Journal of Ubiquitous Systems & Pervasive Networks
Volume 11, No. 1 (2019) pp. 17-25

Practical Defense-in-depth Solution for Microservice Systems

Kai Jander[a], Lars Braubach[b] , Alexander Pokahr[c]

[a]Brandenburg University of Applied Sciences
[b]City University of Applied Sciences Bremen
[c]Helmut-Schmidt-University

Abstract

Microservices are a widely deployed pattern for implementing large-scale distributed systems. However, in order to harden the overall system and when crossing datacenter boundaries, the authenticity and confidentiality of microservice calls have to be secured even for internal calls. In practice, however, in many cases no internal security mechanisms are employed mainly due to the increased complexity on backend side. This complexity arises as result of standard security mechanisms like TLS requiring secrets for each involved microservice. Building on previous work [19], in this paper we present a novel communication architecture based on roles that on the one hand guarantees a high level of security and on the other hand remains easy to manage. The approach provides encryption, forward secrecy and protection against replay attacks even for out-of-order communication.

Keywords: *Microservices, Security, Encryption, Confidentiality, Authentication*

1. Introduction

In order to allow horizontal scaling, large-scale software systems are generally distributed and therefore consist of multiple machines cooperating via networks. In this context, microservices are a novel development pattern for such systems in which the overall functionality is provided by a large number of small software components. These components use and provide software services and are developed and maintained by independent teams, which enjoy high degrees of autonomy regarding development and deployment of technical solutions as long as the service functionality is provided as agreed. Unlike traditional development approaches where development teams and operations teams are separated, microservice teams provide all development and operation requirements for their service (DevOps).

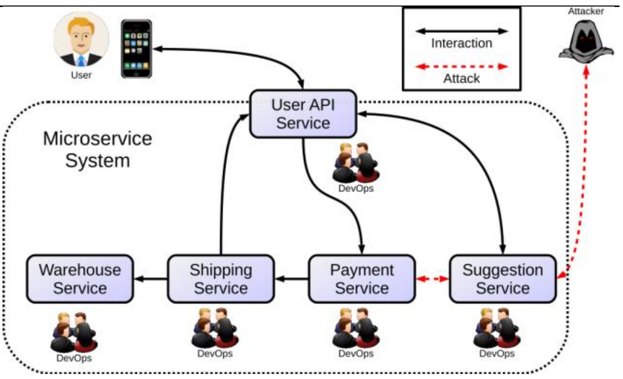

Figure 1: Example scenario of a microservice system

Figure 1 shows a simplified microservice system scenario. The system consists of five services working together to provide an application like a web shop to an external user. The user runs a client application such as a mobile app or a browser application, which accesses the external user API provided by the system. This API service is then able to invoke further internal services to provide the requested functionality: a suggestion service that provides purchase suggestions to the user, a payment service to process user payments, a shipping service that initiates the shipping with a logistics provider and a warehouse service for managing the physical products. Each of these services are maintained

* Kai Jander. Tel.: +49 152 534 80 400
Fax: - ; E-mail: jander@th-brandenburg.de

DOI: 10.5383/JUSPN.11.01.003

by an autonomous DevOps team that will develop, maintain and deploy the service.

Most microservice systems are based on standard web technology for communication between service and between the external API and the user, though internally each service team is free to choose any suitable technology and if agreed upon, other technology may be used.

However, the segmented nature of microservice systems actually lends itself towards compartmentalizing functionality and therefore mitigating the impact of successful internal and external attacks. For example, in the given scenario the suggestion service only has functional interaction with the User API service and therefore forms a functional domain with it. By restricting interaction of that service to that particular functional domain, the impact of the attack can be substantially reduced. However, in order to be able to form such functional domains, participants need to be able to authenticate membership of these domains. As a result, the focus of this paper will primarily rest on the provision of a suitable authentication mechanism for such funcitonal domain that can then be leveraged to offer ancillary functionality like encryption within the domain.

The communication between the user and the external API is usually secured through a combination of Transport Layer Security (TLS) [10] and additional authentication mechanisms such as passwords. In general, the most important aspect of the network security is the so called *perimeter defense*, which tries to protect the system boundaries from the outside internet [9]. As a result, security of individual microservices tends to be neglected [22] or considered to be handled by a separate security team [29].

This lack of internal security results in a system that, while unauthorized access is generally hard from the outside, allows an attacker to use all of the internal services easily once one of them has been compromised. In the example scenario, if the DevOps team of the suggestion service failed to adequately secure their service, an attack can then use the suggestion service platform to gain access to more valuable targets such as the payment service. Due to the interally open nature of the system this attack could succeed despite the fact that the suggestion service probably does not require access to the payment service.

