn man den Begriff „Microservice“ betrachtet, wird man schnell einen Bezug zu **serviceorientierten Architekturen** (SOA) erwarten. Wie bei Microservices sieht SOA eine Trennung der Anwendungslogik in funktionale Anwendungsteile vor. Diese muss aber im Gegensatz zu Microservices allerdings nicht hart sein, sondern auch innerhalb einer monolithischen Anwendungsstruktur erfolgen. Ebenso sieht SOA vor, dass die Benutzeroberfläche nicht Gegenstand der Services sei. Microservices hingegen postulieren eine Aufteilung der Oberfläche und betrachten diese als Bestandteil eines Dienstes.

Große Ähnlichkeiten weisen Microservices ebenfalls zu **Multiagentensystemen** auf. Dies ist vor allem durch die Eigenschaften eines gerichteten Zwecks, der Autonomie und der Kooperation geschuldet. Dennoch muss man beachten, dass Microservices vielmehr durch pragmatische Ansätze entstanden sind und daher die soziologisch-theoretischen Prinzipien vermissen lassen, welche die agentenorientierte Softwareentwicklung auszeichnet. Darüber hinaus finden wir in Microservices soziotechnische Eigenschaften, die wir in klassischer Agentenorientierung nicht finden. Etwa die Vorgabe von getrennten Entwicklungsteams um die Struktur der Anwendung auf die Struktur der entwickelnden Organisation zu übertragen.

Es gibt allerdings solche Ansätze wiederum in der **organisationsorientierten Entwicklung** nach (Wester-Ebbinghaus, 2010). Dies ist eine Weiterentwicklung des Agentenansatzes, der insbesondere die Organisationsstruktur auf die Anwendung abbildet – und umgekehrt. Microservices sehen ähnliche Synergieeffekte, sind allerdings nicht in dem Maße schachtelbar wie Organisationssysteme.

Im kommenden Unterabschnitt betrachten wir die vorgestellten Microservice-Eigenschaften mit Fokus auf Sicherheit.

2.3 Motivation für Microservices und Sicherheit

Aufgrund der zuvor genannten Eigenschaften lösen Microservices sowohl Herausforderungen als auch Chancen für Sicherheitsaspekte einer Architektur. Herausfordernd ist dabei das Problem, dass ein Informationssystem nur so sicher ist wie ihr schwächstes Glied; dies kann bei einer Vielzahl von kleinen Rechensystem schnell schwierig zu überblicken sein. Insbesondere die Heterogenität der Dienste kann schnell zu Ausfällen und individuellen Sicherheitslücken führen.

Auf der anderen Seite lassen sich Microservices aufgrund ihrer sehr guten Austauschbarkeit leichter warten. Für die jeweiligen Teams ist es wesentlich leichter selektiv Sicherheitslücken zu bekämpfen und neue Versionen eines Dienstes zu veröffentlichen ohne damit ein gesamtes System zu gefährden. Auch die Skalierbarkeit der Instanzen hat zur Folge das Dienstausfällen vorgebeugt werden kann. Insbesondere bei einer DoS-Attacke können z.B. durch das schnelle Beenden und Neustarten von Microservices Gegenmaßnahmen ergriffen werden.

Nachdem wir mit diesem Abschnitt das Microservice-Konzept vorgestellt haben, diskutiert der folgende Microservice-Architekturen und legt dabei seinen Schwerpunkt auf die Sicherheitseigenschaften dieser.

3 Microservice-Architekturen und Sicherheit

Ein Microservice-basiertes System ist ein verteiltes System, das in der Regel auf mehreren Maschinen läuft. Auf jeder dieser Maschine laufen ein oder mehrere Microservices. Dabei dürfen Services sich während des Ausführungszeit nicht gegenseitig beeinflussen. Viele Teile des Systems müssen kommunizieren, entweder es werden Nachrichten bzw. Tasks entgegen genommen oder Nachrichten verschickt (Newman, 2015). Ein breiter Angriffsvektor entsteht.

Dieses Kapitel beschreibt zunächst gängige Technologien zum Bau von Microservices und geht auf Sicherheit im Sinne von Isolation ein. Anschließend werden Möglichkeiten betrachtet, komplexe Systeme auf Microservice-Basis zu ermöglichen. Die dabei entstehenden architekturellen Herausforderungen sowie auch mögliche Sicherheitsprobleme werden hervorgehoben. Mögliche Ansätze zum Schutz eines Microservice-basierten Systems werden diskutiert.

3.1 Technologien zum Bau von Microservices

Ein Microservice sollte replizierbar (und somit skalierbar), sowie einfach zu deployen und sicher sein (Newman, 2015; Richardson, o. J.). Um Skalierbarkeit und einfaches Deployment zu erreichen, bietet es sich wiederum an Microservices in ähnlichen Ausführungsumgebungen laufen zu lassen. Das jedoch steht im Wiederspruch zu dem Konzept der Service-Unabhängigkeit und Autonomie.

3.1.1 Virtuelle Maschinen

Mit herkömmlichen Virtualisierungstechnologien, wie etwa VMWare lässt sich die gewünschte Abstraktion schaffen. Services werden als Images für die virtuelle Maschine (*VM*) spezifiziert und können somit repliziert und portiert werden. Auch das Deployment des Services ist dann einfacher, weil nur noch das Service Image in der entsprechenden virtuellen Maschine gestartet werden muss.

Wenn mehrere Microservices den gleichen Host oder die gleiche VM teilen, sind sie nicht unabhängig (Richardson, o. J.). Bei böswilliger Manipulation oder Infiltrierung eines Services könnten auch andere Services im gleichen System beeinflusst werden. Folgt man dem Konzept so wäre es korrekt und sicher(-er) einen Service pro Host oder VM zu deployen. Das resultiert allerdings in Resourcen-Overhead der benötigt wird, um die (möglicherweise vielen) virtuellen Maschinen bzw. Hosts zu betreiben (Newman, 2015). Abbildung 1 zeigt die Resourcen auf, die bei einem Setup mit virtuellen Maschinen benötigt werden und zeigt vergleichsweise den Resourcen-Verbrauch eines Container-basierten Setups.

Der Ansatz, ein Microservice-basiertes System mit virtuellen Maschinen zu realisieren bringt jedoch auch Sicherheitsvorteile mit sich. Sofern wirklich nur ein Service pro virtueller Maschine läuft, sind alle Services des Systems vollständig unabhängig voneinander. Somit ist das Gesamtystem weniger Anfällig für Ausfälle oder bösartige Manipulation

einzelner Services.

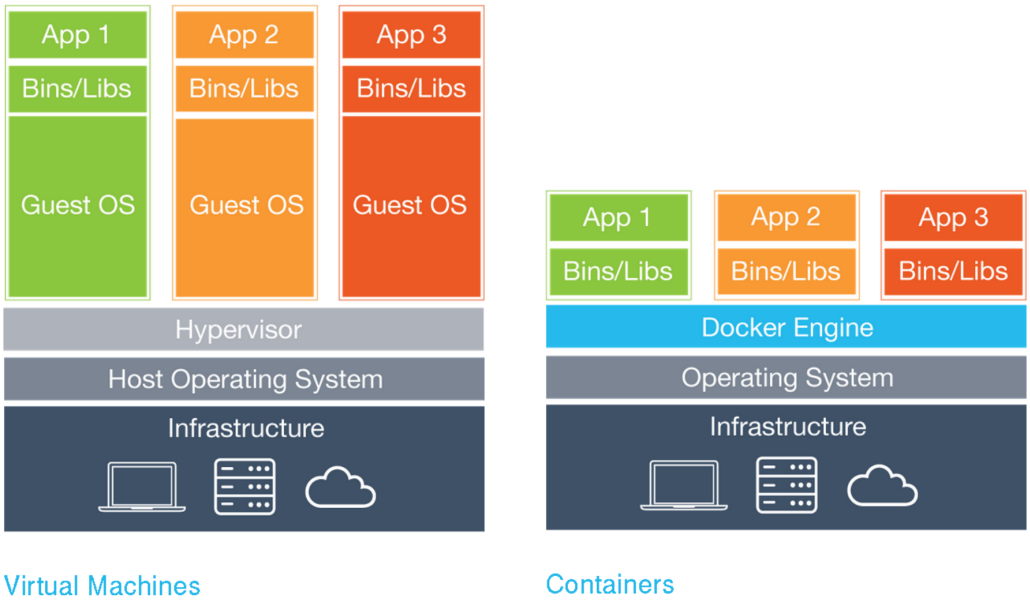

Abbildung 1: Vergleich von standard Virtualisierung und Docker Conainern, entnommen von http://www.docker.com/what-docker

3.1.2 Container

Ein Container ist ein schlanker „Wrapper“ um einen Linux Prozess. Wir betrachten hier *Docker*[3] als Container Engine, es sei aber noch LXC (Linux Containers) erwähnt.

