Navigating the challenges and best practices in securing microservices architecture

Mykhailo Kotenko[1,†], Dmytro Moskalyk[1,†], Valeriia Kovach[2,3,†] and Viacheslav Osadchyi[4,*,†]

[1] *Zhytomyr Polytechnic State University, 103 Chudnivsyka str., 10005 Zhytomyr, Ukraine*
[2] *Center for Information-analytical and Technical Support of Nuclear Power Facilities Monitoring of the National Academy of Sciences of Ukraine, 34a Palladin ave., 03142 Kyiv, Ukraine*
[3] *Interregional Academy of Personnel Management, 2 Frometivska str., 03039 Kyiv, Ukraine*
[4] *Borys Grinchenko Kyiv Metropolitan University, 18/2 Bulvarno-Kudriavska str., 04053 Kyiv, Ukraine*

Abstract

This paper explores the multifaceted challenges of securing microservices architecture, a modern software development approach prioritizing scalability and flexibility. The study addresses issues such as the expanded attack surface, ensuring secure inter-service communication, managing distributed data, and implementing access control mechanisms, all of which pose significant security risks to microservices systems. Through a detailed analysis of existing security practices, the paper highlights critical practices, including service isolation, secure inter-service communication, robust authentication, authorization mechanisms, and strategies for protecting data at rest and in transit. Additionally, the role of modern technologies such as service mesh and API gateway in enhancing the security of microservices systems is examined. The analyzed practices underscore the critical need for a comprehensive and multi-layered security approach that mitigates risks and helps maintain distributed applications' integrity, confidentiality, and availability. Furthermore, the paper provides essential security recommendations for organizations seeking to implement or optimize microservices systems.

Keywords

microservice architecture security, service isolation, securing data in transit, service mesh, access control mechanisms, API gateway, securing data at rest

CEUR-WS.org/Vol-3826/paper1.pdf

1. Introduction

Microservices Architecture (MSA) has rapidly become the cornerstone of modern software development, offering substantial scalability, flexibility, and reliability advantages. In the context of active digital transformation, MSA enables enterprises to swiftly adapt to market changes and technological advancements, making it a priority choice across various industries.

However, the widespread adoption of microservices is accompanied by significant security challenges. Despite its numerous advantages, MSA's decentralized and interconnected structure introduces new vulnerabilities. Each microservice, operating autonomously and interacting over a network, increases the risk of cyberattacks, which can lead to substantial security breaches and system-wide failures. Ensuring the security of such a distributed system requires innovative and comprehensive approaches to counteract the emerging threats in the context of rising cybercrime.

As the adoption of microservices expands, security issues are becoming increasingly pressing. Leveraging the potential of MSA while managing cyber risks is a critical factor for organizations striving to maintain their competitiveness in the digital age, where information security plays a pivotal role in ensuring seamless operations and business growth [1].

1.1. Overview of microservice architecture

MSA represents a significant advancement in software development, addressing the limitations inherent in monolithic systems and service-oriented architecture (SOA) by employing a more granular and adaptive methodological approach. MSA was first introduced at a seminar in May 2011, where the term "microservice" was used for the first time, and its definition was formalized by James Lewis and Martin Fowler in 2014 [2]. The defining characteristics of this architectural paradigm include the design and implementation of software as a collection of small, autonomously scalable services, each operating in an isolated process and communicating with other services through lightweight protocols (Fig. 1) [3, 4].

CPITS-II 2024: Workshop on Cybersecurity Providing in Information and Telecommunication Systems II, October 26, 2024, Kyiv, Ukraine
* Corresponding author.
† These authors contributed equally.

micha.kotenko@gmail.com (M. Kotenko); domoskalyk@gmail.com (D. Moskalyk); valeriiakovach@gmail.com (V. Kovach); v.osadchyi@kubg.edu.ua (V. Osadchyi)

0009-0002-7839-6538 (M. Kotenko); 0000-0002-4421-9325 (D. Moskalyk); 0000-0002-1014-8979 (V. Kovach); 0000-0001-5659-4774 (V. Osadchyi)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