As a result, a *defense in depth* approach for developing microservice systems is generally desirable. While technically possible, this is disregarded due to one or more three factors: First, in most cases, hypertext transfer protocol (HTTP) communication is employed and while HTTP Secure (HTTPS) [28] based on TLS is available, it provides purely channel based security between two hosts identified by certificates without extra work. As a result, only one identity for each side of the communication channel can be used.

Second, while additional authentication and identity management can be implemented using an external authentication system, this would need to be managed separately, possibly even in a centralized fashion.

Finally, the security has to be implemented in large part in the application code. The service developers would have to be sure to use secure communication channels such as HTTPS, ensure proper certificate checks especially if identities beyond host identities are employed. If an external authentication management system is used, tokens would need to be requested and checked.

All of these checks are often quite non-trivial: A public key infrastructures (PKI) certificate trust chain would need to be followed up to a trust anchor, with each step of the chain validated separately. Validity time intervals need to be validated for each certificate. Certificate permissions such as certificate authority signing need to be ensured. In addition, some systems include concrete capabilities within tokens that need to be coordinated with the token provider.

This requires a specialized knowledge set to be available to each of the DevOps teams and strongly distracts from the development focus of the team of providing the service functionality to the system. Some approaches would also undermine team autonomy by requiring coordination with external, possibly centralized systems to acquire certificates and capability sets.

As a result, a microservice system should be able to address the following three challenges in order to encourage developers to internally harden their systems:

Role Support:

How can service communication be secured independently of communication channels while providing potentially multiple roles to be attached to the communication?

Autonomy:

How can authentication be provided in a flexible manner that allows each DevOps team to make independent decisions regarding identities?

Ease of Use:

How can this be achieved in an easy-to-use manner that does not require extensive expert knowledge within the team and without resorting to centralized dependencies for the team?

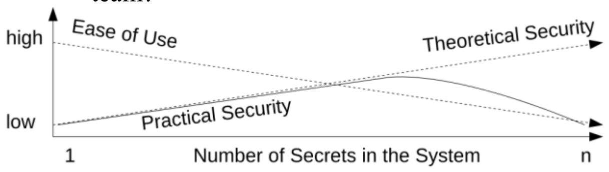

Figure 2: Trade-offs: Security versus ease-of-use

In Fig. 2 the well-known relationship beween security and ease-of-use [26] (increases in one dimension will typically lead to a decrease in the other dimension) is illustrated with respect to the microservice scenario. It can be seen that defense in depth needs at least one secret for the access of services. Of course, one secret is easy to handle but comes with the danger that an attacker only needs to gain access to one service in order to compromise the whole backend. On the other hand, one could enforce maximal theoretical security by having independent secrets for each of the services. This makes establishing the communication between services far more complex and error-prone. It could even lead to a reduced security level as DevOps begin working around security measures when those are perceived as unreasonably strict and labourious [24]. Thus, the security level in practice will be best when a good trade-off can be found beween security effort and ease of use of the system. With respect to microservice systems this means that a solution should be aimed for avoiding both extremes.

In the next Section existing microservice approaches for service authentication and encryption are discussed. Afterwards, in Section 3 fundamental requirements for a solution are presented and in Section 4 a novel approach addressing these requirements is conceptually described. Details of its implementation are introduced in Section 5. Thereafter, in Section 6 an evaluation of the approach in the context of an example scenario is discussed. A summary and an outlook on future work are given in Section 7.

2. Microservice Approaches for Service Authentication and Encryption

Since microservices are a design pattern and not restricted to any particular technology, any applicable technology can be used to develop distributed systems based on the approach. For example, the developers of a microservice system may always choose to implement their own internal security approach, however, this would require a high degree of expert knowledge in the development teams as mentioned.

Table 1: Approaches for securing microservice systems

	Role Support	**Autonomy**	**Ease of Use**
Plain TLS	- (No Roles)	o (lightweight key management effort)	o (PKI knowledge req., certificate mngt. For O(n2) connections)
TLS + Auth Service	+ (roles via auth. service)	- to o (increased effort for role management)	- (same as plain TLS + central mngt. of auth. service)
Microservice Framework	- (only one role)	o (same role for all services)	o (bus management)

As a result, existing technologies and frameworks are used to support microservice development. When based on the three challenges in Section 1, the most common technologies used in microservice systems can be categorized into three different approaches as shown in Table 1.

The first approach is relying on TLS to secure a connection between two hosts. This includes the common case of basing the service invocation of the microservice system services on representational state transfer (REST) [14], in which case HTTPS can be used. TLS supports authentication of both sides of a communication channel through server and client certificates. Provided the authentication is successful and sufficient, the following communication is encrypted and authentication based on the original certificates is ensured.

