2018 IEEE Symposium on Service-Oriented System Engineering

Overcoming Security Challenges in Microservice Architectures

Tetiana Yarygina
Department of Informatics
University of Bergen, Norway
Email: tetiana.yarygina@uib.no

Anya Helene Bagge
Department of Informatics
University of Bergen, Norway
Email: anya.bagge@uib.no

***Abstract*—The microservice architectural style is an emerging trend in software engineering that allows building highly scalable and flexible systems. However, current state of the art provides only limited insight into the particular security concerns of microservice system.**

With this paper, we seek to unravel some of the mysteries surrounding microservice security by: providing a taxonomy of microservices security; assessing the security implications of the microservice architecture; and surveying related contemporary solutions, among others Docker Swarm and Netflix security decisions. We offer two important insights. On one hand, microservice security is a multi-faceted problem that requires a layered security solution that is not available out of the box at the moment. On the other hand, if these security challenges are solved, microservice architectures can improve security; their inherent properties of loose coupling, isolation, diversity, and fail fast all contribute to the increased robustness of a system.

To address the lack of security guidelines this paper describes the design and implementation of a simple security framework for microservices that can be leveraged by practitioners. Proof-of-concept evaluation results show that the performance overhead of the security mechanisms is around 11%.

***Index Terms*—Microservices, defense-in-depth, SOA, distributed systems, cloud, PKI, authentication, MTLS, REST, JWT.**

I. Introduction

Microservices is an architectural style inspired by Service-Oriented Architecture in combination with the old Unix principle of "do one thing and do it well". Microservices are intended to be lightweight, flexible and easy to get started with, fitting in with modern software engineering trends such as Agile development, Domain Driven Design (DDD), cloud, containerization, and virtualization. The most frequently listed benefits of microservice architecture [1], [2] are organizational alignment, faster and more frequent releases of software, independent scaling of components, and overall faster technology adoption. The disadvantages include the need for making multiple design choices, the difficulty of testing and monitoring, and operational overhead when compared to typical non-distributed solutions.

The modern use of the term *microservice* dates back to 2011 [3], but the community has so far not reached a full consensus on a formal definition of the style. The fundamental basis of microservices brings together design principles from distributed systems and services, and from classic programming principles of abstraction, modularity, separation of concerns and component-oriented design. Although the underlying principles are widely explored in the academic literature, research on microservices themselves is lagging behind the rapid adoption and development in the software industry.

Understanding the distinctiveness of microservices is crucial. In particular, microservices bring new security challenges, and opportunities, that were not present in traditional monolithic applications. These challenges include establishing trust between individual microservices and distributed secret management; concerns that are of much less interest in traditional web services, or in highly modular software that only runs locally. Effective microservice security solutions need to be scalable, lightweight and easy to automate, in order to fit in with the overall approach and be adopted by industry users. For instance, manual security provisioning of hundreds or thousands of service instances is infeasible. As services are migrated from offline applications and monolithic web services to a microservice-style architecture, code that was never designed to be accessible from outside is now exposed through Web APIs, raising multiple major security concerns.

The past years have seen rapid adoption of microservices in the industry, and yet there has been surprisingly little focus on security. There is a growing body of literature [4], [5] that recognizes the need for microservice security evaluation. Security is one of the greatest challenges when building real-world systems, hence there is an urgent need to address the security concerns in microservice architecture.

Researchers have not treated microservice security in much detail; Fetzer [6] discusses how microservices can be used to build critical systems if the execution is warranted by secure containers and compiler extensions. Otterstad & Yarygina [7] suggests a combination of isolatable microservices and software diversity as a mitigation technique against low-level exploitation. Sun [8] explored the use of an Intrusion Detection System (IDS) for fine-grained virtual network monitoring of a cloud infrastructure based on Software Defined Network (SDN). While being presented as a solution for microservices, in reality it does not exploit any microservice-specific features.

Although the studies by Fetzer [6] and Otterstad & Yarygina [7] highlight the isolation benefits of microservice design, a systematic understanding of how the architecture affects security is still lacking. Moreover, microservice security is an overloaded term that requires proper clarification, and there is

0-7695-6394-5/18/$31.00 ©2018 IEEE
DOI 10.1109/SOSE.2018.00011

no overview of emerging industry security practices either.

This paper identifies and unravels some of the mysteries surrounding microservice security, and is the first study to undertake a holistic approach to microservice security. In summary, we make five contributions to the understanding of microservice security:

- putting microservices and their security in the bigger context of SOA and distributed systems (Section II);
- decomposing the notion of microservice security into smaller and more familiar components, with a formalized attack model (Section III);
- analyzing the security implications of microservice design (Section IV);
- identifying several prominent microservice security trends in industry (Section V);
- presenting an open source prototype security framework for microservices (Section VI).

II. What Microservices Really Are

Before we proceed with the security challenges for microservices, it is important to understand how microservices relate to other software architectural styles as well as to agree on the definitions.

A. Defining Microservices

Numerous terms are used to describe microservices, the most common microservice definitions are presented below:

- Newman [1] defines microservices as small autonomous services build around the following principles: *model [services] around business concepts*, *adopt a culture of automation*, *hide internal implementation details*, *decentralize all things*, *isolate failure*, and make services *independently deployable* and *highly observable*.
- Lewis and Fowler [3] view microservices as *"an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."*

The term has also been used in unrelated ways; for example, Kim [9] uses the term "microservice" when referring to the basic security primitives in context of formal methods: authentication microservice, integrity microservice, among others. This definition should not be confused with the modern definition of microservice architecture.

B. Defining Service-Oriented Architecture (SOA)

SOA can be seen as a predecessor to microservices; Josuttis [10] gives the following definition: "SOA is an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners".