Docker und Docker-Container bieten ähnliche Funktionen wie eine virtuelle Maschine und zugehörige VM Images. Allerdings sind Container um ein Vielfaches leichtgewichtiger als VMs. Während eine VM ein Gast-Betriebssystem und eigene Bibliotheken enthält, enthalten Container nur eine Anwendung und deren benötigte Abhängigkeiten. Container teilen sich den Kernel des ausführenden Host-Betriebssystems und laufen als einzelner Prozess im Userland (Newman, 2015). Die Grafik 1 zeigt deutlich, auf welche Weise Container leichtgewichtiger sind als virtuelle Maschinen.

Zusätzlich werden Container durch Container Images spezifiziert – genauso wie bei VMs – was die Vervielfältigung ungeheuer einfach macht. Allerdings ist die Modifikation von Dockerimages einfacher als das Ändern eines VM Images. Docker ermöglicht mithilfe eines sogenannten *Dockerfile*s sehr einfache Manipulation von Images. Docker bietet hervorragende Sicherheit im Sinne von Isolation. Durch die intelligente Verwendung von *Namespaces*, *Network Interfaces* und *Cgroups* werden in Containern laufende Prozesse vollständig von anderen Prozessen des ausführenden Systems isoliert (Newman, 2015; Marmol, Jnagal & Hockin, 2015). Laufende Docker Container haben einen eigenen Namespace. Für Prozesse innerhalb des Containers sind jegliche andere Prozesse nicht sichtbar und nicht beeinflussbar – ob auf dem ausführenden Host-Betriebssystem oder in anderen Containern. Jeder Container hat einen eigenen Network-Stack, wodurch jeder Container nur Einfluss auf seinen eigenen Traffic nehmen kann. Zuletzt wird der Resourcenverbrauch reguliert durch Control Groups, ebenfalls ein Linux Kernel Feature. Es wird garantiert,

[3] https://www.docker.com

dass ein Container genügend Resourcen bekommt, um zu arbeiten; es kann auch eingeschränkt werden, dass ein Container nicht zu viele Resourcen des Host-Betriebssystems verbraucht[4].

Dockers Charakteristika legen nahe, wie gut diese Technologie geeignet ist um die Anforderungen abzudecken, die eine Microservice-Architektur mit sich bringt. Durch das Design von Docker sind in Containern laufende Prozesse vollständig unabhängig voneinander. Services lassen sich in Form von Images einfach replizieren und portieren. Das ausführende System ist unwichtig, solange es ein Linux System ist und Docker installiert ist. Zum Thema Microservices, die mit Hilfe von Docker realisiert werden finden sich viele aktuelle Softwareprojekte.

3.2 Orchestrierung von Microservices

Es gibt verschiedene Möglichkeiten, Microservices miteinander interagieren zu lassen um daraus ein komplexeres System zu bilden. Generell gilt für Microservices, dass sie *klein*, auf *eine Aufgabe* spezialisiert und *autonom* sind (Newman, 2015). Das bedeutet insbesondere auch, dass Services eines Systems einander meist nicht kennen. Im Folgenden werden einige Herausforderungen diskutiert, die beim Bau eines Microservice-basierten Systems entstehen. Es werden Software Lösungen und Frameworks vorgestellt, die viele der Problemstellungen intelligent lösen. Sicherheitskritische Bereiche werden aufgezeigt und Maßnahmen zu deren Schutz erläutert. Anschließend wird das Open-Source-Projekt *Kubernetes* von Google vorgestellt.

3.2.1 Systemanforderungen

Bei einem System aus autonomen Microservices muss auch das Deployment der Services autonom sein. Services zu starten/stoppen, zu verändern oder zu verfielfätigen sollte vom Gesamtsystem ermöglicht werden, während andere Services davon nicht beeinflusst werden. Das Gesamtsystem muss den Ausfall einzelner Services verkraften. Für Entwickler oder Administratoren muss ersichtlich sein, welche Komponenten des Systems korrekt oder inkorrekt arbeiten (Newman, 2015). Aus diesen Anforderungen und aus der Spezifikation von Microservices (siehe Abschnitt 2) ergeben sich einige problematische Fragestellungen, die gelöst werden müssen.

Wenn Services autonom agieren und einzeln an- und abschaltbar sind, wie können sie dann miteinander kommunizieren? Microservices haben in der Regel keine feste IP Adresse, unter der sie immer erreichbar sind. Dieses Problem wird als *Service Discovery* bezeichnet. Services müssen einander im System finden, um Nachrichten austauschen zu können. Technisch lässt sich das Problem beispielsweise durch den Einsatz einer DNS (Domain-Name-System) lösen. Auch *etcd*[5] ließe sich dafür verwenden.

Wie lässt sich das Verhalten einzelner Services nachvollziehen? Dazu kann zentralisiertes Logging verwendet werden; alle Microservices schicken ihre Logs zu einem Service, der allein dafür zuständig ist, Lognachrichten zu akkumulieren. Ein sehr häufiges Setup ist der sogenannte *ELK-Stack*[6]. Dabei wird *Logstash* als zentralisierter Log-Akkumulator verwendet, die Logs werden anschließend in einer *Elasticsearch* gespeichert und mit *Kiba-*

[4]Detaillierte Informationen zu Docker Security: https://docs.docker.com/engine/articles/security/
[5]https://coreos.com/etcd/docs/latest/
[6]https://www.elastic.co/webinars/introduction-elk-stack

na visualisiert. Ebenfalls populär, insbesondere im Container Umfeld sind *InfluxDB*[7] und *Grafana*[8] womit sich Zeitbasiert Metriken gut visualisieren lassen.

Das Sichten von Lognachrichten erfordert in der Regel menschliche Interaktion und nur mit Logging lässt sich nicht gezielt abfragen, ob ein Service überhaupt arbeitet oder nicht. Daher werden *Healthchecks* für jeden einzelnen Microservice benötigt. Ein Healthcheck gibt Auskunft darüber, ob ein Service läuft und arbeitsfähig ist. Für jeden Microservice kann ein Healthcheck unterschiedlich sein, beispielsweise ein einfacher Ping für einen Rest-Service oder ein Select-Statement für eine Datenbank.

Wie lässt sich das System skalieren? Da die Microservices des Systems einfach replizierbar und portierbar sind, kann bei hoher Last auf einem Service einfach weitere Services hinzugefügt werden, die die selbe Aufgabe erfüllen. Man spricht von horizontaler Skalierung. Damit dies jedoch fehlerfrei funktionieren kann, muss Service Discovery (s.o) und auch Loadbalancing im System vorhanden sein.

Wenn mehrere Services die gleiche Aufgabe übernehmen, muss das Gesamtsystem entscheiden, welcher Service welche Aufgaben erhält. Das nennt man Loadbalancing. Gleiche Arbeit wird von replizierten Services übernommen, aber wie kann entschieden werden, welcher Service gerade bereit für neue Last ist? Hier können simple Verfahren wie etwa Round-Robin eingesetzt werden; häufig werden jedoch erprobte Systeme wie etwa *HAProxy*[9] oder *Nginx*[10] verwendet.

Zuletzt bleibt noch die Frage nach Sicherheit. Zum einen muss die Kommunikation zwischen den Services sicher sein, Nachrichten dürfen nicht manipuliert werden. Klassische Angriffsszenarien wie beipielsweise ein Man-In-The-Middle Angriff zwischen Services sollte nicht möglich sein. Zum anderen muss auch das gesamte System gesichert sein. Die Ausführung, das Deployment oder Balancing von Services darf für Unbefugte nicht manipulierbar sein.

Im Folgenden stellen wir kurz das Open-Source-Projekt *Kubernetes* vor. Es wird diskutiert, wie das System in seiner Gesamtheit vor unbefugtem Zugriff geschützt wird. Auf die „Servicekommunikation und Sicherheit“ gehen wir dann im darauf folgenden Kapitel detaillierter ein.