However, the most common use case of TLS authenticates only the server side based on host names using a public PKI. If client authentication is required, it is usually performed in-band through the use of a separate password or similar authentication. The use of client certificates is rare and support would have to be explicitly invoked by the application code. Furthermore, authentication based on host name only may be insufficient. While TLS does not strictly mandate how the certificates are used and validated, thus allowing custom authentication approaches, these complex approaches must be implemented in application code.

Authentication extensions for TLS are available [8], but library support for them is generally poor. For example, RFC 8492 [18] describes an extension for securing password-based authentication, however, standard Java libraries do not offer the necessary cipher suites.

In any case, the implementer would still be forced to deal with detailed authentication aspects if they are used. This means that there is a tradeoff between ease-of-use and sophisticated authentication styles such as roles. With server-side-only authentication, the implementation effort is moderate, but role support is not available. Team autonomy is restricted by the use of an external PKI.

The second approach attempts extending host-authenticated TLS with in-band authentication options. In most cases, a trusted third party is used to issue permits in the form of tokens like e.g. JSON Web Tokens (JWT) [20]. The trusted third party can be either implemented manually as a separate service or with a single sign-on framework such as OAuth2 [17]. Some approaches can be quite advanced such as Shibboleth [21], which allows federated security that can exhibit higher degrees of autonomy for the DevOps team. However, deploying and integrating such systems is very complex and once again requires specialized knowledge.

Finally, there are microservice frameworks that support the development of microservice systems by providing a middleware for service acquisition, invocation and communication. They also offer programming models to deal with the distributed and concurrent nature of such systems. While microservice frameworks offer the potential for a well-integrated security solution that enable easy implementation of security concepts, features offered by such frameworks are often quite limited.

For example, the Vert.x framework [12] offers an event bus for communication between services. While the bus itself can be secured using TLS, every service that has access can use all capabilities of the bus without limitation. As a result, defense in depth would have to be built on top of it and implemented by the service providers. Another example is the Lagom framework [11], which is based on Akka [3]. Here, the same issue applies: TLS is available, more advanced authentication and defense in depth is left up to the service implementation.

Since microservice frameworks are comparably easy to use, offer a high degree of transparency and come with a helpful programming model, it would be advantageous for microservice frameworks to also provide flexible and autonomous security solutions that allow microservice systems to provide defense in depth. In the following sections we will present a solution for the microservice framework Jadex which provides a high degree of security, high performance while allowing for easy usage and autonomous management.

3. Requirements

The basis of communication for services is the exchange of messages. In order to support multiple roles getting attached to messages, all messages need to be encrypted and associated with one or more identities. The goal of a security system is to provide the service implementation with the identities associated with the message and the assurance that confidentiality was maintained and the associated identities are verified. The service implementation can then implement the authorization layer based on the identities provided.

Roles are a useful pattern in this regard since they are not restricted purely to an identifier of a single entity but can be claimed by multiple identities if necessary. A role that can only be claimed by a single entity therefore becomes equivalent to that entity. As a result, specific support for entity (e.g. particular services or hosts) are not necessary and can be subsumed as roles.

Claims to a role can be proven by a prover to a verifier through a number of means, typically either using a shared secret in possession of both the prover and verifier or using a digital signature generated by the prover with a secret key that is verified by the verifier with a corresponding public key that follows a trust chain to a trust anchor for the role.

While digital signatures tend to be the most powerful approach, in particular allowing the verifier to verify roles without being able to claim them, it generally requires complex management of a public key infrastructure (PKI). If the use case is simple enough for an approach based on a shared secret, this overhead can and usually is avoided as can be shown by the continuing popularity of WPA2-PSK (Wi-fi Protected Access with Pre-Shared Key, see [25]) for easy wireless network deployment.

As a result, a solution should therefore be capable of supporting both approaches, allowing the service implementers to pick an approach most suitable for their use case. Furthermore, passwords as a particular type of shared secret, despite their weaknesses, are widely used due to their convenience and should therefore be also supported in some fashion.

Confidentiality should be ensured by means of authenticated encryption. The authentication can be based on the role identities of the participants. Furthermore, if possible, forward secrecy should be provided through the use of ephemeral keys to prevent compromising past communication if long-term keys are exposed.

Finally, since the communication may be routed through intermediary systems, these should be unable to read the message content. Additionally, they should be prevented from modifying the message content without the recipient noticing the modification. Finally, the message should reveal as little information to the intermediary as possible but enough to ensure delivery is possible.