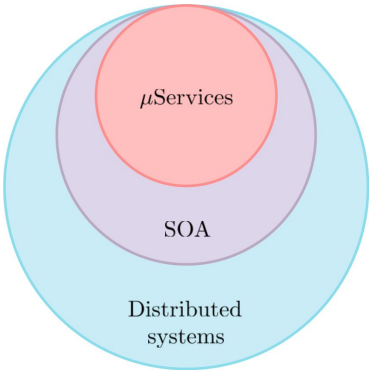

Fig. 1. Microservice architecture in perspective. Microservices are an implementation approach to SOA. SOA is a subclass of distributed systems.

The core technical concepts of SOA are:

- *Services.* Service in the context of SOA is *"an IT realization of some self-contained business functionality"* [10]. Based on multiple existing definitions of a service, additional situation-dependent attributes that services may have are: self-contained, coarse-grained, visible/discoverable, stateless, idempotent, reusable, composable, vendor-diverse.
- *Interoperability.* SOA interoperability in most cases is achieved via the Enterprise Service Bus that enables service consumers to call the service providers.
- *Loose coupling.* Loose coupling is needed to fulfill the goals of flexibility, scalability, and fault tolerance. Loose coupling is a principle that aims to minimize dependencies so the change in one service does not require a change of another service.

An important aspect of SOA, as stated in the SOA Manifesto, is that *"SOA can be realized through a variety of technologies and standards"*.

C. Déjà Vu

Majority of opinions on microservices fall into one of the two following categories: (1) microservices are a separate architectural style [3], [11]; (2) microservices are SOA [1], [12]. Some sources [4], [13] take a middle ground by viewing microservices as a refined SOA. The frequently mentioned differences are the degree of centralization, size of services, decomposition based on bounded context, and level of independence. However, these criteria are too vague to be used in a rigorous scientific comparison. They are neither easily quantifiable—it is unclear how the service size should be measured, nor sufficient to serve as style constraints.

Based on the above-listed definitions of SOA and microservices it is clear that these two approaches are similar. The definitions of SOA services and microservices are almost identical. This conclusion is important because it means that the security issues faced by practitioners who were migrating to SOA are now relevant for practitioners migrating to microservices. Figure 1 depicts this relation.

Zimmermann [12] is the first to systematically compare microservices and SOA, by surveying the authoritative technical blog posts and online articles on microservice principles. After contrasting the distilled microservice principles and SOA characteristics, he concludes that "the differences between microservices and previous attempts to service-oriented computing do not concern the architectural style as such (i.e., its design intent/constraints and its platform-independent principles and patterns), but its concrete realization (e.g., development/deployment paradigms and technologies)" [12].

Hence, we may expect many of the concerns and solutions that apply to SOA to also be applicable to microservices.

D. Distributed Systems

When talking about microservices many sources repeat the same fallacy of contrasting microservices to monolithic applications. The real situation is not as binary. In practice, a "monolithic" application may be highly modular internally, being built from a large number of components and libraries that may have been supplied by different vendors, and some components (such as a database) may be also distributed across the network. The issues of decomposition, concerns separation, and designing and specifying APIs will be similar regardless of whether API calls are made locally or across the network.

The essence of microservices is that they are (or compose to form) highly modular, distributed systems, reusable through a network-exposed API. This implies that microservices inherit advantages and disadvantages of both distributed systems and web services.

While distributed systems bring multiple highly desirable benefits such as scalability on demand and embrace of heterogeneity, these systems also come with drawbacks. Distributed systems are more challenging to develop in the first place. Moreover, monitoring, testing, and debugging such systems is more difficult than for non-distributed systems. Other well-known problems include maintaining data consistency and replication among nodes, node naming and discovery due to constantly changing network topology, and dealing with unreliable networks.

Important distinguishing characteristics of distributed systems over monolithic systems can be summarized as follows (adapted from Tanenbaum [14]): 1) The overall system state is unknown to individual nodes; 2) Individual nodes make decisions based on the locally available information; 3) Failure of one node should not affect other nodes. The microservice style enforces these requirements.

E. Microservices in Context of Other Technologies

A technology closely related to microservice philosophy is unikernels [15]. Unikernels are fixed-purpose OS kernels which run directly on a hypervisor. A full system would consist of multiple unikernels performing different tasks in a distributed fashion. The microservice design principles of small independent loosely coupled single-purpose components fit well into the unikernels world. Unikernels promise to provide high security and strong isolation of virtual machines while still being lightweight like containers.

Some programming languages, such as Erlang and its successor Elixir, support the distributed computing model by design. Erlang is an actor-based programming language. In the Erlang world "everything is a process" and the only process interaction method is through message passing. Erlang language features encapsulation/process isolation, concurrency, fault detection, location transparency, and dynamic code upgrade [16, p.29]. Erlang has advanced monitoring tools, and it is used for several high-load systems such as the messaging service WhatsApp. Systems built with Erlang are inherently microservice-like. As the aforementioned examples show, the core microservice principles can be rediscovered in many technologies.

F. Summary: Essence of Microservices

Compared to other software and service architectures, these are the distinctive characteristics of microservices:

- *Distributed composition:* Microservices build on other microservices and communicate across the network. (Compared to monolithic services.)
- *Modularity:* Microservices will tend to offer finer-grained APIs that lend themselves to flexible reuse. (Compared to big, single-purpose APIs/applications.)
- *Encapsulation:* Services are to a large degree encapsulated and isolated from others, and may even be written in different programming languages. (Compared to libraries and object-oriented encapsulation.)
- *Network service:* Services are network-accessible, and reuse happens by contacting the service rather than by installing and linking to a library. (Compared to a classic modular design.)

III. Layered Security for Microservices

Building secure systems is hard. The classic security objectives are data confidentiality and integrity, entity and message authentication, authorization, and system availability. Naturally, these abstract objectives can be realized in many different ways in practice. How these objectives are met and to what degree are the system-specific architectural decisions that depend on the particular threat model, budget, and expertise. An important issue is where in the system to place security mechanisms.

We argue that microservice security is a multifaceted problem and it relies heavily on underlying technologies and the environment. To get to the bottom of it, we need to decompose microservice security into its components.