3.2.2 Kubernetes

Es gibt viel Bewegung im Feld um Microservices und Docker. Viele großangelegte Projekte, getrieben durch führende Software-Konzerne[11] nehmen maßgeblich Einfluss auf die Softwareentwicklung in diesem Gebiet. Kubernetes, eines dieser Projekte ist eine open-source Software, gestartet von Google und wird von einer wachsenden Community weiter vorangebracht.

> *„Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts.“(*Kubernetes Documentation*, o. J.)*

[7] https://influxdb.com/

[8] http://grafana.org/

[9] http://www.haproxy.org/

[10] http://nginx.org/

[11] Kubernetes//Google: http://kubernetes.io, Mesos//Apache: http://mesos.apache.org, Azure//Microsoft: http://azure.microsoft.com, ECS//Amazon: https://aws.amazon.com/de/ecs/

Kubernetes bietet eine Menge von grundlegenden Abstraktionen und Konzepten, die bei der Entwicklung von Microservice-Systemen hilfreich sind. Kubernetes lässt sich ebenfalls hervorragend dafür verwenden, *PaaS*-Systeme (Platform-as-a-Service) zu realisieren; diesen Anwendungsfall wollen wir hier nicht näher betrachten.

Kubernetes erstreckt sich als transparent verteiltes System über eine Gruppe von physikalischen Maschinen (*Nodes*). Sämtliche Resourcen, so wie etwa CPU und Memory werden transparent für das Gesamtsystem nutzbar. Die nun folgenden Begriffserklärungen stützen sich alle auf die Kubernetes Dokumentation, siehe (*Kubernetes Documentation*, o. J.).

Container werden in Kubernetes als Gruppen bestehend aus mindestens einem Container betrachtet; diese Abstraktion nennt sich *Pod*. Ein Pod abstrahiert einen logischen „Host“ für Container. Alle Container innerhalb eines Pods laufen auf der gleichen Node und teilen das gleiche Schicksal (*Kubernetes Documentation*, o. J.). Pods werden niemals bewegt. Wenn eine Neu-Lokation stattfinden muss, etwa weil eine Node aus dem Gesamtsystem ausscheidet, werden alle Pods (und darin lebende Container) die auf dieser Node liefen zerstört. Neue Pods mit Containern, die die gleiche Aufgabe übernehmen wie die zerstörten werden auf einer neuen, noch verfügbaren Node gestartet.

Pods können repliziert werden. In einem einfachen Beispiel enthält ein Pod genau einen Container, der eine einfache Aufgabe erledigt (etwa Requests entegennehmen). Sollte nun der eine Pod nicht mehr in der Lage sein, alle Requests in zufriedenstellender Zeit zu beantworten, so wird es nötig diesen Dienst zu beschleunigen. In Kubernetes kann dazu mithilfe eines *Replication Controller*s spezifiziert werden, wie viele Pods mit diesem Dienst gestartet werden sollen. So müssten z.B. zwei Dienste mit der gleichen Aufgabe dann exakt halb so viel arbeiten wie nur ein Dienst. Replication Controller bilden für den Microservice-Einsatz-Kontext das Konzept der horizontalen Skalierbarkeit ab. Replication Controller bieten darüber hinaus aber noch mehr. Sie garantieren, dass von einem Pod zu jeder Zeit exakt so viele Replica existieren, wie spezifiziert. Wenn es zu viele oder zu wenig gibt, werden neue gestartet oder überflüssige Pods terminiert. Dies geschieht völlig dynamisch. Bei einer Spezifikationsänderung werden nur so viele hinzugefügt bzw. entfernt wie nötig. Es wird garantiert, dass mindestens immer ein Dienst läuft. *„Think of it similarly to a process supervisor, only it supervises multiple pods across multiple nodes instead of individual processes on a single node“* (*Kubernetes Documentation*, o. J.).

Da Pods jeweils eine eigene IP im Kubernetes System haben, müssen sie entsprechend einzeln angesprochen werden. Das wird jedoch durch die Ersetzbarkeit und dynamische Erzeugung und Termination erschwert, weil neue Pods neue IPs bekommen. Wie findet man eine Gruppe von replizierten Pods und wie spricht man sie an? Dafür gibt es in Kubernetes das Konzept des *Service*. Ein Service in Kubernetes kapselt eine Gruppe von Pod-Replica und bietet ein festes Interface, um die Pod-Gruppe anzusprechen. Der Service übernimmt außerdem das Loadbalancing zwischen den einzelnen Pods.

Abbildung[12] 2 zeigt das Zusammenspiel von Pods, Services und Replication Controllern. Alles beginnt mit dem Container *„nginx-tester“*, der eine nginx beherbergt, die wiederum http-Requests entgegen nimmt. Dieser Container wird durch einen Pod gekapselt. Damit bietet der Pod, ansprechbar unter einer kurzlebigen IP diesen Dienst an. Der Pod wiederum wurde sechs mal repliziert, was durch den blau gefärbten Replication Control-

[12]Grafik erzeugt mit k8s-visualizer, https://github.com/0ortmann/k8s-visualizer

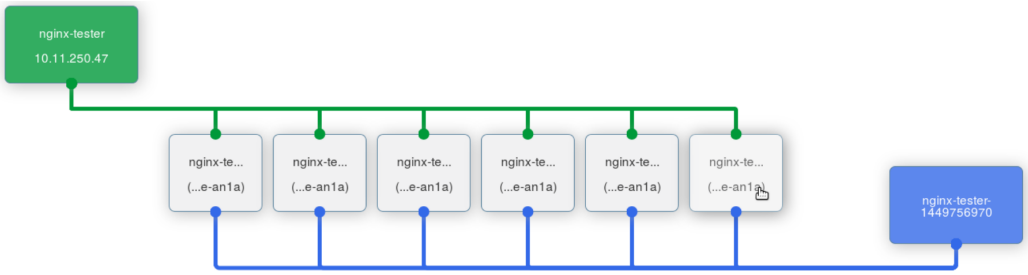

Abbildung 2: 6 replizierte Pods (weiß), 1 Service (grün), 1 Replication Controller (blau)

ler dargestellt wird. Alle sechs Pods bieten unter sechs verschiedenen IP Adressen nun den Dienst „nginx-tester“ an. Um diese Gruppe einheitlich ansprechen zu können, gibt es einen Service – dargestellt in grün. Der Service hat eine feste IP im Kubernetes System, sichtbar in der Grafik unterhalb des Service Namens in der grünen Box. Möchte man diesen Service ansprechen, so erreicht man ihn unter seiner festen IP. Der Service leitet den Request an einen der sechs möglichen „nginx-tester“ Pods weiter. Werden Pods gelöscht / erzeugt, so wird der umgebende Service dies transparent, also ohne dass es einem Nutzer auffallen würde, handhaben. Dies ist angedeutet in Abbildung 3. Der dortige Service wird ankommende Requests transparent auf einen der vier verfügbaren Pods (weiß) balancen, zwei Pods sind gerade nicht ansprechbar (gelb).

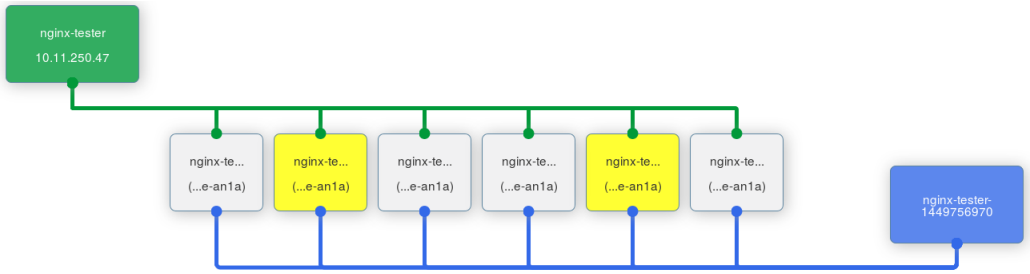

Abbildung 3: 6 replizierte Pods, davon 4 laufend (weiß) und zwei startend (gelb), 1 Service (grün), 1 Replication Controller (blau)

Durch den Einsatz eines Systems wie Kubernetes werden viele (nicht alle) sicherheitskritische Fragen für den Anwender gelöst. Kubernetes baut für die interne Resourcen-Koordination eigene Subnetze auf; es werden interne IP Adressen vergeben, die nach außen hin nicht existieren. Es ist unmöglich einen Man-in-the-middle Angriff zu Praktizieren, solange man außerhalb des Subnetzes ist. Erst wenn ein potenzieller Angreifer innerhalb des Gesamtsystems Zugriff auf die einzelnen Subnetze hat, ist Abhören oder Modifizieren von Traffic möglich.