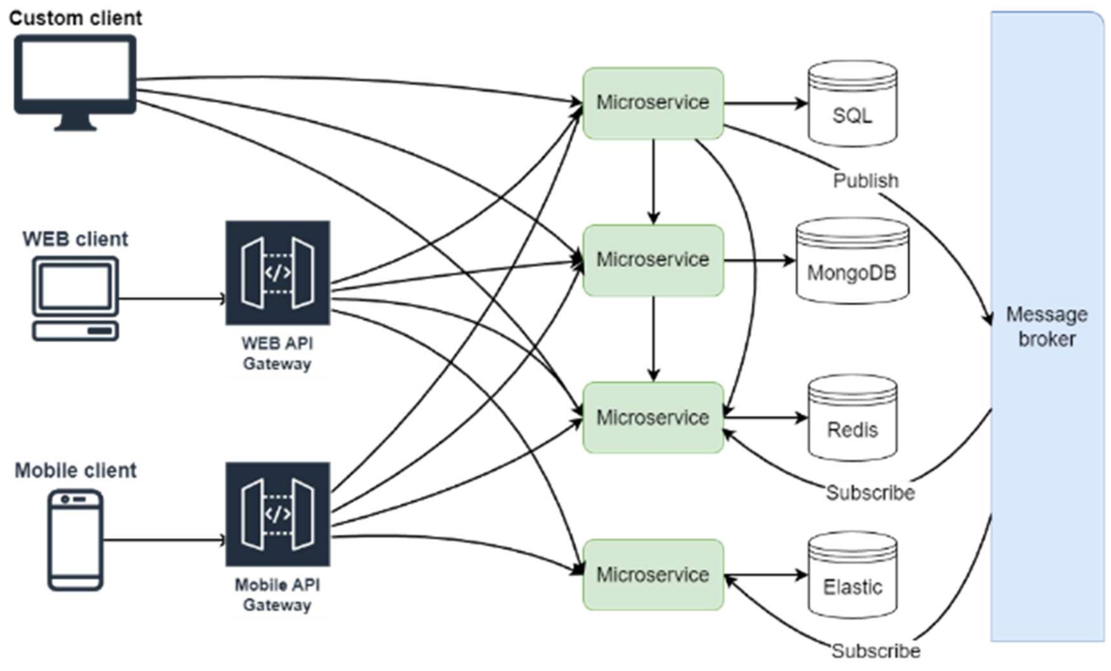


Figure 1: MSA diagram

The differentiation between MSA and SOA is distinct and significant. In contrast to SOA, which focuses on implementing a strategy of service reuse across a wide range of applications, MSA offers a strategy that minimizes the sharing of components, instead emphasizing the decomposition of services to ensure their maximum independence and autonomy [3]. This independence is critically important for the implementation of continuous integration and continuous deployment (CI/CD) methodologies, as it allows services to be developed, tested, deployed, and scaled independently of one another, thereby facilitating the use of optimal technology-stacks tailored to specific requirements and functionalities [4]. Additionally, MSA simplifies decentralized management of data and architectural decisions, enabling greater adaptability and flexibility in selecting technologies and architectural approaches [5].

The transition to MSA is often driven by the need to address the limitations associated with monolithic architectures, particularly in scalability, maintenance, and the dynamics of developing new features [6]. The microservices approach allows organizations to scale individual components according to demand independently, simplifies the management of smaller codebases, and facilitates rapid development iterations. Moreover, this architectural model enhances resilience to system failures by ensuring that the failure of a single service does not lead to a system-wide outage, and it promotes organizational adaptability by aligning service boundaries with corporate strategic directions [7].

The implementation of MSA is a complex process that involves analyzing the existing system, strategic transformation, implementing the new paradigm, and continuous management, support, and adaptation. This process not only reflects the need to revise development and management practices to align with MSA's decentralized and distributed nature but also highlights the necessity of addressing specific challenges. These challenges include network latency, data consistency, service orchestration, service integration, data management in a decentralized environment, comprehensive security measures, and developing and deploying productive monitoring and logging systems [2, 5, 6].

Empirical studies of the successful application of MSA by well-known companies such as Netflix, Amazon, and Uber illustrate the significant potential of this architecture in ensuring a prominent level of flexibility, scalability, and resilience in software development processes [2]. Implementing MSA and realizing its benefits requires organizations to understand its key principles and effectively manage the challenges associated with this approach.

1.2. Importance of security in microservices

Implementing robust security in microservices systems is critically important due to the critical role these systems play in modern application architectures. As companies increasingly rely on microservices for scalability and performance, ensuring their security is essential for maintaining trust, protecting sensitive data, and ensuring uninterrupted operations.