A. Taxonomy of Microservice Security

To illustrate the underlying complexity of the problem, we divided microservice security concerns into six categories or layers as shown in Table I: hardware, virtualization, cloud, communication, service, and orchestration. We do not claim the proposed layering is complete, but we do claim that it provides a good overview of the topic.

TABLE I
PROPOSED HIERARCHICAL DECOMPOSITION OF MICROSERVICE SECURITY ISSUES INTO LAYERS

Layers	Threat examples	Mitigation examples
Hardware	Hidden under abstraction, but still a reliability and security concern; hardware bugs are extremely dangerous because they undermine security mechanisms of other layers [17]; hardware backdoors can be introduced at manufacturing time [18].	Designing own hardware; diversification of hardware [18]; use of Hardware Security Modules (HMS).
Virtualization	Deployment affects security; OS processes offer little separation from other services in the same system; containers and VMs offer more protection against compromised services. Attacks include: Sandbox escape, hypervisor compromise, and shared memory attacks; also, use of malicious and/or vulnerable images is another serious security concern [19].	Preferring deployment options with stronger isolation; secure configurations such as no shared library access and no shared hardware cache; verification of image origin and integrity; timely software updates; principle of least privilege.
Cloud	Cloud computing brings a myriad of security concerns [20], including unlimited control of cloud provider over everything it runs; there are few technical options to prevent disruption or attacks from a malicious provider.	Reverse sandboxes protects enclaves of code and data from any remote attack including attacks from OS and hypervisor. SGX (Intel), SME/SEV.
Communication	Classic attacks on the network stack and protocols; attacks against protocols specific to the service integration style (SOAP, RESTful Web Services [21], [22]). Attacks include: eavesdropping (sniffing), identity spoofing, session hijacking, Denial of Service (DoS), and Man-in-the-Middle (MITM); also attacks on TLS: Heartbleed [23] and POODLE [24].	Use of standard and verified security protocols such as TLS or JSON security standards. Security aspects of the chosen service integration style should be considered. Trivial mitigation like not re-using credentials across services.
Service/ Application	Typical and still very common application-level security problems are SQL injection flaws, broken authentication and access control, sensitive data exposure, Cross-Site Scripting (XSS), insecure deserialization, general security misconfiguration. The ten most critical web application security risks are published annually by OWASP [25].	Static/dynamic code analysis, manual code review. Basic software engineering practice: input validation, error handling, clear and well-documented APIs. Protection of the data at rest (encryption). Avoid languages especially vulnerable to, e.g., buffer and integer overflows.
Orchestration	Management, coordination, and automation of service related tasks, including scheduling and clustering of services. Microservice network structure may change continuously due to services being stopped, started, and moved around; service discovery [26], [27] provides a DNS-like central point for locating services. Attacks include: compromising discovery service and registering malicious nodes within the system, redirecting communication to them.	Protection of orchestration platform and its components is critical, but not well-investigated area. A secure implementation of service discovery and registry components is important.

This decomposition is based on the basic computer science principles and common sense. While it is inspired by the OSI model and its seven layers of communication, the proposed hierarchical decomposition also incorporates new trends in software engineering such as virtualization and orchestration. This decomposition is applicable to any modern distributed system and is not specific to microservices or SOA. However, it is crucial for the discussion of microservice security.

The decomposition shows that multiple security choices should be made on each level. There are many places where security components reside, and, more importantly, where security can fail. The system should not be treated as a black box. Some levels bleed into each other, e.g. virtualization is a main enabling technology for cloud computing. The separation of security concerns is not always strict.

The two bottom-most layers, hardware and virtualization, are at least partially accessible to an attacker with shell access on the host or virtual machines/containers correspondingly. A malicious hardware manufacturer that provides hardware with backdoors is a threat. On the Cloud level, a cloud vendor itself is a threat. Other tenants are also a potential threat in the cloud using various side-channel attacks, such as the FLUSH+RELOAD technique [28], or Meltdown and Spectre [29]. For communication and orchestration levels, a network attacker inside the perimeter who can eavesdrop and manipulate traffic is a major concern. For the service/application level, the threat of an external attacker should be considered.

The price of addressing the threats on different levels varies. For example, hardware concerns are extremely difficult to address, if at all practically possible. Most developers would be concerned with service/application and communication layers because addressing the security concerns on these levels is cost-feasible for them.

B. Attack Model: Redefining Perimeter Security

Until recently perimeter defense was the most common approach to security of microservice-based systems [1][p.173]. From the modern security perspective, perimeter security is in general considered insufficient—we should rather assume that the other services in the system may be compromised and hostile ("trust no one"). The rise of microservices, as well as advances in security automation, facilitate placement of additional security mechanisms inside the perimeter (see Figure 2). In other words, defense in depth as a concept of placing multiple and sometimes overlapping security mechanisms on different levels throughout the system becomes more feasible with microservices.

We assume an adversary is able to compromise at least one service inside the perimeter and wants to move laterally through the system to gain full control. If internal services blindly trust whoever is calling them, then a single compromised microservice will allow an attacker to manipulate all the other nodes in the microservice network, for example by issuing arbitrary malicious requests that the nodes will

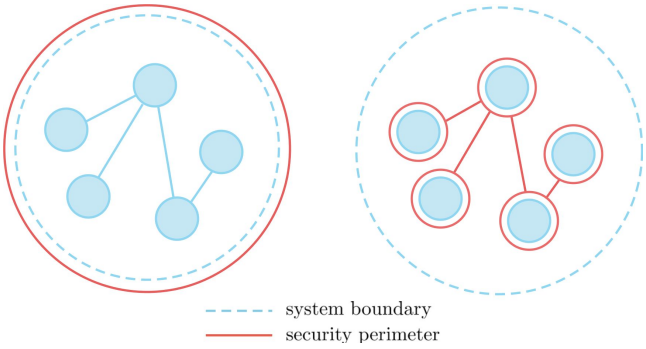

Fig. 2. Microservices redefine the notion of perimeter security and move towards defense in depth. The figure excludes infrastructure services, such as API Gateway or monitoring service

fulfill. The latter is sometimes referred to as a confused deputy problem [1]. The adversary can attempt to eavesdrop on the inter-service communication, insert and modify data in transit. We also make a standard cryptographic assumption that the adversary is computationally bound.