Das böswillige Manipulieren von Anwendungen ist noch schwieriger und nur möglich, wenn der umgebende Container einer Anwendung manipuliert werden kann. Um einen Container zu manipulieren muss zunächst der entsprechende Pod lokalisiert und inflitriert werden; wenn ein Angreifer die Container-Ebene erreicht hat, müssen immer noch die Isolationsmechanismen von Docker überwunden werden.

Die Konfiguration und Manipulation von Kubernetes Resourcen ist geschützt durch Authentifikation via OAuth und SSH. Ferner ist es zu empfehlen, den Zugriff auf Schnittstellen zu Kubernetes durch eine Firewall zu schützen, sodass nur vertrauenswürdige Per-

sonen am System arbeiten dürfen. In einem Produktionssystem könnte man beispielsweise den Zugriff auf ein Kubernetes Cluster mit einer Firewall schützen, die nur die IP Adressen des Entwickler-Büros durchlässt. Jeder Entwickler wiederum muss sich per OAuth oder SSH authentifizieren.

Im folgenden Kapitel betrachten wir ein verallgemeinertes Microservice-System und heben die Spezifikation und die Schwachpunkte der Kommunikationswege zwischen Microservices in der Vordergrund.

4 Kommunikation und Sicherheit

Wie in Kapitel 3 beschrieben, besteht ein Microservice-basiertes System aus einer Menge von mindestens einem physikalischen Host und beliebig vielen Microservices. Um überhaupt von einem System sprechen zu können, müssen die einzelnen Komponenten noch vernetzt werden. Wir benutzen im Folgenden den Ausdurck *Service-to-Service-Kommunikation* – StS-Kommunikation. Services kommunizieren miteinander, tauschen Nachrichten aus, fordern Dinge an die von einem anderen Service berechnet werden müssen etc (Newman, 2015). Diese Kommunikation kann angegriffen werden, insbesondere wenn die Kommunikation über unsichere Kanäle stattfindet wie etwa das World-Wide-Web.

Im Folgenden wollen wir verschiedene Möglichkeiten von Sts-Kommunikation erläutern, Vor- und Nachteile diskutieren und auf Sicherheitsaspekte eingehen.

4.1 Kommunikationswege

Kommunikation zwischen Microservices findet fast immer event basiert statt. Die Eventverarbeitung kann synchron oder asynchron stattfinden (Newman, 2015). Dadurch, dass Microservices immer genau eine logische Aufgabe erfüllen, werden sie meist von extern dazu angestoßen etwas zu verarbeiten (*receive-event*). Wenn ein Microservice etwas verarbeitet hat, dann gibt er entweder das Ergebnis zurück oder verschickt eine Nachricht, dass nun ein Ergebnis vorliegt (*send-event*). „Applications built from microservices aim to be as decoupled and as cohesive as possible [...] receiving a request, applying logic as appropriate and producing a response“ (Fowler & Lewis, 2014). Wir werden dies an den zwei populärsten Verfahren zur Sts-Kommunikation erläutern: *Message Queues* und *REST* (Fowler & Lewis, 2014; Newman, 2015). Dabei werden Aufgabengebiete herausgestellt, für die die jeweilge Kommunikationsart im Mircoservice Kontext am geeignetsten erscheinen.

4.1.1 REST

REST ist die Kurzschreibweise für „Representational State Transfer“, eine zustandslose, protokollbasierte Frage-Antwort-Kommunikationsform. Darüber hinaus beschreibt REST umgangssprachlich auch einen Architektur-Ansatz, um verteilte Anwendungen zu entwerfen (Webber, Parastatidis & Robinson, 2010). Für detaillierte Informationen zum Thema REST empfehlen wir *REST in Practice, O'Reilly.*

Im Microservice Kontext lässt sich REST hervorragend für Sts-Kommunikation verwenden (Fowler & Lewis, 2014). Dabei wird in der Regel das HTTP Protokoll verwendet. Um über REST Schnittstellen kommunizieren zu können, müssen die jeweiligen Micro-

services in der Lage sein, einander anzusprechen. Es müssen also die IP-Adressen der einzelnen Komponenten bekannt sein, etwa durch Service-Discovery (siehe 3.2.1).

HTTP Kommunikation kann belauscht und unterbrochen werden, außerdem ist sie anfällig für Man-in-the-Middle Angriffe und vergleichbare Szenarien. Um ein wenig mehr Sicherheit zu schaffen, kann zur Sts-Kommunikation auch HTTP/s verwendet werden können. HTTP/s für sich allein bringt nach (Newman, 2015) nur einen geringen Vorteil. Zwar ist dann der Traffic verschlüsselt, aber in der Regel sind die APIs der Services frei ansprechbar. Daher wird zusätlich empfohlen, dass die einzelnen Microservices sich bei anderen authentizieren. Ein solider und einfacher Ansatz wäre etwa Basic-Auth über HTTP/s. Die Übetragung von Username und Passwort erfolgt verschlüsselt; es kann sichergestellt werden, dass nur berechtigte Services anderen Services Nachrichten schicken dürfen. Außerdem kann zu späteren Zeitpunkten anhand von Logs und Usernamen nachgewiesen werden, wann und zwischen welchen Services eine Kommunikation stattgefunden hat. Im folgenden Kapitel werden wir Verfahren zum Schutz und zur Authentizierung detaillierter behandeln.

REST eignet sich für Microservice-basierte Systeme insbesondere, wenn der Bedarf nach synchroner Kommunikation gegeben ist. Nachdem ein Service mit einem Request angesprochen wurde, erwartet der Anfragende eine entsprechende Antwort. Als Übertragungsformat kommen meist leichtgewichtige und ausdrucksstarke Text-Formate zum Einsatz wie JSON und XML. Von Vorteil ist auch, dass es eine breite Unterstüzung in Form von Frameworks und HTTP Clients gibt, sodass meist nur minimaler Overhead entsteht, um einer Software-Komponente zu ermöglichen via REST zu kommunizieren.

Da die Kommunikation via REST synchron ist, ergeben sich Probleme wenn ein Service neu gestartet, gelöscht oder ersetzt wird. Bestehende Verbindungen werden unterbrochen und erwartete Anworten bleiben aus. Der Anfrangende Service muss selbst eine Fehlerbehandlung implementieren, wenn benötigte Antworten nicht kommen.

4.1.2 Message Queues

Das Konzept einer Queue werden wir nicht näher erläutern. Einen guten Überblick liefert (Hohpe & Woolf, 2003). Kommunikation über Queues ist immer asynchron. Wenn etwas in einer Queue abgelegt wurde, verweilt es dort bis es wieder herausgenommen wird. Sender und Emfänger müssen einander nicht kennen. Es kann beliebig viele Sender und beliebig viele Empfänger geben. Queues lassen sich auch als Puffer verwenden.

Im Microservice Kontext lassen sich Queues insbesondere gut dafür verwenden, asynchrone, horizontal skalierbare Worker-Chains zu realisieren (Newman, 2015). Dabei wird eine große logische Verarbeitungsaufgabe in viele kleine Einzelschritte zerlegt. Ein Microservice bildet die Arbeitsoperation eines Teilschrittes ab. Dabei lauscht ein Microservice auf einer Queue, sobald etwas verfügbar ist wird es abgearbeitet und das Ergebnis auf eine andere Queue geschrieben. Wenn ein Verarbeitungsschritt absehbar mehr Zeit kostet als andere, so können einfach weitere Microservices gestartet werden, die an der gleichen Queue lauschen, sodass dort mehr Durchsatz geschaffen wird.

Ein gewaltiger Vorteil von Queueing ist, dass diese Kommunikationsarchitektur das Problem der Service-Discovery etwas abschwächt. Worker-Services müssen nur die entsprechende Queue kennen, auf der sie lauschen bzw. schreiben. Services müssen sich gegenseitig nicht kennen, um zu kommunizieren. So wird das Skalieren, Stoppen oder Ersetzen von Services völlig transparent für das Gesamtsystem.