In summary a defense-in-depth solution should support at least:

- A role-based authentication model
- Peer-to-peer authentication (without trusted third party)
- Flexible secret mechanisms including e.g. keys and passwords
- End-to-end security

4 Solution Concepts

A microservice system generally consists of multiple processes, often server applications like web servers, that are distributed on multiple machines, which can be either virtual or real. Each process can offer one or more services and make use of other services. Since each service within a process has access to the memory space of the process and could therefore undermine any further subdivision, it makes sense to treat those processes as a single entity with regard to the roles it has, regardless of the number of services provided by the process. If further separation between services is needed, the process providing multiple services could be split up in multiple processes for each service.

The basic approach for setting up secure and authenticated communication between two of such processes is the execution of a key exchange between those processes and authenticating that exchange with all of the roles available to each process. The processes would then be able to verify both that the exchange was authenticated (if at least one role could be verified) as well as associate all verifiable roles with the exchanged key. The resulting ephemeral key can then be used in a symmetric authenticated encryption scheme. Messages that are encrypted and can be validated using the ephemeral key can then be tagged with the roles that were verified during the key exchange.

Based on the requirements, the resulting system should support role authentication based both on public/private key pairs as well as shared secrets. Since shared secrets are further subdivided into passwords and keys, three approaches can be differentiated: passwords, keys and asymmetric key pairs as represented by X.509 certificates.

In this scenario, the use of public/private keys is relatively straightforward: The contributions to the key exchange are signed for each role with the private key and the public key chain is transmitted with the signature to the other participant who can verify the signature using the public key trust anchor associated with each role, resulting in a list of verified roles.

Table 2: Overview of authentication approaches

	Convenience	**Entropy**	**Asymmetric Roles**
Password	High	Low	No
Symm. Key	Medium	High	No
Key Pair (X.509)	Low	High	Yes

As shown in Table 2, while keys and passwords are technically both a type of shared secret, they are distinguished here with regard to their assumed entropy. Keys are assumed to contain sufficient entropy to make any brute force attacks computationally infeasible. Therefore, authentication using a key simply takes the key exchange contribution as input and generates a message authentication code (MAC) using the key. This procedure is then repeated by the other participant and the MACs are compared: If they match, then the associated role is considered verified.

Passwords are often considered to be a very convenient way of authentication. Depending on complexity, they can be memorized by the user and therefore can always be entered if the user is present. However, this convenience is often due to the generally low entropy of passwords: High-entropy passwords are often difficult for people to remember, so simpler, low-entropy passwords are chosen. Although, in production environments using microservices the memorability of secrets might not a problem and keys can safely be used there are scenarios in which passwords are beneficial as well. For example when prototyping systems lightweight solutions are helpful as changes might be required frequently. Additionally, in all cases when client authentication is obligatory, both options should be considered. Passwords e.g. allow a user to switch devices and access services by simply entering the password. In contrast, using a key typically requires some form of data transfer e.g. using a key file, repository or communication channel.

However, low-entropy passwords are an easy target for brute force attacks: A completely random password that is 8 characters long containing, potentially, upper- and lowercase letters as well as numbers has an entropy of 6 2 8 &ape; 2 48 possible combinations and thus roughly 48 bits.

If an attacker is able to mount even a moderately fast brute force attack, the password will be discovered quickly. Brute force attacks tend to be especially effective if they can be mounted offline, i.e. without interacting with participants who know the password.

As a result, a simple MAC using the password as the key is insufficient since it would allow an attacker to perform a fast, offline brute force attack which could not be resisted by the password's low entropy. WPA2-PSK attempts to resolve this issue by using an key derivation function (KDF). A key derivation function performs a large number of both CPU- and memory-intensive operations such as expanding and contracting hashes on the password and using the result similar to a high-entropy key. This approach can slow down brute force attacks considerably since they are required to repeat the same number of expensive operations for each trial.

However, a major disadvantage of this approach results from the fact that legitimate participants each also need to perform the expensive operation albeit only once. As a result, the complexity of the KDF needs to be carefully balanced between slowing brute force attacks and effort required by legitimate participants. This situation is made worse by the fact that this balance shifts with hardware capabilities: As hardware gets more processing capabilities and memory becomes cheaper, the KDF iterations and memory footprint needs to be expanded to keep pace. Since this changes the output of the KDF, such changes are not backward-compatible.

A better approach is therefore to prevent the attacker from performing fast offline attacks and forcing them instead to perform online attacks, which can be slowed hardware-independently by the participants. Another advantage is that such attempts can be logged for later analysis and even stopped by establishing a maximum number of trials per given timeframe and an accompanying blacklist that will impede brute force attempts.