Security is a trade-off between minimizing the budget and covering more attack vectors. For practical reasons, we assume the hardware and cloud providers are trusted, as well as the way microservices are deployed provides high degree of isolation. We believe that the given model is the most realistic approximation of the real world and reaches the limits of what software developers and security engineers will be willing to accept in practice. The following discussion is centered around this model.

IV. Security Implications of Microservice Design

Several important security properties emerge as side effects of microservice design. Herein, we describe and evaluate the properties distilled from conducting a careful literature review on the subject [1]–[4], [11]–[13], [27], as well as discussions with practitioners and personal experience. We also provide an interpretation of the implications.

a) Do one thing and do it well: The properties of context boundary, design around business concepts, and Domain Driven Design (DDD) usually results in a smaller codebase per microservice. If we attempt to measure microservices size in lines of code (LOC), then the more LOC there are, the more bugs the service has. LOC and bugs are statistically correlated. More bugs in general means more exploitable bugs in particular. Less complex code is easier to maintain. A smaller codebase of individual microservices results in a smaller attack surface given the other conditions, such as the overall code quality, remain the same. This statement can be further supported by reduced cognitive complexity of the code that facilitates better code comprehension for individual developers. This is not necessarily true for system architects who still need to maintain a more global view of the system.

b) Automated, immutable deployment: Services should be immutable: to introduce permanent changes to microservices, services should be rebuilt and redeployed. Microservices immutability improves overall system security since malicious changes introduced by an attacker to a specific microservice instance are unlikely to persist past redeployment. Automation should be leveraged in maintaining the security infrastructure. Immutability aids the security of microservices similarly to how immutability promotes correctness in programming languages [30].

c) Isolation through Loose Coupling: Both SOA and microservices are built around the concept of loose coupling. Microservices take the concept even further by emphasizing the share nothing principle and strict data owning. This implies that each service can be isolated: only able to access the information it needs, and only able to access the particular services it needs. This limits the damage should an individual service be compromised.

Better isolation as an inherent security benefit of microservice design has been discussed by Otterstad & Yarygina [7]. Fetzer argues [6] that microservices can be used to build critical systems where integrity, confidentiality, and correct execution inside microservices is warranted by secure containers and compiler extensions. Such secure containers were implemented as Docker containers using an Intel Software Guard Extensions (SGX) enclave that can protect the microservices running inside such secure containers from OS, hypervisor, and cloud provider level attacks—with a degree of isolation for each component that is much greater than what is achieved in typical monolithic applications.

d) Diversity through System Heterogeneity: Distributed systems are often heterogeneous systems. Microservice architecture embraces this fact by allowing the individual components to be written in any programming language and/or technology given that they retain same interfaces (service contracts). This results in natural diversity of components in the microservice architecture.

Otterstad & Yarygina [7] suggested diversification of microservices as a mitigation technique against low-level exploitation. Although system diversity as a security inducing property is not new, the application of it in microservice setting is. Diversity in computer systems can be achieved in many ways including the use of different programming languages, compilers, OS/basic images. Microservice design philosophy readily allows for such approaches to be taken. Even coexistence of older and newer versions of the same microservice adds to the heterogeneity of the system. Otterstad & Yarygina [7] has also suggested use of N-version programming to improve microservice security.

e) Fail Fast: Although fault tolerance does not directly translate to security, we believe it contributes enough to be listed. If the main point is to disrupt the service, such as in case of Denial of Service (DoS) attacks, fault tolerance does directly translate to security. In contrast to monolithic systems where a failure is often total, distributed systems can be characterized with partial failures where only some of the nodes fail. A microservice network should tolerate the presence of partial failures and limit their propagation. The Circuit breaker pattern [26] prevents cascading failures

and increases overall system resilience by adjusting the node behavior if the network interactions with its peers fail partially or completely. Following the fail fast principle will decrease the likelihood of attacks succeeding and minimize the possible damage. Fundamentals of fault tolerance in distributed systems can be found in a book by Tanenbaum [14].

V. Emerging Security Practices

Although there are currently few industry practices for microservice security, some interesting trends present themselves. The first one is the use of Mutual Transport Layer Security (MTLS) with a self-hosted Public Key Infrastructure (PKI) as a method to protect all internal service-to-service communication. The second trend is use of tokens and local authentication. Both approaches are leaning towards the defense in depth approach to security and further support our attack model presented in Section III-B.

A. Mutual Authentication of Services Using MTLS

While sharing the same underlying concept, two different solutions for establishing trust between microservices were developed simultaneously by Docker and Netflix.

1) Docker Swarm Case: Docker Swarm is a container orchestration solution and clustering system for Docker that allows building distributed systems. Docker Swarm is particularly interesting because a) it is a popular platform for implementing microservices; b) it has a variety of built-in security features. MTLS is used by all the nodes in a swarm to authenticate each other, encrypt all the network traffic, and differentiate between worker and manager nodes [31], [32]. Docker Swarm automatically deploys its own PKI to provide identity to all the nodes.

The first manager node generates a new self-signed root Certificate Authority (CA) along with a key pair. A hash of the root CA and a randomly generated secret are sealed into a token that is provided to all other nodes during (re)deployment. To join a swarm, a node verifies the identity of the remote manager based on the token, generates a preliminary certificate data and sends a certificate signing request (CSR) to the manager together with the token. After verifying the secret from the token, the manager issues a certificate to the node. When a node needs to connect to another node, both nodes will authenticate to each other and set up a TLS tunnel, using the certificates issued to them by the CA. See Figure 3.