Probelematisch ist allerdings hier die Integrität von Nachrichten. Services sollten sich auf ein dediziertes Format einigen und jede angenommene Nachricht zunächst validieren und verifizieren. Da Sender und Empfänger sich nicht kennen, ist es schwierig den Absender eindeutig festzustellen. Allerdings kann durch ein geteiltes Nachrichtenformat, etwa durch ein JSON-Schema oder Ähnliches sichergestellt werden, dass eingehende Nachrichten überhaupt verarbeitet werden können.

Queues implementiert man in der Regel nicht selbst, sondern greift auf bestehende Lösungen zurück. Es gibt zwei sehr aktuelle Message Queues, die für Microservice-Systeme verwendet werden. Das ist zum einen *RabbitMQ*[13], zum anderen *ZeroMQ*[14]. Beim Einsatz jedweder Lösung sollte man stets darauf achten, dass Nachrichten vor Modifikation durch unbefugte geschützt sind. So könnte etwa bei jeder Nachricht in der Queue noch ein entsprechender Hash mitgeschickt werden, der zusätzlich zum Nachrichtenformat zur Validierung der Nachricht verwendet wird.

Im Folgenden werden wir Schutzziele definieren und auf Microservice-Systeme sowie auf Kommunikation übertragen. Es werden einige konkrete Verfahren und Bibliotheken vorgestellt, mit denen sich diese Schutzziele umsetzen lassen.

5 Schutzziele

Nachdem wir bisher auf die Architektur und die Kommunikation von Microservices eingegangen sind, betrachten wir in diesem Abschnitt die Angriffspunkte einer solchen Infrastruktur. Explizit werden dazu Schutzziele der Informationssicherheit betrachtet. Diese sind ein guter Maßstab dafür, ob bestimmte Angriffe gegen ein Informationssystem möglich sind oder nicht. In dieser Arbeit werden wir für drei Ziele Schutzmechanismen vorschlagen, welche für Microservices in Betracht gezogen werden können. Die ausgewählten Schutzziele spielen dabei eine besondere Relevanz in verteilten Systemen.

Zunächst werden wir daher in Unterabschnitt 5.1 auf das Ziel der Vertraulichkeit eingehen. Weitergehend setzt sich Unterabschnitt 5.2 mit der Integrität und zuletzt Unterabschnitt 5.3 mit der Verfügbarkeit von Informationen innerhalb eines Microservice-Netzwerks auseinander.

5.1 Vertraulichkeit

Zur Einführung der drei Schutzziele ziehen wir die Definitionen von Bedner und Ackermann zurate, um auf diese näher eingehen zu können. Zunächst widmen wir uns der Vertraulichkeit.

„Informationsvertraulichkeit ist bei einem IT-System gewährleistet, wenn die darin enthaltenen Informationen nur Befugten zugänglich sind. Dies bedeutet, dass die sicherheitsrelevanten Elemente nur einem definierten Personenkreis bekannt werden. Dazu sind Maßnahmen zur Festlegung sowie zur Kontrolle zulässiger Informationsflüsse zwischen den Subjekten des Systems nötig (Zugriffsschutz und Zugriffsrechte), sodass ausgeschlossen werden kann, dass Informationen zu unautorisierten Subjekten ‚durchsickern'." (Bedner & Ackermann, 2010)

Für eine Microservice-Architektur betrifft dies vor allem die Kommunikation zwischen

[13] https://www.rabbitmq.com/
[14] http://zeromq.org/

den einzelnen Microservice-Instanzen. Um diese für beide Gesprächspartner abzusichern werden zwei verschiedene Verfahren vorgestellt, zum einen die Handshake-basierte Authentifikation, bei der sich beide Gesprächspartner einander vorstellen, und die Token-basierte Authentifikation, bei welcher ein Geheimnis, ein sogenanntes „Token“, die Inhalte der Kommunikation quittiert und belegt.

5.1.1 OAuth-basierte Autorisierung

Als erstes diskutieren wir ein Handshake-basiertes Autorisierungsverfahren. Im Webumfeld ist das populärste das bereits zuvor angesprochene OAuth-Framework. Es ist aufgenommen worden von der *Internet Engineering Task Force* (IETF). Die Spezifikation für die 2012 veröffentlichte Version 2 des OAuth-Frameworks ist in den *Request for Comments* (RFC) Nummer 6749 der IETF zu finden. Es ist insbesondere für *Single-Sign-On*-Dienste geeignet; also in den Fällen, bei denen man für verschiedene Dienste die gleichen Zugangsdaten verwenden möchte. (Hardt, 2012)

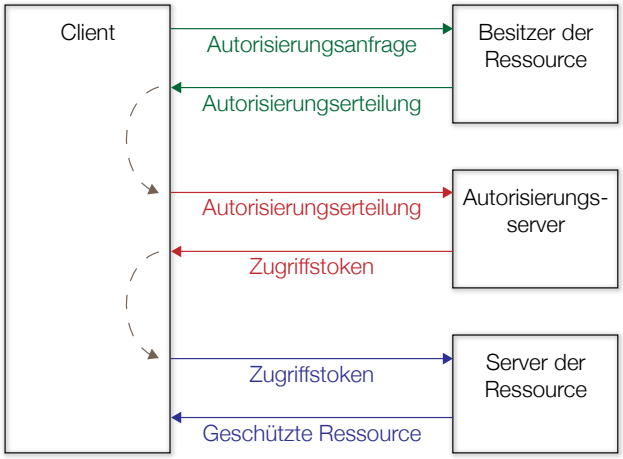

Abbildung 4: Protokollablauf bei OAuth 2 (übersetzt aus RFC 6749).

Zunächst wird aufgezeigt, wie eine Autorisierung mithilfe von OAuth 2 stattfinden kann, um auf Zugriff auf eine externe Ressource zu erhalten. Ein solcher Ablauf ist in Abbildung 4 dargestellt. Eine Applikation, die auf die Ressource zugreifen möchte, fungiert dabei als Client. Vorab muss sich dieser beim Besitzer der Ressource eine Autorisierung erteilen lassen. Der Besitzer ist eine Entität, welche über die Informationen der erfragten Ressourcen verfügen kann. Etwa ist dies ein Benutzer oder eine Organisation, deren Details abgefragt werden.

Erteilt der Besitzer die Autorisierung, kann der Client dessen Quittung nutzen um einen Autorisierungsserver zu kontaktieren. Dieser verwaltet die Zugriffe auf die von ihm angebotenen Ressourcen. Sobald der Client die Autorisierungserteilung vorgelegt hat, stellt der Server ein Zugriffstoken aus. Dieser hat dabei die Möglichkeit, die Gültigkeit des Tokens zeitlich zu begrenzen und regelmäßige Aktualisierungen anzufordern.

Abschließend kann der Client mithilfe eines Zugriffstokens nun auf die Ressourcen zugreifen, welche dem Besitzer gehören und dem Client freigegeben wurden. Dies können verschiedene Orte oder auch schreibende Zugriffe sein. Damit bietet OAuth für Microservices eine gute Möglichkeit, einen Autorisierungsservice anzubieten, über den Tokens für die anderen Microservices angefordert werden können. Im Folgenden stellen wir ein leichtgewichtiges Verfahren vor, welches ähnlich funktioniert.

5.1.2 JWT-basierte Authentifikation

Das zuvor gezeigte OAuth-Verfahren offenbart durch seinen Aufbau, dass wir bereits eine recht komplexe Infrastruktur zur Verfügung stellen müssen, um das Framework umzusetzen: drei Applikationen, welche über Formulare mit dem Benutzer interagieren und zusätzlich mit einer Datenbank in welcher die Benutzerdaten abgelegt werden müssen.