Forcing the attacker to perform online attacks can be accomplished by a set of protocols called password-authenticated key exchanges (PAKE). In such protocols, a shared secret is agreed upon in a key exchange protocol that is authenticated using the password. The end result of such an exchange is a randomly generated, high-entropy key. An attempt by an attacker to manipulate such an exchange results in the keys on each side to differ. The PAKE protocol chosen for this approach is the password-authenticated key exchange by juggling (J-PAKE [16]). For each password-authenticated role a J-PAKE exchange is set up which is then performed in parallel to generate a key for each password-authenticated role.

The resulting high-entropy key can then be used as if the role had been secured with a high-entropy key in the first place: A MAC is generated based on the key exchange contribution and the J-PAKE-generated key and verified by the receiver using the same generated key. If a mismatch occurs, it means that either the passwords did not match in the first place or an attacker attempted to manipulate the J-PAKE exchange. In either case the role is not accepted as valid and the exchange has to be either repeated or accepted without the role being accepted.

5 Implementation

The protocol was implemented in the Jadex Active Components framework,[7] [6] which offers message exchange and service invocation using a multi-transport overlay network. Since Jadex builds its service call layer on its message exchange layer, authentication and encryption was implemented for the message exchange, therefore automatically providing its capabilities to the service call layer as well.

Messaging in Jadex is transparent and can be performed using multiple types of communication channels such as TCP/IP [27] and Websockets [13]. When a message is being sent to a remote Jadex platform, the local platform attempts to find a communication channel, called transport. Potentially, there can be multiple transports to a remote platform and platforms can dynamically change available transports as well. While transports are chosen based on performance criteria, the choice is relatively consistent but messages can be sent using multiple different transports to the same platform. As a result, while Jadex is supposed to provide at-most-once semantics for messages, it does not preserve message order.

Figure 3 shows the handshake protocol between two participants who wish to communicate. The goal of this protocol is to establish an ephemeral symmetric key between the two participants and securely associate the key with the verifiable roles available to each side. The protocol starts either when the initiator attempts to send its first message to the responder or occasionally by participants with established key to negotiate a new key to maintain forward secrecy.

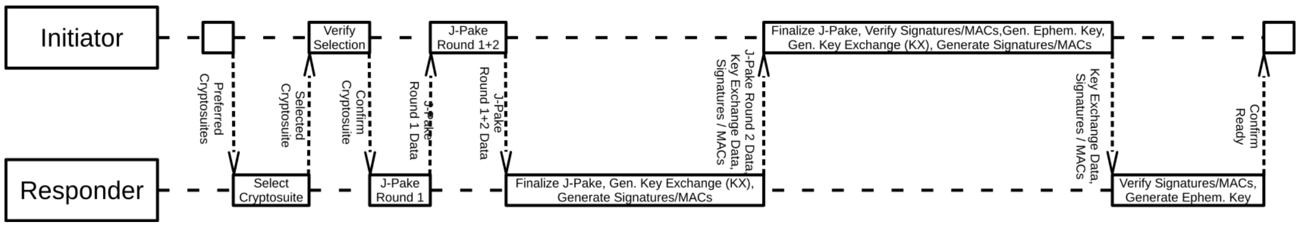
Initiator
Responder
Preferred Cryptosuites
Select Cryptosuite
Selected Cryptosuite
Verify Selection
Confirm Cryptosuite
J-Pake Round 1
J-Pake Round 1 Data
J-Pake Round 1+2
J-Pake Round 1+2 Data
Finalize J-Pake, Gen. Key Exchange (KX), Generate Signatures/MACs
J-Pake Round 2 Data, Key Exchange Data, Signatures / MACs
Finalize J-Pake, Verify Signatures/MACs,Gen. Ephem. Key, Gen. Key Exchange (KX), Generate Signatures/MACs
Key Exchange Data, Signatures / MACs
Verify Signatures/MACs, Generate Ephem. Key
Confirm Ready

Figure 3: Handshake protocol between participants

The initiator begins by sending a list of names of proposed cryptosuites which are sets of concrete cryptographic primitives used during handshake and later communication. The initiator must ensure to select only suites of acceptable security since the responder can select any of the proposed suites. The responder will then select a single suite based on its own preferences. If no match is found, the protocol fails. The selected suite is then sent to the intiator to confirm the selection. This is done for modularity, so an agreement about the suite is confirmed before the exchange itself starts. An optimization of this approach would be to interleave the subprotocols to a greater extent, however, since this exchange is independent of the communication channel, it is expected to occur only on occasion to ensure forward secrecy.

After suite selection, the J-PAKE exchange is started by the responder by sending the data set of the first round of J-PAKE for all its passwords to the initiator. The initiator then generates its own first round J-PAKE data set and calculates the second round information and sends both back to the responder. If no passwords are used, these J-PAKE parts of the handshake could be skipped.