The default behavior for the nodes is to automatically renew their certificate every three months, but shorter intervals can also be used. The update does not happen simultaneously for all the nodes but instead takes place within given time-frame due to security reasons. Docker Swarm also supports rotation of the CA certificate and embedding into already existing PKI. Another popular container orchestration solution, Kubernetes, does not support MTLS with automated certificate provisioning at the moment, but the work on Kubelet TLS Bootstrap feature is ongoing.

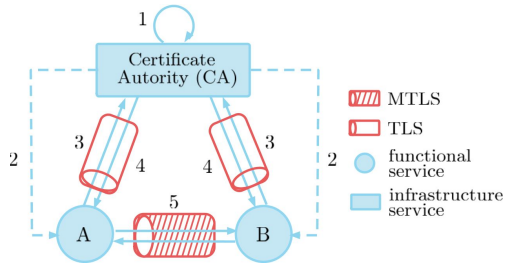

Fig. 3. A generic solution for microservice trust based on MTLS. 1) CA service generates a self-signed root certificate and a shared secret. 2) A cryptographic hash of the certificate and a shared secret are extracted from the CA service and provisioned into the new benign service either automatically or manually. 3) A benign service establishes a one-way TLS connection with the CA service, verifies CA identity using the provisioned hash of the CA certificate, and if successful submits a CSR and shared secret. 4) Upon successful verification of the received CSR and shared secret, CA service signs and sends back the newly issued certificate. 5) Two benign services communicate over MTLS.

2) Netflix Case: Netflix internal microservice network utilizes a PKI based on short-lived certificates for TLS with mutual authentication [33]. The Netflix approach builds on the notions of short- and long-lived credentials. The long-lived credentials are provisioned into the service during a bootstrap procedure, stored either in Trusted Platform Module (TPM) or SGX, and are required to obtain and update short-lived credentials. While the Docker source code is publicly available, the Netflix PKI solution is not.

The idea of issuing certificates with a short lifetime as a solution to the certificate revocation problem on the Web has been suggested before [34]. The short expiration time of certificates limits the utility for revocation mechanisms.

TLS with mutual authentication addresses problems of service authentication and traffic encryption, but not service authorization. Moreover, user to service authentication and authorization are still left to the discretion of developers.

B. Principal Propagation via Security Tokens

After a user has been authenticated by the gateway, the microservices behind it will be processing user's requests. Microservices should be aware of the user authentication state, i.e. whether the user was authenticated, and what the user's role is in authorization context. The user needs to be identified multiple times in each service down the operation chain, as each service calls other services on the users behalf.

1) Security tokens and relevant standards: Token-based authentication is a well known commonplace security mechanism that relies on cryptographic objects called security tokens containing authentication or authorization information. A security token is created on the server side upon the successful validation of the clients credentials and given to the client for subsequent use. Security tokens substitute the client's credentials within limited time-frame. Token-based authentication via HTTP cookies is a prominent example. Fu et al. [35] gives a detailed security evaluation of the approach.

Token-based authentication advanced even further with the widespread adoption of OpenID Connect [36], a security

standard for single sign-on and identity provisioning on the Web. The same functionality is provided by the Security Token Service (STS). STS is a component of the WS-Trust standard [37] that extends WS-Security standard with methods for issuing, renewing, and validating security tokens. Other relevant standards are JSON security standards: JSON Web Signature (JWS), Encryption (JWE), and Token (JWT).

2) Reverse Security Token Service: A noteworthy trend in the industry is the use of JWT for principal propagation within the microservice network. Multiple informal records of the approach can be found [38]. Although no formal description of it exists in scientific literature, those familiar with the above-listed security standards will find the suggested approach closely related to the existing standards.

Token-based user-to-service authentication where each service understands tokens allows transporting user identity and user session state through the system in a secure and decentralized manner. After the user is authenticated with the authentication service, a security token representing the user will be generated for *internal* use within the microservice network. In this paper, we refer to a separate service that is responsible for the token generation as a Reverse STS.

Limited lifetime of the security token is achieved by including an expiration time in its body. For security reasons, a shorter token lifetime is desired. Information about the user and intended audience can be included if needed. The token will be passed to the microservices involved in processing the request. Before executing the received request each microservice will validate the adjacent token using a corresponding public key. Token validation is a mandatory first step of the request processing. See Figure 4.

This approach fits well with the second design principle of distributed systems: individual nodes make decisions based on locally available information (see Section II-D). It facilitates loose coupling of the services and is highly scalable while having no overhead of a centralized solution. Moreover, there is good tool support for this approach. OAuth 2.0, a standard for delegated authorization, and OpenID Connect, an authentication layer on top of OAuth 2.0, can be tailored for inter-service security.

There are three caveats with this approach. The first is an assumption that the clock synchronization problem is non-existent. To validate tokens, both the token issuing node and the node performing validation must have their clocks synchronized. This is usually straightforward to handle, e.g., using NTP. The second one is that the tokens must be sent over a protected channel, i.e. TLS, otherwise the tokens can be intercepted and re-used within their validity period. Having short validity time is a partial solution to this problem. The third one is that the private key of the token issuing service must be kept safe at all times. If the private key is compromised, any user can be impersonated by the attacker.

C. Fine-Grained Authorization

Although no prominent trends for microservice authorization exist in industry at the moment, we will mention several promising approaches.

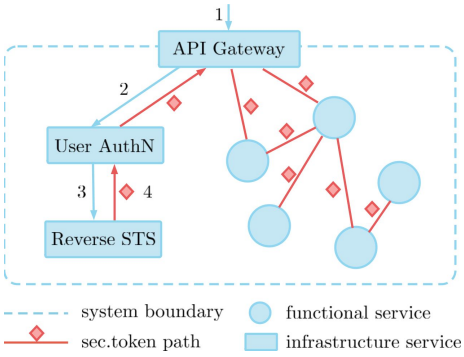

Fig. 4. A generic token-based authentication scheme for microservices that enables a user-to-service authentication and identity propagation based on cryptographic tokens. 1) Incoming user request hits API Gateway. 2) API-Gateway prompts user for authentication by redirecting to the dedicated user authentication service. 3) Requesting a security token. If user is authenticated successfully, the authentication token service generates a token that represents the given user inside the system. 4) Returning a security token. The token is passed alongside the user request to the downstream services. This token is designated for internal use only and is not given to the user.