Wenn man daher eine Authentifizierung etwas minimaler aufbauen möchte, bieten sich daher *JSON Web Tokens* (JWT) an. Bei diesen können beliebige Informationen encodiert werden, die mithilfe eines Schlüssels zertifiziert werden, um dessen Ursprung zu bestätigen und die Unversehrtheit zu schützen. Diese Technologie hat es im Mai 2015 ebenfalls geschafft, Bestandteil der IETF-Spezifikationen zu werden und ist als RFC mit der Nummer 7519 zu finden. (Jones, Bradley & Sakimura, 2015)

Ein JWT besteht aus drei Informationen: Zunächst wird ein Header übertragen, welcher den verwendeten Algorithmus zur Kodierung der Signatur enthält und den Typen des Tokens expliziert; derzeit ist hierfür nur „JWT“ zulässig. Als nächstes folgen die Daten, welche übertragen werden sollen. Diese werden in Form eines JSON benutzt. Die Daten und der Header werden Base64-Kodiert und durch einen Punkt getrennt. Abschließend wird eine Signatur auf Basis dieses Datums berechnet. Dazu wird der im Header deklarierte Algorithmus, entweder *HMAC* oder *RSA*, benutzt. Die Signatur wird nach einem weiteren Punkt an das bisher berechnete Datum gehängt und bildet damit das vollständige Token. Ein Beispiel zeigt Abbildung 5.

Verwendet man zur Kodierung den *HMAC*-Algorithmus, so wird ein Geheimnis benutzt, welches nur dem JWT-generierenden Knoten bekannt ist. Bei *RSA* ist es hingegen möglich, ein Schlüsselpaar aus privatem und öffentlichem Schlüssel zu verwenden. Dies ist insbesondere für Microservice-Umgebungen interessant, denn so kann einem Microservice, der lediglich für das Anmelden zuständig ist, der private Schlüssel zum Erstellen des Tokens bekannt sein. Allerdings kann allen Microservices hingegen der öffentliche Schlüssel vorliegen – so kann jeder Microservice beim Eintreffen einer Nachricht die Gültigkeit des Tokens überprüfen und im Falle einer Fälschung die Weiterverarbeitung verweigern.

Das nächste Schutzziel, welchem sich dieser Aufsatz widmen wird, ist die Integrität der zu übertragenen Daten zu schützen. JWT kann, wie wir angegeben haben, auch dazu


```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0aG
VtYSI6Ik1pY3Jvc2VydmljZXMgdW5kIFNpY2hlcmhla
XQiLCJtb2R1bCI6IkVudHdpY2tsdW5nIHZlcnRlaWx0
ZXIgU3lzdGVtZSIsImt1cnNuciI6IjY0LTQwMCJ9.gb
0y7zl4-IIZrRdWD14UaSGPz34vXqAPecoyQVeOrntFx
PSkhryDHn2ts3Wn7XyHjli1HDdLZbqV5f1SzHNKZfz_
Ww8Tl2xzKLURpX9QxuI2PvFQZF0bQ7PeKqmZD7OU4eI
4y9L6srzedEjsDDISbaFXTaIJp_j1JtM9XTtEgi2Vy9
TAIZczCacnxuJgxd-Nk40k6KgnFPz1ETIu9G3JyyMxP
Enh1pw7EzIefJnRaaswC0K1evjQFK6dFvIlgXuevM3u
gZcI7lA2f1PeMud1fkZu3yaw27jeeLZRU1WiRhMV9EB
5elsYiscnpoVjuEk4C8T-p3oji43zf1IT7XvBiQ
```


Abbildung 5: Ein *JSON Web Token*. In Rot ist der Header angegeben, Violett gibt die übertragenen Daten an (Thema und Modul dieses Aufsatzes) und in Blau ist die in *RSA*-kodierte Signatur enthalten.

verwendet werden, würde allerdings so für größere Datenmengen zweckentfremdet werden.

5.2 Integrität

Neben der Gewissheit, den Sender einer Nachricht zurückverfolgen zu können, interessiert uns als empfangendes System im Folgenden, auch über deren Inhalt eine Aussage treffen zu können. Diese wird durch das Schutzziel der Integrität abgedeckt und wie folgt definiert.

„Integrität oder Unversehrtheit bedeutet zweierlei, nämlich die Vollständigkeit und Korrektheit der Daten (Datenintegrität) und die korrekte Funktionsweise des Systems (Systemintegrität). Vollständig bedeutet, dass alle Teile der Information verfügbar sind. Korrekt sind Daten, wenn sie den bezeichneten Sachverhalt unverfälscht wiedergeben. Die Integrität bedeutet, dass Daten im Laufe der Verarbeitung oder Übertragung mittels des Systems nicht beschädigt oder durch Nichtberechtigte unbefugt verändert werden können. Beschädigungs- oder Veränderungsmöglichkeiten sind das Ersetzen, Einfügen und Löschen von Daten oder Teilen davon.“ (Bedner & Ackermann, 2010)

Wir gehen zunächst auf ein Verfahren ein, um die Datenintegrität zu schützen und werden daraufhin Möglichkeiten zum Schutze der Systemintegrität in Microservices aufzeigen.

5.2.1 Datenintegrität

Ein besonderes Problem, welches durch die lose Kopplung der Microservice-Instanzen entsteht, ist, dass die Daten zur Weiterverarbeitung abhör- und manipulierbare Wege über standardisierte Protokolle wählen müssen. Wie bereits festgestellt, erfolgt die Kommunikation etwa über REST und damit über HTTP.

Eine Lösung, die sich hier anbietet, sind Verschlüsselungsverfahren. Das oben angesprochene Framework *Kubernetes* beispielsweise verteilt unter allen Knoten Zertifikate, die ausschließlich im Container-Netzwerk bekannt sind. So können HTTPS-Verbindungen zwischen den Microservice-Instanzen hergestellt werden, die dank AES-Verschlüsselung nicht mehr abgehört oder manipuliert werden können.

Weitere Protokolle, um komplexeren Anforderungen zu genügen, sind das verschlüsselte *WebSocket*-Protokoll WSS oder das sichere FTP, sprich SFTP. Eine Verbindung via *WebSockets* bietet ohnehin die Möglichkeit an, vor dem Senden der Daten eine Verschlüsselung zu installieren. Allerdings ist hier zu beachten, dass diese im Browser zurückverfolgt werden kann. Eine SFTP-Verbindung funktioniert über das SSH-Protokoll, welches verschiedene Möglichkeiten zur sicheren Kommunikation bietet; etwa auch wie bei JWT mit einem Schlüsselpaar aus öffentlichem und privatem Schlüssel.

5.2.2 Systemintegrität

Neben der Integrität der Daten ist es auch wichtig, die Integrität eines Microservice-Systems zu wahren. Ein Ansatz dafür ist zum Beispiel das automatisierte Testen. Wie dieses auf die lose Kopplung einer Microservice-Architektur angewendet werden kann, um die korrekte Arbeitsweise jener zu validieren, wird in dem Vortrag (Clemson, 2014) geschildert. In diesem wird gezeigt, wie Integrationstests genutzt werden können, um das Zusammenspiel mehrerer Microservices zu testen.

Bei einem Integrationstest werden bestimmte Teile eines gesamten Informationssystems als abgeschlossenes Subsystem getestet. So können Fehler oder Schwächen einer

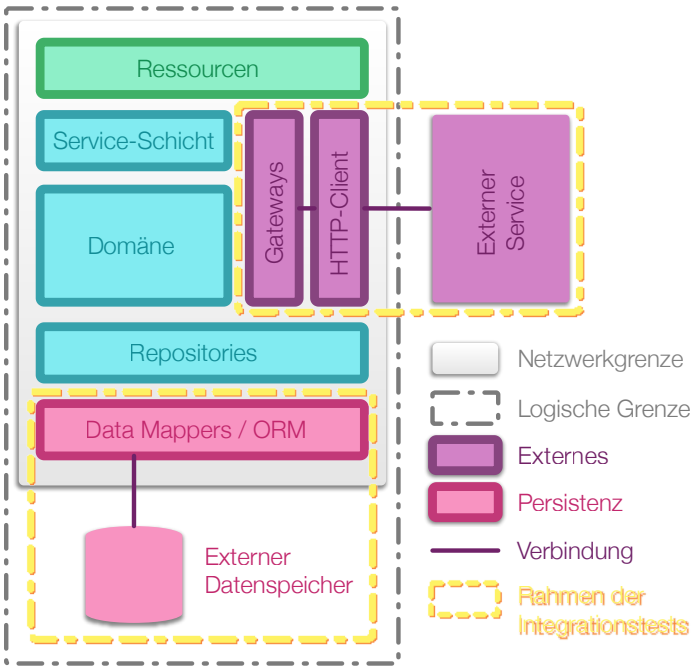

Abbildung 6: Rahmen der Integrationstests in einer Microservice-Architektur (übersetzt aus Clemson, 2014).