Microservices-based systems manage substantial volumes of sensitive and personal data. Safeguarding this information is paramount to mitigate breaches that could result in financial losses, legal ramifications, and reputational harm to the organization. A secure microservices architecture protects customer data, corporate intellectual property, and mission-critical data from unauthorized access and exploitation.

Secure microservices systems are also crucial for maintaining application reliability and availability. Security incidents, such as unauthorized access or denial-of-service attacks, can disrupt service operations, leading to downtime and degraded performance.

Compliance with regulatory requirements is another important reason for ensuring robust security in microservices systems. Many industries are subject to regulations that mandate data protection and specific security measures [8]. Failure to comply with these requirements can result in severe fines and loss of business. Prioritizing security enables organizations to meet regulatory demands and avoid associated risks [9, 10].

Security in microservices systems also preserves customer trust and confidence. Customers expect businesses to protect their information in an era of frequent data breaches and cyberattacks. Demonstrating a commitment to security enhances an organization's reputation, fosters customer loyalty, and provides competitive advantages.

1.3. Research purpose

This research aims to analyze the primary security challenges that arise during the implementation of MSA and review practices and strategies designed to address these challenges. The findings will serve as a foundation for developing recommendations that enhance the security of microservices systems, thereby ensuring the integrity, confidentiality, and availability of applications within a distributed architecture.

2. Security challenges in MSA

MSA offers numerous advantages, such as scalability, flexibility, and the autonomous deployment of services. However, these benefits are accompanied by significant security challenges that must be addressed to maintain the system's integrity and confidentiality.

The transition to MSA provides enhanced scalability and maintainability, but it also significantly increases the attack surface compared to traditional monolithic systems. MSA consists of numerous independently deployed services, each functioning as a separate network endpoint, which increases the potential entry points for malicious actors [11]. This distributed nature necessitates carefully protecting each service, as a breach in one could compromise the entire system [12].

As microservices scale horizontally, the attack surface expands proportionally as each service deployment introduces new network ports and application programming interfaces (APIs) [13]. This complicates the maintenance of uniform security standards, as different microservices may utilize various dependencies, libraries, and frameworks, each with vulnerabilities.

Frequent updates and deployments, which are characteristic of microservices, further exacerbate these security challenges by providing malicious actors with more opportunities to exploit unpatched vulnerabilities or misconfigurations.

Unlike monolithic applications, where components interact within a single process, microservices interact over a network, exposing the system to risks such as interception, spoofing, and unauthorized access [14]. Each interaction becomes a potential attack vector, necessitating careful consideration of security issues.

The complexity is further heightened by the diversity of communication protocols and schemes, such as synchronous messaging, asynchronous message queues, and event-driven mechanisms [15]. Each method has vulnerabilities, making it challenging to ensure consistent security across all service interactions.

Additionally, the MSA's dynamic nature intensifies security concerns. Services are often deployed, scaled, and terminated on demand, leading to constant changes in network topology. This makes it challenging to maintain secure communication channels and authenticate services effectively.

The distributed structure of microservices complicates security, particularly regarding data management and protection. Ensuring consistent data security across various services is challenging, as data is stored in multiple databases, caches, and storage systems [16]. This necessitates robust protection strategies, including encryption, access control, and continuous monitoring to detect breaches [17].

One of the primary challenges is ensuring that each microservice has the appropriate data access level. Microservices performing specific business functions may require separate datasets. It is critically important to ensure that only authorized services have access to and can modify data. Additionally, compliance with data privacy regulations demands strict control over data processing and storage.

Cloud environments exacerbate privacy concerns, as cloud technology consumers are increasingly worried about the potential misuse or compromise of stored information [17].

It is important to note that regulations such as GDPR, HIPAA, and CCPA establish stringent data protection standards, including encryption, auditing, providing users with the right to access and manage their data, and other measures. Compliance with these regulations requires implementing comprehensive security systems that account for the diversity of technologies and frameworks used in microservices.

Another challenge in MSA is ensuring the authenticity of each service involved in communication. If one service is compromised, it can affect other services, leading to potential application misuse [17]. To mitigate this risk, microservices applications must implement robust authentication mechanisms that allow interaction only between authenticated services [18].