1) Security Tokens for User Authorization: Various access control mechanisms exist including Role Based Access Control (RBAC) and Attribute Based Access Control (ABAC). RBAC and its predecessors are user-centric access control models. Therefore, they do not account for relationship between the requesting entity and the resource. For fine-grained authorization on resources, such as access to a specific API call, ABAC should be used.

Security tokens can include authorization information. For example, RBAC authorization roles can be incorporated into JWT tokens as an additional attribute.

2) Inter-Service Authorization Based on Certificates: Use of digital certificates for authorization rather than authentication was first suggested in 1999, e.g. Simple Distributed Security Infrastructure (SDSI) [39] and Simple Public Key Infrastructure (SPKI) [40]. Later, Marcon [41] proposed a certificate-based permission management system for SOA.

Not all microservices are deployed equal: only microservices co-dependent by design should be able to call each other. If the microservice network already relies on MTLS and self-hosted PKI, the same PKI can provide the basic service to service authorization. A separate signing certificate should be created per microservice *type*. This certificate will be used to sign the certificates of all instances of the same type.

Let's assume there are three types of microservices: A, B, and C. There are multiple instances of each type. While B is connected to both A and C, no direct connection is allowed between A and C. The default rule is to trust no one. To allow access to B from A and C, all instances of B should be preconfigured to trust the certificates signed by certificate type A and C. To allow access to A and C from B, the certificate for type B should be added to A's and C's trust lists.

VI. Microservice Security Framework

As shown, no standard way to deal with microservice security concerns exists. The existing implementations are often closed source (Netflix MTLS), not directly portable to other environments (Docker Swarm MTLS), and in general not well documented or understood. Moreover, a performance cost of using the existing security solutions in a microservice setting is unknown. To partially address this problem we implemented a microservice security framework, *MiSSFire* that provides a standard way to embed security mechanisms into microservices. Furthermore, we evaluated the performance of the framework against a toy microservice-based bank system of our design *(MicroBank)*.

A. Design and Implementation

When designing the MiSSFire framework we tried to address the main microservice security challenge—the problem of establishing trust between individual microservices. The core design criteria were security, scalability, and automation. We followed the defense in depth principle and the attack model introduced in Section III-B. Security mechanisms that we implemented are heavily based on the emerging security practices from Section V, specifically mutual authentication of services using MTLS and principal propagation via JWT.

The framework consists of a set of infrastructure services that need to be up and running within a system and a template for a regular functional service. Currently, the framework is bundled with two infrastructure services that expose relevant REST APIs:

- *The CA service* is a core part of the self-hosted PKI that enables MTLS between microservices. It generates a self-signed root certificate and signs CSR from other services. See Figure 3 for more details.
- *The Reverse STS* stays behind a user authentication service (not included) and generates security tokens in JWT format. A new JWT is generated per user request. See Figure 4 for more details.

The template for a regular functional service simplifies integration with the infrastructure services by providing all the necessary functionality. Additionally, it forces use of MTLS for all connections and requires presence of JWT for all incoming requests.

B. Experiment

Although performance is not a security property, it has been an important deciding factor for adoption or rejection of security mechanisms in the real world. In this section we address the question of how the common security mechanisms that are bundled in our framework impact the performance of an actual microservice-based system.

1) MicroBank: To test the framework we needed an actual microservice application. For this purpose we developed our own fictitious microservice-based bank system that consists of the following microservices:

- *API gateway*: The main entry point to the system.

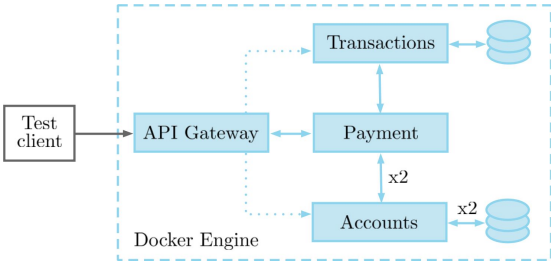

Fig. 5. Experiment setup: payment operation

- *Accounts*: Manages user accounts.
- *Payment*: Provides payment functionality.
- *Transactions*: Handles transaction operations.
- *Users*: Manages users.

The system was built using mainstream development techniques for microservices. The system is written in the Python v2.7 programming language. Although the microservice style does not dictate what communication protocols or styles should be used, in practice REST APIs and JSON format are the default choices. Each service in the system exposes a set of relevant REST APIs. This is done by using the web framework Flask and WSGI HTTP server Gunicorn. The number of workers per server is adjustable.

The 'shared nothing' architecture is a central part of the loose coupling concept in microservice world. Therefore, Accounts, Transactions, and Users microservices maintain individual SQLite databases. For the simplicity of the experiment there is only one instance of each microservice type.

The services of the bank model can be run as separate processes or in Docker containers. The whole bank model can also be run as a multi-container Docker application using the compose tool (though both the framework and the sample application can work independently of Docker). The tool automates configuration and creation of containers and allows to start the whole system with a single command. The source code has a partial unit-test coverage.

2) Methodology: The test client registers two users and opens bank accounts for them. Then, the test client carries out a series of payment operations between the two users. In the experiment, 50 concurrent test clients are started simultaneously, where each test client performs 100 sequential payment operations. To measure the system performance, an average execution time of 5000 payment requests is calculated on the client side.

The payment operation involves four microservices as shown in Figure 5. Several factors contribute to the time it takes to perform one payment operation. These factors are network delays and processing time inside microservices including database access time.