Schnittstelle entdeckt werden und es wird verhindert, dass bei zunehmender Änderung der einzelnen Komponenten bestimmte Abkommen nicht mehr eingehalten sind, was zu Laufzeitfehlern führen könnte.

Abbildung 6 zeigt eine exemplarische Microservice-Architektur auf. Hierbei sind die Bereiche hervorgehoben, bei denen die Integrationstests unseres Dienstes ansetzen sollen. Der untere Abschnitt zeigt **Persistent-Integrationstests**. Dabei wird getestet, ob ein Microservice korrekt mit einer Datenbank, einem Cache oder der Abstraktion einer solchen arbeitet. Weitergehend bildet der rechte Bereich **Gateway-Integrationstests** ab. Hier können Fehler bei der Umsetzung der verwendeten Protokolle, *Message Queues* oder auch in der JWT-Implementierung aufgedeckt werden. Außerdem wird festgestellt, ob ein Microservice die essentiellen Daten überträgt, die er vorgibt zu senden.

Mit der Integration verlassen wir den Bereich der softwaretechnisch lösbaren Sicherheitsprobleme und widmen uns im Folgendem dem letzten Schutzziel, welches wir betrachten werden, nämlich das der Verfügbarkeit eines Microservice-Systems.

5.3 Verfügbarkeit

Zuletzt widmen wir uns in dieser Arbeit der Verfügbarkeit als Schutzziel eines Microservice-Systems. Dabei ist die Maxime unseres Systems, nach Möglichkeit permanent erreichbar zu sein. Die folgende Definition erläutert den Verfügbarkeitsbegriff ausführlicher.

„Die Verfügbarkeit betrifft sowohl informationstechnische Systeme als auch die darin verarbeiteten Daten und bedeutet, dass die Systeme jederzeit betriebsbereit sind und auf die Daten wie vorgesehen zugegriffen werden kann. Zum einen muss die Datenverarbeitung inhaltlich korrekt sein und zum anderen müssen alle Informationen und Daten zeitgerecht zur Verfügung stehen und ordnungsgemäß verarbeitet werden.“ (Bedner & Ackermann, 2010)

Microservices sind, wie bereits einleitend erwähnt, sehr einfach horizontal zu skalieren. Dadurch haben wir die Möglichkeit, viele parallele Instanzen zu benutzen welche die gleiche Aufgabe erfüllen. Dies machen wir uns im zunächst genannten Lösungsansatz zunutze. Daraufhin erläutern wir einen neuartigen Ansatz, wodurch dieses Prinzip sogar noch weiter verbessert werden kann, um eine noch bessere Verfügbarkeit zu erzielen.

5.3.1 Redundanz und Lastverteilung

Durch die lose Kopplung und die harten Grenzen können Microservice-Instanzen ohne weitere Mühen vervielfältigt werden. Da die Kommunikation über Webtechnologie erfolgt, können Microservices auch über Systemgrenzen hinweg auf mehrere Rechnerknoten verteilt werden. Das heißt wir haben zunächst die Möglichkeit, einen Datenknoten mit allen Microservices durch Aufrüsten der Hardware zu skalieren und können anschließend unter vertretbarem Aufwand einen zweiten Knoten mit dem ersteren verbinden, um auf diesem weitere Instanzen auszuführen.

Haben wir mehrere Instanzen eines Microservices, benötigen wir einen Entscheider, welcher einen ankommenden Zugriff einer Instanz zuordnet. Einen solchen Entscheidungsvorgang nennen wir **Lastverteilung** (vom Englischen *Load Balancing*). Zuvor bereits wurde eine der populärsten Implementierungen dieser angesprochen, nämlich die Software *HAProxy*. Es gibt diverse Strategien, nach denen Lastverteilung ablaufen kann, und welche auch durch *HAProxy* unterstützt werden.

- Die einfachste Strategie ist eine, welche auf Basis des **Zufalls** entscheidet, welcher Instanz ein Auftrag zugeordnet wird. Allerdings wird dabei nicht berücksichtigt, wie groß die Aufträge sind, sodass in ungünstigen Fällen eine Instanz häufiger arbeitsintensive Aufträge erhält als andere.

- Als zweites kann auch **Sharding** in Betracht gezogen werden. Dafür teilen wir den ankommenden Datenbestand nach bestimmten Prinzipien auf möglichst gleichgroße Teilmengen auf. Etwa kann auf Basis des Alphabets (A-K, L-R, S-Z) oder eines Zahlenbereichs ein *Shard* bestimmt werden. Eine Microservice-Instanz ist dann für einen solchen *Shard* verantwortlich.

- Wir können die Last auch abhängig von der **Auslastung** eines jeweiligen Knoten verteilen. Dazu fragt die Lastverteilung die in Betracht gezogenen Knoten regelmäßig, wie viele Aufträge diese zur Zeit bearbeiten und weist ihnen dementsprechend weitere zu. Problematisch bei diesem Verfahren ist, dass es zusätzliche last erzeugt und technisch anspruchsvoller als die beiden zuvor erklärten Strategien ist.

Lastverteilung ermöglicht es der Architektur, mehrere Instanzen eines Microservices zu verwenden. Dabei ist zu erwähnen, dass wir einen neuen Flaschenhals erzeugen, da jegliche Microservice-Zugriffe zunächst der Lastverteilung anvertraut werden. Dies stellt einen sogenannten *„Single Point of Failure“* dar. Im Folgenden zeigen wir daher eine Verbesserung des oben genannten Prinzips auf.

5.3.2 Clientseitige Lastverteilung

In dem Blog-Beitrag (Li, 2015) wird ein neuartiger Ansatz motiviert, welcher die Verfügbarkeit von Lastverteilung verbessert. Dies ist notwendig, da, wie eben festgestellt, der

Last-verteilende *Load Balancer* einen Flaschenhals darstellt. Sollte dieser ausfallen, ist das System nicht mehr erreichbar.

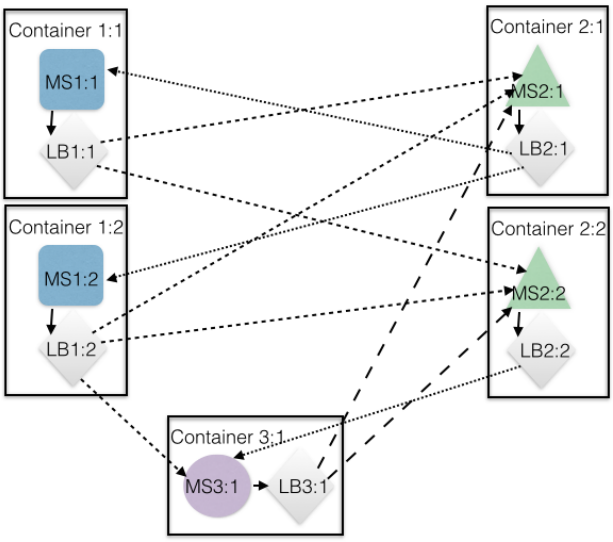

Abbildung 7: Schematische Darstellung von clientseitiger Lastverteilung (aus Li, 2015). Drei heterogene Microservices (MS) verfügen jeweils über eine Lastverteilung (LB), welche alle Instanzen anderer Microservices kennt.

Abhilfe für dieses Problem stellt der folgende clientseitige Lastverteilungsdienst dar. Diese funktioniert so, dass jeder Microservice, welcher auf die Dienste anderer angewiesen ist, über seine eigene Lastverteilung verfügt. Ein solcher Aufbau ist in Abbildung 7 schematisiert: Ein Microservice verfügt über seinen eigenen *Load Balancer*, an welchen er seine ausgehende Kommunikation exklusiv leitet. Dieser kennt alle anderen Microservice-Instanzen und kann daher Nachrichten an diese weiterleiten.

Die von Li entwickelte Grundstruktur *„Baker Street“*[15] ist eine Umsetzung dieser Architektur. Kern dieser sind die drei folgenden Komponenten.