The responder then finalizes the J-PAKE exchange. At this point, both sides have generated shared keys authenticated by their shared passwords. The key exchange algorithm is then initialized by the responder, which generates the key contribution of the responder. This contribution is hashed and authenticators are generated based on the hash: For both shared keys and password-derived keys a MAC is generated with the key contribution hash as input. For X.509 certificates, a signature for the hash is created. The second round data of the J-PAKE exchange, the key exchange contribution and the authenticators are sent to the initiator.

The initiator finalizes the J-PAKE exchange, generates its key exchange contribution and generates the ephemeral key. The third round of the J-PAKE protocol which confirms that the keys match can be omitted at this point since the generated keys are used for authentication only: If the keys differ, the authentication will fail. The received authenticators are verified, only roles with a verified authenticator are associated with the ephemeral key. The intiator also generates authenticators for its own key the same way as the responder contribution and sends those along with its key contribution to the responder.

Afterwards, the responder generates the ephemeral key and verifies the authenticator and sends a confirmation that it is now ready to exchange messages. After this exchange, both sides possess a shared ephemeral key that is associated with the roles that could be verified. If no role could be verified, it can be indication of either a man-in-the-middle attack or no matching roles between

the participants. In this case, the participants can either abort the communication or accept that the communication is untrusted and only has opportunistic encryption available.
The shared ephemeral key is then used in an authenticated encryption algorithm with nonces to encrypt messages. In order to prevent replay attacks, nonces are only accepted once by the receiver. Since the message order is not guaranteed, a sliding window is used: Successfully decrypted messages advance the window and mark the nonce as invalid. Lower nonces are valid until their message arrives, at which point they will be marked as invalid in the data structure. In order to limit memory consumption, any nonce below a fixed distance from the highest received nonce are automatically considered invalid whether they have arrived or not (unarrived messages are considered lost).
The current implementation only offers a single cryptosuite that attempts to secure the key exchange as much as possible while offering high symmetric performance after the exchange. The cryptographic primitives contained in this suite are as follows: The Blake2b [2] algorithm is used both as hash and MAC function. The authenticated encryption combines the ChaCha20 [5] symmetric cipher with the Poly1305 [4] [23] authenticator.
For the key exchange, two algorithms are run in parallel and their outputs are hashed together to hedge against each of their potential weaknesses: The Ed448 elliptic curve algorithm [15] is used for the historically good record of elliptic curve cryptograpgy, while the NewHope algorithm [1] is used to potentially provide quantum security, provided the authentication is also quantum-secure (i.e. MACs based on keys are used). By hashing their outputs together, an attacker would have to tackle both of the algorithms.

With regard to the challenges identified in the introduction, the solution supports *roles* as described above. Furthermore, the solution achieves the same

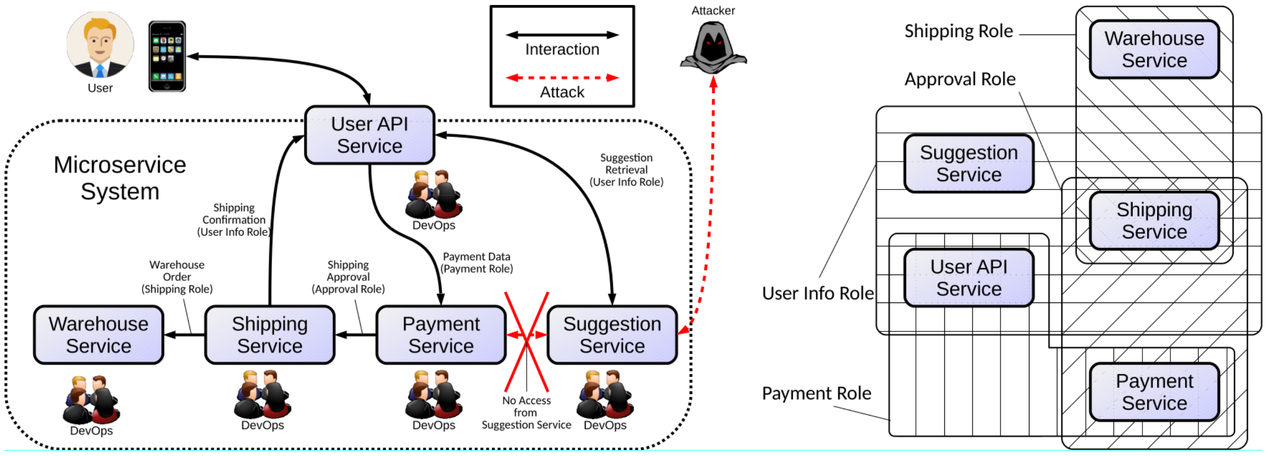
User
Interaction
Attack
Attacker
User API Service
Microservice System
Suggestion Retrieval (User Info Role)
Shipping Confirmation (User Info Role)
DevOps
Payment Data (Payment Role)
Warehouse Order (Shipping Role)
Shipping Approval (Approval Role)
Warehouse Service
Shipping Service
Payment Service
Suggestion Service
No Access from Suggestion Service
Shipping Role
Approval Role
User Info Role
Payment Role