In the given setup, payment operations always succeed. If the operation is invalid, such as an attempt of transferring a negative amount, or if the recipient bank account is closed in the middle of the payment operation, the implemented system

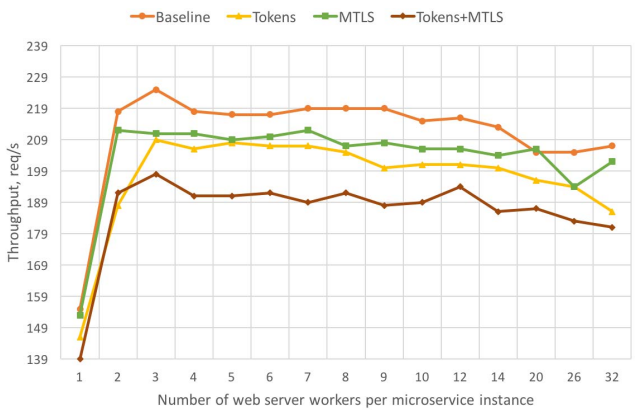

Fig. 6. Performance of the bank model under load of 50 test clients making payments

will roll back to revert the partially made changes. This would adversely affect the payment operation execution time.

The experiment consists of four parts:

- *Baseline.* Running the test client against the bank model with security features disabled.
- *Tokens.* Running the test client against the bank model with the Reverse STS service in place and JWT tokens validation.
- *MTLS.* Running the test client against the bank model with the CA service in place and all communication secured with MTLS.
- *Tokens+MTLS.* Running the test client against the bank model with all security features enabled.

The experiment was run on a MacBook Pro with a 2,6 GHz Intel Core i7 processor and 16 GiB of memory. The computing resources dedicated to Docker were 8 CPUs and 8 GiB of memory. Both the test clients and the bank model resided on the same machine, all communication was performed via localhost.

3) Results and Discussion: Figure 6 presents the results of our experiment. As expected the performance in the baseline case is higher than in the cases of tokens and MTLS. Closer inspection of the data shows that the tokens decrease performance by 7% on average, while the MTLS impact is around 4%. The most interesting aspect of this figure is that the difference between the baseline case and the case with all security mechanisms in place is relatively small, and accounts for around 11% in the given setup. Based on the fact that microservice solutions are slow in general, we believe this security overhead is still acceptable, especially for security critical applications. There is a relatively high cost to setting up HTTPS connections and validating tokens, which we reduce by pooling and reusing existing connections between services when possible.

Ueda et.al. [42] showed that the performance of the microservice version can be around 80% lower than the monolithic version on the same hardware configuration. This is a significant overhead that industry is willing to take and in many cases has already embraced. This also means that the overhead of having a microservice-based solution in first place is so high, that the impact of security mechanisms becomes insignificant in comparison.

From the chart, it can be seen that the best throughput is achieved with 3 workers (except for the MTLS case). When more workers are added the throughput starts to deteriorate slowly. These results deviate from an expected behavior where the throughput scales linearly up to the number of cores available, assuming the parallelizable portion of the program completely dominates the execution time. This may be related to the clients sharing the same CPU; though it may also be due to specifics of Gunicorn.

The relatively slow performance of our bank model can be attributed to the following: 1) Synchronous HTTP communication: sequential requests and no parallelization; 2) Slow database access. 3) Python is not as efficient as certain other programming languages; 4) Suboptimal configuration of the web servers. Also, the bank model is only a proof-of-concept and offers limited performance and scalability.

C. Evaluation

1) Security Considerations: Our framework relies exclusively on well-known security mechanisms and standards such as SSL and JWT. No new cryptographic primitives or protocols are introduced. Although implementation flaws are always possible, we need to rely on security and trustworthiness of the building blocks.

The CA service and the Reverse STS are security-critical. It makes sense to run these two services in a hardened environment. SGX-capable servers in the public cloud offered by Microsoft Azure [43] is one of possible solutions.

2) Performance Evaluation: Our experiments show that a microservice network introduces latencies on the order of milliseconds; in this setting the performance hit of basic security features becomes negligible.

3) Framework Limitations: Currently, the framework lacks multiple important security features. It is not a production-ready tool, and should mostly be seen as a proof of concept. The framework is overly simplified: it does not address authorization, there is no key rotation or key revocation mechanisms. Also, it is currently limited to Python.

4) Reproducibility and Future Work: To support reproducible research and allow others to improve on our results we released our code open source under GNU GPLv3 license. The source code of both MiSSFire framework and the MicroBank are publicly available at GitHub (https://github.com/yarygina/MiSSFire and MicroBank), including setup and benchmarking scripts. In future investigations, we intend to improve on the aforementioned limitations of our framework.

VII. Conclusion

In this paper, we have examined the microservice architectural style, with a particular focus on its security implications. Microservices bring together concepts from both service-orientation, distributed systems and fundamental software engineering principles of abstraction, reuse and separation of

concerns. This combination brings both new challenges that must be addressed, as well as old security challenges in a new wrapping. The microservice style of highly isolated, easily redeployable distributed components also implies new opportunities for better security, e.g., through increased diversity or restricting data access to only the services that need it.

As a concrete example of microservice-specific security, we have developed a small, openly available prototype framework for establishing trust and securing microservice communication with MTLS, self-hosted PKI and security tokens. Our case study shows that there is little extra performance cost to securing microservice communication, likely due to the overall high overhead of the communication itself.

With increased industry adoption of microservices, and the overall increasing threat level on the Internet, researching and developing secure microservices is crucial; and, with microservices seen as a lightweight, easy-to-use approach to SOA, we believe it is particularly important that security solutions are also lightweight, easy to use, and accessible to real-world developers.

REFERENCES

[1] S. Newman, *Building Microservices*. O'Reilly Media, 2015.

[2] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice (part 1): Reality check and service design," *IEEE Software*, vol. 34, pp. 91–98, January-February 2017.

[3] J. Lewis and M. Fowler. (2014) Microservices—a definition of this new architectural term. [Accessed Nov. 1, 2017]. http://martinfowler.com/articles/microservices.html

[4] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *CoRR*, vol. abs/1606.04036, 2016.