Authorization in microservices systems also presents unique challenges due to the need for granular access control. Unlike monolithic architectures, where access control is typically centralized, microservices require distributed authorization mechanisms to manage permissions across various services properly [19]. This makes it challenging to ensure consistent authorization policies, especially in environments where services are frequently scaled. Additionally, each service must be granted only the minimum necessary privileges to perform its functions, limiting potential damage from a compromised service [17]. Excessive privilege allocation can lead to significant security risks, as attackers may exploit these excessive permissions to escalate their access and cause further harm.

Additionally, the heterogeneity of microservices environments, often involving multi-cloud deployments, complicates authentication and authorization management. Using diverse technologies and platforms requires a centralized yet flexible approach to security management to maintain consistent security policies across all services [18]. However, achieving such centralization without creating a single point of failure or a performance bottleneck remains a significant challenge.

3. Best practices for microservice security

3.1. Libertinus fonts for Linux

In the realm of MSA, service isolation is a fundamental aspect that must be considered to ensure robust security practices. Service isolation involves designing each microservice as an autonomous and isolated unit capable of functioning independently from others. The primary goal of service isolation is to contain potential security breaches within individual services, preventing the compromise of one component from affecting the entire system. Through careful isolation, each service is granted only the minimal resources and permissions necessary for its operation, significantly reducing the attack surface and the potential impact of security incidents.

Sandboxing is a service isolation technique that creates a controlled environment for executing programs by restricting their access to system resources. This technique can be successfully implemented through virtualization and containerization, which allow processes to be isolated, ensuring an important level of security and system stability.

Virtualization creates isolated environments through Virtual Machines (VMs), each with its own operating system and software (Fig. 2). Hypervisors such as Xen, VMware, and KVM manage multiple VMs, ensuring their independent and secure operation [20, 21]. Virtualization is particularly relevant in scenarios requiring complete isolation, such as multi-tenant cloud environments [21].

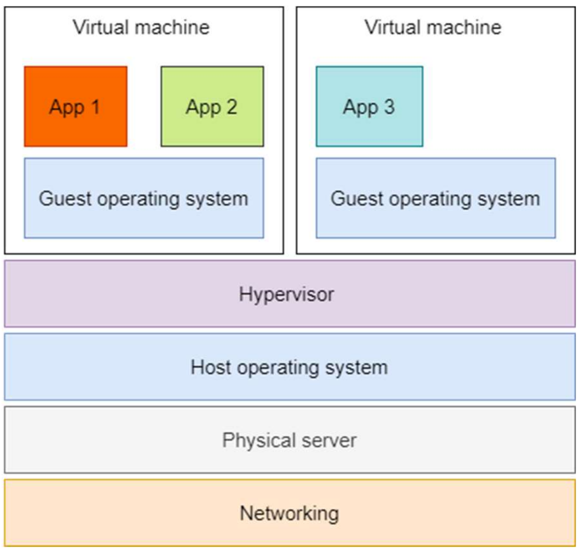


Figure 2: Virtualisation using VM

This technology allows consolidating multiple operating systems on a single physical server, optimizing hardware utilization and reducing infrastructure costs. Each VM operates independently, protecting against security breaches or resource conflicts [20, 21]. This reduces the risk of an attacker's lateral movement between services.

The "Service per VM" approach [22] involves associating each microservice with a separate VM. This approach ensures isolation, as issues or threats in one service do not affect others. The primary advantages include ease of managing dependencies, enhanced security through isolation, and the ability to use different operating systems and software for different services. However, there are drawbacks: resource costs can be high due to the need to allocate separate VMs for each service, and scaling may become challenging due to the limitations on the number of VMs that can be hosted on a single physical server.

Containerization is an evolution of virtualization that provides a lightweight and convenient way to isolate services. Containers encapsulate an application and its dependencies into a single portable unit that can run consistently across different environments (Fig. 3) [20]. This is achieved through container engines like Docker, which use operating system-level virtualization to isolate applications within namespaces and groups. Such isolation prevents processes in different containers from interacting, enhancing security by containing potential vulnerabilities within a single container [20, 23].