Figure 4: Example scenario of a microservice system

autonomy as other microservice frameworks like Vert.x that also provide a ready to use messaging infrastructure. Considering *ease of use*, our first experiences with the new security architecture are promising. Roles can be defined bilaterally between two services, but also, e.g. as a convention that is agreed upon in the company, applied to the system as a whole, if needed. Yet, in no case a central authentication service is needed for roles, which reduces management effort (usability) and supports the DevOps approach (autonomy).

6 Evaluation

In Section 1 we have shown a simplified microservice system scenario based on a web shop application. In this scenario it was shown that in a standard microservice system design, in which all services are trusted, a successful attack on a weak but relatively unimportant service like the suggestion service can lead to subsequent severe breach of a more sensitive service like the payment service.

The right side of Figure 4 shows how the services can be assigned roles in the scenario: The user information role used to publish information to the user is available to the suggestion service, the shipping service and the user API service by distributing a secret among them. The information can consist of either a suggestion from the suggestion service or a shipping confirmation from the shipping service.
The payment role is used by the user API to submit payment information to the payment service, while the approval role is used by the payment service to authorize shipping to the shipping service.
Finally the shipping role is used by the shipping service to issue the final shipping order to the warehouse.
In total, given a pre-shared key approach, a total of only four secrets are needed to secure the system.
The left side of Figure 4 demonstrates how the system can mitigate and contain a service breach and prevent the escalation in that scenario. Identical to the original scenario, it is assumed that an attacker has successfully breached the suggestions service and is now trying to

leverage that breach to gain access to the payment service. However, in this scenario he was only able to capture the user information role, which the payment service does not recognize. The attacker is therefore able to manipulation purchase suggestions or send out fake shipping confirmations, but is unable to access the payment service of the system. As a result, while the attack was successful by definition, its impact has been contained and the attacker is unable to leverage the microservice system to gain access to additional services.

The roles for each service functionality can be negotiated and agreed by the two microservices planning an interaction independent of other project participants. In fact, it can be integrated in the process of designing the inter-service API between two services which aims to define the syntax and semantics of the interactions. As a result, the key management process can be seamlessly integrated in existing microservice processes, furthering both *ease of use* of the approach as well as *autonomy* of the microservice teams.

In addition, a centralized management structure such an identity management system is avoided. This has the procedural benefit of introducing no global administrative entity which every microservice team is required to interact with, a situation microservice architectures specifically try to avoid since such processes tend not to scale up to very large software projects, leading to loss of developer productivity.

Furthermore, any secret used by the system that ends up being compromised only affects a small part of the system and can be replaced through action of a small number of microservice teams rather than a global reintroduction of secrets such in the case of a compromised centralized identity management or in case of a global shared secret.

The system allows the use of single shared-secret roles in case of less critical functionality, trading some of security for simpler key management. For example, in the scenario the functionality of the shipping service that provides shipping confirmation as well as the suggestion service use the same role to provide information to the user API. This means that in case of a breach additional functionality will unlock for the attacker, but it can be considered a worthwhile trade for the simpler key-handling procedure of the user API and the rest of the services given the low impact of the functionality available.

In summary, the proposed solution is able to fulfill the requirements of Section 3. It enables end-to-end security for services using role-based authentication with peer-to-peer based validation. The secrets for roles can be chosen among different options including passwords and keys.

7 Conclusion and Outook

Security in microservice systems is often neglected and responsibility is delegated to an API gateway which shields the internal services and provides authentication as well as authorization. While convenient for the DevOps this approach lacks defense in depth and a hacker only needs to find a weakness in the gateway to get access to any of the hidden backend microservices. Current solutions with TLS are either complex by requiring a PKI or need certificate distribution mechanisms.

In order to prevent this, an approach is proposed that combines a high degree of security with ease of use by reducing the number of secrets to be managed in the system. The novel approach provides authentication and decentralized role-based authorization out of the box. The solution has been designed to be adoptable with minimum effort and tries to hide as many security aspects from the application layer as possible. It supports authentication via password, key and key pair and ensures that low entropy approaches are leveraged using key deviation functions. The approach has been realized in the Jadex framework.