- Der *HAProxy*-basierte *Load Balancer* **Sherlock** welcher lokal jeder Microservice-Instanz beiwohnt. Er findet heraus, wohin Verbindungen von dieser Instanz wandern sollen.
- Der *Health Checker* **Watson** über den ebenfalls jede lokale Instanz verfügt. Er übermittelt den Zustand des Microservices und meldet diesen ebenfalls initial an.
- Zuletzt gibt es das **Datawire-Verzeichnis**, ein leichtgewichtiger globaler Dienst-erkundungs-Mechanismus, welcher Informationen von jeder Watson-Instanz erhält und Verfügbarkeitsänderungen zu jeder lokalen Sherlock-Instanz weiterleitet.

Mit diesen drei Komponenten haben wir eine hohe Verfügbarkeit für die Grundstruktur unserer Microservice-Systems erzielt: Fällt ein Knoten mitsamt seines *Load Balancers* aus, weiß das *Datawire*-Verzeichnis Bescheid und die Anwendung bleibt verfügbar, da es keinen zentralen Knoten mehr gibt, der nicht antworten könnte.

[15] *Baker Street* ist unter Apache-Lizenz 2 veröffentlichte freie Software, siehe http://bakerstreet.io/.

Damit haben wir für alle drei Schutzziele der Vertraulichkeit, der Integrität und der Verfügbarkeit ausreichende Lösungsmöglichkeiten aufgezählt und schließen im folgenden Abschnitt die Ausarbeitung mit einer Zusammenfassung der Ergebnisse.

6 Zusammenfassung

Dieser Aufsatz gibt einen groben Überblick über das Modell der Microservices und zeigt deren Perspektiven in Puncto Sicherheit auf. Dabei motivieren wir dieses Modell zunächst über die Heterogenität von WebServices und die Möglichkeiten die es in Bezug auf Softwareverteilung eröffnet.

Danach geht der zweite Abschnitt auf die Eigenschaften von Microservice-Systemen ein. Es werden eingehend die Prinzipien der Kohäsion, der Autonomie und der Kooperation besprochen und deren Auswirkungen auf die Entwicklung diskutiert. Außerdem grenzen wir Microservices von anderen Modellen, wie etwa serviceorientierten Architekturen oder Multiagentensystemen ab. Schließlich motivieren wir insbesondere die Herausforderungen und Chancen die sich durch Microservices für Systemsicherheit ergeben.

Mit Abschnitt 3 werden Technologien präsentiert, um Microservices umzusetzen und zu kapseln. Dabei werden virtuelle Maschinen und Container vorgestellt und verglichen. Beide Konzepte bieten Mechanismen zur Isolation von Anwendungen. Weiterhin wird eine Anforderungsanalyse an ein Microservice-basiertes System aufgestellt. Aufgrund der Aktualität wird Kubernetes als mögliche Abstraktion eines solchen Systems ausgewählt und näher erläutert. Konzepte zur Replizierung, Skalierung und Balancing von und zwischen Containern werden diskutiert. Sicherheitsfragen werden formuliert und beantwortet. Dabei liegt der Fokus im Speziellen auf dem Schutz des Gesamtsystems.

Im darauffolgenden Abschnitt zeigen wir verschiedene Möglichkeiten zur Sts-Kommunikation auf. Dabei werden insbesondere REST und Message Queues betrachtet. Mögliche Schwachstellen bei der Sts-Kommunikation werden herausgestellt; der Bedarf nach applikationsseitig implementiertem Schutz wird deutlich gemacht. Aufgrund der Heterogenität eines Microservice-basierten Systems muss jede Komponente selbst in der Lage sein, ankommende Nachrichten zu validieren und zu verifizieren.

Weitergehend zeigt Abschnitt 5 drei explizite Schutzziele auf: die Informationsvertraulichkeit, die Integrität und die Verfügbarkeit eines Informationssystems. Für Vertraulichkeit zeigen wir *OAuth* und *JWT* als Möglichkeiten zur Authentifizierung auf. Die Integrität unterscheiden wir in Datenintegrität und Systemintegrität und gehen jeweils auf Lösungsansätze dazu ein. Bei der Verfügbarkeit nutzen wir die Skalierbarkeit der Microservices aus und motivieren zwei Ansätze zur Lastverteilung.

Addendum

Unter https://github.com/0ortmann/EvS-MS-Demo bieten wir eine Referenzimplementation eines Microservice Systems zu illustrativen Zwecken. Es werden vier unterschiedliche Microservices verwendet, die miteinander interagieren. Drei der Services sind selbst implementiert. Ein autonomes Frontend, realisiert mit *React*[16], ein Bildverarbeitungs-Backend in Python und ein Service der das Speichern von Bildern und Anmelden von Benutzern

[16] https://facebook.github.io/react/

handhabt, geschrieben in PHP. Als Speicherlösung – die als vierter Microservice verstanden wird – verwenden wir eine *MongoDB*[17]. Unsere Referenzimplementation verwendet unter anderem *JSON Web Tokens*. Dockerfiles stehen zur Verfügung, um die einzelnen Services in Container zu kapseln.

Im Zusammenspiel bieten all diese Microservices ein System, mit dem es einem Nutzer möglich ist, über eine Webpage im Browser PNG-Bilder hochzuladen und diverse Algorithmen zur Kantenerkennung darauf anzuwenden. Das anschließende Ergebnis wird ebenfalls in Bildform in der Datenbank gespeichert und der Nutzer erhält die Möglichkeit, es im Browser anzuschauen oder herunterzuladen.

Das Beispiel macht deutlich, dass nur im Zusammenspiel aus all den Services ein sinnvolles Gesamtsystem entsteht. Weiterhin wird klar, wie eine Authentifizierung via JWT umgesetzt werden kann. Dieses Beispiel ist kein *Boilerplate*, sondern eine alleinstehende Referenzimplementation im Zuge dieser Ausarbeitung.

Literatur

Auth0 Inc. (o. J.). *Introduction to JSON Web Tokens.* Webartikel. Zugriff am 2015-12-30 auf http://jwt.io/introduction/

Bedner, M. & Ackermann, T. (2010). Schutzziele der IT-Sicherheit. *Datenschutz und Datensicherheit*, *5*, 323 - 328.

Clemson, T. (2014, November). *Testing Strategies in a Microservice Architecture.* Präsentation. Zugriff am 2015-12-31 auf http://martinfowler.com/articles/microservice-testing/

Fowler, M. & Lewis, J. (2014, März). Microservices. A definition of this new architectural term.

Hardt, D. (2012, Oktober). *The OAuth 2.0 Authorization Framework* (RFC Nr. 6749). Internet Engineering Task Force. Internet Requests for Comments. Zugriff am 2015-12-29 auf https://tools.ietf.org/html/rfc6749 doi: http://dx.doi.org/10.17487/RFC6749

Hohpe, G. & Woolf, B. (2003). *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Jones, M., Bradley, J. & Sakimura, N. (2015, Mai). *JSON Web Token (JWT)* (RFC Nr. 7519). Internet Engineering Task Force. Internet Requests for Comments. Zugriff am 2015-12-30 auf https://tools.ietf.org/html/rfc7519 doi: http://dx.doi.org/10.17487/RFC7519

Kubernetes documentation. (o. J.). http://kubernetes.io/v1.1/docs/.

Li, R. (2015, Oktober). Baker Street: Avoiding Bottlenecks with a Client-Side Load Balancer for Microservices. Zugriff am 2015-12-08 auf http://thenewstack.io/baker-street-avoiding-bottlenecks-with-a-client-side-load-balancer-for-microservices/

Marmol, V., Jnagal, R. & Hockin, T. (2015, Februar). Networking in containers and container clusters. Netdev 0.1, Ottawa, Canada.

Meier, Homer, Hill, Taylor, Bascode, Wall, … Bogawat (2008). *Chapter 5. Deployment*

[17] https://www.mongodb.org/

Patterns. Zugriff am 2014-12-07 at 19:15 auf https://apparchguide.codeplex.com/wikipage?title=Chapter%205%20-%20Deployment%20Patterns

Newman, S. (2015). *Building microservices.* O'Reilly Media, Incorporated.

Richardson, C. (o. J.). *A pattern laguage for microservices.* http://microservices.io/patterns/.

Webber, J., Parastatidis, S. & Robinson, I. (2010). *Rest in practice: Hypermedia and systems architecture.* O'Reilly Media.

Wester-Ebbinghaus, M. (2010). *Von Multiagentensystemen zu Multiorganisationssystemen* (Unveröffentlichte Dissertation). Universität Hamburg.