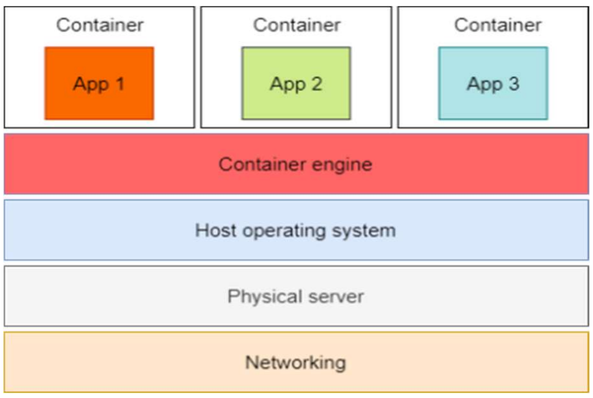


Figure 3: Containerisation on host OS

The "Service per container" deployment model [22] allows each microservice to run in its container. The primary advantages include ease of deployment and scaling, efficient resource utilization, and rapid startup and shutdown of containers. However, containerization also has its drawbacks: it offers less isolation than VMs, can present challenges in managing dependencies and network configurations, and requires orchestration tools to manage many containers.

Coordinated management of containers and their orchestration provides additional capabilities for ensuring the security of a containerized environment. Tools such as Kubernetes and Docker Swarm offer features that include resource constraint management, scheduling, load balancing, health checks, fault tolerance, and automatic scaling [23]. Kubernetes allows setting CPU and memory usage limits for each container, preventing resource exhaustion. Health check and fault tolerance mechanisms automatically detect and replace compromised containers, maintaining overall system security and availability [23]. Orchestration tools also support automatic deployment and scaling, which are vital for DevOps, and enforce security and compliance policies. For instance, Kubernetes can control network traffic between containers, allowing only authorized communication paths, thereby protecting sensitive data and preventing unauthorized access.

Resource isolation is a crucial aspect of both containerization and virtualization. Containers are used to manage and limit the resources available to each container, such as CPU and memory [20, 21]. Virtualization similarly abstracts hardware resources into multiple VMs, each with dedicated resources. This ensures that each VM operates independently without resource conflicts with other VMs on the same physical machine.

Such isolation of containers and VMs allows for the efficient allocation and scaling of resources, supporting the stability of critical services and reducing the risk of failures and attacks aimed at resource exhaustion.

Combining containerization and virtualization can provide a high level of security and system flexibility (Fig. 4). For example, running containers within VM combines the advantages of both technologies: optimal resource utilization and rapid container startup, along with the robust isolation and security guarantees provided by VM [21, 23]. This multi-layered approach offers additional protection: even if a container is compromised, the virtualization layer creates an extra security boundary, safeguarding the hardware and other VMs from potential threats [13, 21].

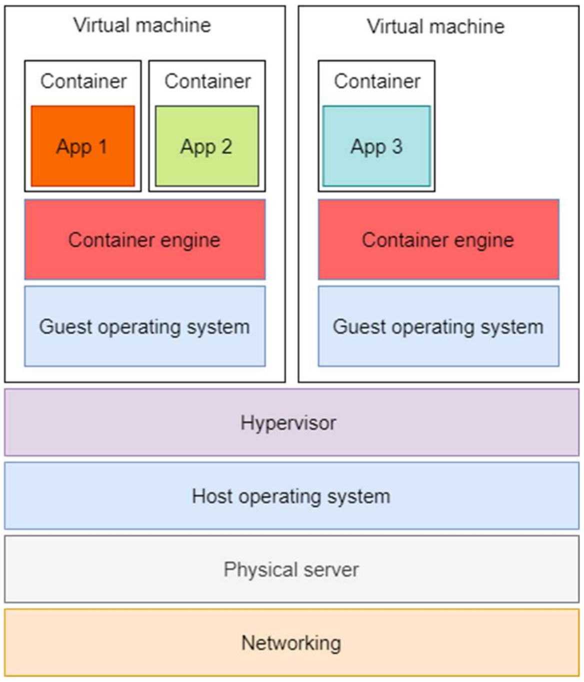


Figure 4: Combining containerization and virtualization using VM infrastructure

Micro-segmentation is an innovative security strategy designed to enhance the protection of microservices architecture. It involves dividing the network infrastructure into smaller, manageable segments, allowing security policies to be applied directly where the resources reside (Fig. 5). This differs from traditional network segmentation methods, such as VLANs and firewalls, which often do not provide sufficiently granular protection in dynamic cloud environments [24]. By implementing micro-segmentation, organizations can ensure that only authorized entities within the network can access specific applications or data. This approach significantly reduces the risk of lateral movement by potential attackers, thereby improving overall network security [25, 25].