As part of future work it is planned to optimize the efficiency of the protocol and to implement the protocol for standard REST via HTTP. The former can be achieved by reducing the number of messages while the latter can be done by first using the proposed protocol to establish a symmetric key that can be subsequently used to encode the content of all further HTTP traffic. This means that invocations could remain plain HTTP but are secured except for the headers.

References

[1] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchangea new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, 2016. USENIX Association.

[2] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, pages 119–135. Springer-Verlag, 2013. https://doi.org/10.1007/978-3-642-38980-1_8

[3] C. Baxter. *Mastering Akka*. Packt Publishing, October 2016.

[4] D. J. Bernstein. The poly1305-aes message-authentication code, November 2004. https://doi.org/10.1007/11502760_3

[5] D. J. Bernstein. Chacha, a variant of salsa20, January 2008.

[6] L. Braubach and A. Pokahr. Developing Distributed Systems with Active Components and Jadex. *Scalable Computing: Practice and Experience*, 13(2):3–24, 2012.

[7] L. Braubach, A. Pokahr, and K. Jander. Jadexcloud - an infrastructure for enterprise cloud applications. In S. Ossowski F. Klügl, editor, *Proceedings of Eighth German conference on Multi-Agent System TEchnologieS (MATES)*, pages 3–15. Springer, 2011. https://doi.org/10.1007/978-3-642-24603-6_3

[8] M. Brown and R. Housley. Transport layer security (tls) authorization extensions. RFC 5878 (Standard), May 2010. https://doi.org/10.17487/rfc5878

[9] Y. Choi, C. Sershon, J. Briggs, and C. Clukey. Survey of layered defense, defense in depth and testing of network security. *International Journal of Computer and Information Technology*, 3:987–992, September 2014.

[10] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246 (Standard), August 2008. https://doi.org/10.17487/rfc5246

[11] M. Eisele. *Developing Reactive Microservices - Enterprise Implementation in Java*. O'Reilly Media, Inc., May 2017.

[12] C. Escoffier. *Building Reactive Microservices in Java*. O'Reilly Media, Inc., May 2017.

[13] I. Fette and A. Melnikov. The websocket protocol. RFC 6455 (Standard), December 2011. https://doi.org/10.17487/rfc6455

[14] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[15] M. Hamburg. Ed448-goldilocks, a new elliptic curve. *IACR Cryptology ePrint Archive*, 2015:625, 2015.

[16] F. Hao and P. Y. A. Ryan. Password authenticated key exchange by juggling. In *the 16th International Workshop on Security Protocols, SPW'08*, 2008.

[17] M. Hardt. The oauth 2.0 authorization framework. RFC 6749 (Standard), October 2012. https://doi.org/10.17487/rfc6749

[18] Dan Harkins. Secure password ciphersuites for transport layer security (TLS). *RFC*, 8492:1–40, 2019. https://doi.org/10.17487/RFC8492

[19] Kai Jander, Lars Braubach, and Alexander Pokahr. Defense-in-depth and role authentication for microservice systems. In *Proc. 9th International Conference on Ambient Systems, Networks and Technologies*, Procedia Computer Science (open-access), pages 456–463. Elsevier Science, 2018. https://doi.org/10.1016/j.procs.2018.04.047

[20] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). RFC 7519 (Standard), May 2015. https://doi.org/10.17487/RFC7519

[21] R. L. Morgan, S. Cantor, S. Carmody, W. Hoehn, and K. Klingenstein. Federated Security: The Shibboleth Approach. *EDUCAUSE Quarterly*, 27(4):12–17, 2004.

[22] S. Newman. Microservice insecurity. *Container Solutions Blog*, July 2017. http://container-solutions.com/-microservice-insecurity/.

[23] Y. Nir and A. Langley. The websocket protocol. RFC 7539 (Standard), December 2015.

[24] Donald A. Norman. The way i see it: When security gets in the way. *Interactions*, 16(6):60–63, November 2009. https://doi.org/10.1145/1620693.1620708

[25] Institute of Electrical and Electronics Engineers (IEEE). Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pages 1–3534, Dec 2016.

[26] J. Pepin. Security vs. convenience. https://auth0.com/blog/security-vs-convenience/, 2018.

[27] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168. https://doi.org/10.17487/rfc0793

[28] E. Rescorla. Http over tls. RFC 2818 (Standard), May 2000. https://doi.org/10.17487/rfc2818

[29] M. Schöfmann. Security challenges in microservice implementations. *Container Solutions Blog*, January 2016. https://container-solutions.com/security-challenges-in-microservice-implementations/.