[5] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *Service-Oriented Computing and Applications (SOCA 2016)*, Nov 2016, pp. 44–51.

[6] C. Fetzer, "Building critical applications using microservices," *IEEE Security Privacy*, vol. 14, no. 6, pp. 86–89, Nov 2016.

[7] C. Otterstad and T. Yarygina, *Low-Level Exploitation Mitigation by Diverse Microservices*. Springer, 2017, pp. 49–56.

[8] Y. Sun, S. Nanda, and T. Jaeger, "Security-as-a-service for microservices-based cloud applications," in *Cloud Computing Technology and Science (CloudCom 2015)*. IEEE, 2015, pp. 50–57.

[9] S. Kim, F. B. Bastani, I.-L. Yen, and R. Chen, "High-assurance synthesis of security services from basic microservices," in *Software Reliability Engineering (ISSRE 2003)*. IEEE, 2003, pp. 154–165.

[10] N. Josuttis, *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[11] M. Richards, *Microservices vs. Service-Oriented Architecture*. O'Reilly Media, 2015.

[12] O. Zimmermann, "Microservices tenets: Agile approach to service development and deployment," *Computer Science - Research and Development*, pp. 1–10, 2016.

[13] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 2: Service integration and sustainability," *IEEE Software*, vol. 34, no. 2, pp. 97–104, Mar. 2017.

[14] A. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007.

[15] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Queue*, vol. 11, no. 11, pp. 30:30–30:44, Dec. 2013.

[16] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, The Royal Institute of Technology, Stockholm, Sweden, Dec. 2003.

[17] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack," in *25th USENIX Security Symposium*. USENIX, Aug. 2016, pp. 1–18.

[18] V. Mavroudis, A. Cerulli, P. Svenda, D. Cvrcek, D. Klinec, and G. Danezis, "A touch of evil: High-assurance cryptographic hardware from untrusted components," in *Computer and Communications Security (CCS'2017)*. ACM, 2017, pp. 1583–1600.

[19] T. Combe, A. Martin, and R. Di Pietro, "To Docker or not to Docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

[20] H. Takabi, J. B. D. Joshi, and G. J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security Privacy*, vol. 8, no. 6, pp. 24–31, Nov 2010.

[21] C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful web services vs. Big Web Services: Making the right architectural decision," in *17th Int'l World Wide Web Conf. (WWW2008)*, Beijing, China, 2008, pp. 805–814.

[22] T. Yarygina, "RESTful is not secure," in *Applications and Techniques in Information Security (ATIS 2017)*. Springer, 2017, pp. 141–153.

[23] (2014, Apr) The Heartbleed bug in OpenSSL library. [Accessed Nov. 1, 2017]. http://heartbleed.com/

[24] B. Möller, T. Duong, and K. Kotowicz. (2014, Sep) This POODLE bites: Exploiting the SSL 3.0 fallback. [Accessed Nov. 1, 2017]. https://www.openssl.org/~bodo/ssl-poodle.pdf

[25] (2017) Owasp top 10 – 2017: The ten most critical web application security risks. [Accessed Nov. 1, 2017]. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

[26] F. Montesi and J. Weber, "Circuit Breakers, Discovery, and API Gateways in Microservices," *CoRR*, vol. abs/1609.05830, 2016.

[27] C. Richardson and F. Smith, *Microservices From Design to Deployment*. NGINX, Inc., 2016.

[28] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium*. USENIX, 2014, pp. 719–732.

[29] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[30] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull, "Exploring language support for immutability," in *Int'l Conf. on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 736–747.

[31] D. Mónica. (BSides Lisbon 2016) MTLS in a microservices world. [Accessed Nov. 1, 2017]. https://www.youtube.com/watch?v=apma_C24W58

[32] Official Docker v17.06 documentation. Manage swarm security with public key infrastructure. [Accessed Nov. 1, 2017]. https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki

[33] B. Payne. (USENIX Enigma, 2016) PKI at scale using short-lived certificates. [Accessed Nov. 1, 2017]. https://youtu.be/7YPIsbz8Pig

[34] R. Rivest, "Can we eliminate certificate revocation lists?" in *In Financial Cryptography*. Springer-Verlag, 1998, pp. 178–183.

[35] K. Fu, E. Sit, K. Smith, and N. Feamster, "The dos and don'ts of client authentication on the Web." in *USENIX Security Symposium*, 2001, pp. 251–268.

[36] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "OpenID Connect Core 1.0," 2014, [Accessed Nov. 1, 2017]. http://openid.net/specs/openid-connect-core-1_0.html

[37] A. Nadalin, M. Goodner, M. Gudgin, D. Turner, A. Barbir, and H. Granqvist, "OASIS Standard Specification. WS-Trust 1.4," Apr 2012, [Accessed Nov. 1, 2017]. http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html

[38] B. Doerrfeld, "How to control user identity within microservices," Nordic APIs blog, 2016, [Accessed Nov. 1, 2017]. https://nordicapis.com/how-to-control-user-identity-within-microservices/

[39] R. L. Rivest and B. Lampson, "SDSI—a simple distributed security infrastructure," 1996, https://people.csail.mit.edu/rivest/sdsi10.html.

[40] C. M. Ellison, B. Frantz, B. Thomas, T. Ylonen, R. Rivest, and B. Lampson, "SPKI certificate theory," IETF RFC 2693, 1999, [Accessed Nov. 1, 2017]. https://tools.ietf.org/html/rfc2693

[41] A. L. Marcon, A. O. Santin, L. A. de Paula Lima, R. R. Obelheiro, and M. Stihler, "Policy control management for web services," in *Integrated Network Management (IM'09)*. IEEE, 2009, pp. 49–56.

[42] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *IISWC 2016*. IEEE, 2016, pp. 85–94.

[43] "Introducing Azure confidential computing," Microsoft Azure, Sep 2017, [Accessed Nov. 1, 2017]. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/