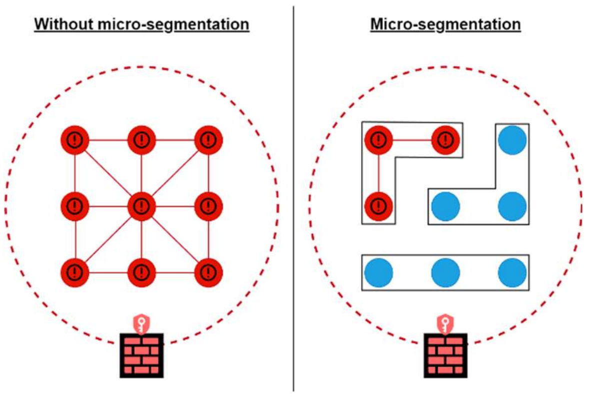


Figure 5: Network security comparison: traditional vs micro-segmentation

The use of micro-segmentation to secure microservices offers several significant advantages. First, this approach enhances threat detection and prevention by restricting communication between processes. Unauthorized access attempts are quickly identified as early indicators of potential intrusions. This approach aligns with the zero-trust security model, ensuring consistent enforcement of security policies across various infrastructures [26].

Furthermore, micro-segmentation supports compliance with security standards and regulatory requirements related to data protection and network security by ensuring proper isolation and safeguarding of sensitive data and critical applications.

Micro-segmentation is particularly valuable in dynamic environments where applications and workloads frequently move between on-premises servers and cloud systems. It ensures that security policies follow applications regardless of location, maintaining robust protection against attacks [25].

Implementing micro-segmentation requires a deep understanding of the network infrastructure and its components. There are several approaches to micro-segmentation, including native micro-segmentation, which leverages underlying infrastructure such as hypervisors or operating systems; third-party models that utilize virtual firewalls; overlay models with agent software on servers; and hybrid models that combine multiple approaches [24, 26]. Each of these approaches has its advantages, and the choice depends on the organization's specific needs and existing infrastructure.

For example, native micro-segmentation provides granular control without additional hardware, making it ideal for virtual environments. Third-party models offer centralized management of distributed virtual firewalls, which is suitable for large-scale deployments [24]. Overlay models enable dynamic policy enforcement through central controllers or orchestration devices, providing high visibility and control over workflow communications [26].

Achieving successful micro-segmentation involves several key steps. First, deep visibility into network resources is essential to identify all active applications and their dependencies. This can be accomplished using visualization and mapping tools [25]. Once these dependencies are identified, logical groups of applications can be created to implement security policies, ensuring the optimal group size to avoid overly broad or excessively narrow coverage [25].

The next step is the development of security policies. These policies must be carefully detailed and tailored to meet the specific security requirements of each application group. Once developed, the policies are implemented and continuously monitored to ensure compliance and proper functioning [26]. Monitoring involves tracking all traffic to detect anomalies and any breaches of policies. Additionally, enforcement mechanisms should be established to enable swift responses to identified threats, automatically isolating and investigating potential security incidents [25].

3.2. Securing data in transit

Ensuring secure data transmission between microservices is the next crucial element of the overall security strategy in MSA. Since microservices continuously exchange data over the network, it is essential to secure this process to protect the system from potential threats such as data interception or tampering. This section examines approaches to securing communication between services, including encryption and mutual authentication.

Transport Layer Security (TLS) is a protocol that secures communications between services in MSA. It operates below the application layer, providing end-to-end encryption at the transport level, which is crucial for maintaining the confidentiality and integrity of data transmitted between services [12].

Establishing a TLS connection begins with a “handshake” process (Fig. 6), during which upstream and downstream services exchange messages to agree on encryption parameters. Initially, the upstream service sends a “Client hello” to the downstream service, listing supported ciphers and protocol versions. In response, the downstream service sends a “Server hello”, where it selects the encryption parameters and provides its certificate for authentication. The upstream service verifies the validity and authenticity of the downstream service’s certificate. The upstream service proceeds with the connection establishment if the certificate is valid. Next, key exchange occurs. The upstream service generates a “premaster secret”, encrypts it with the downstream service’s public key, and then sends it. The downstream service decrypts the “premaster secret” using its private key. Based on this value, the upstream and downstream services generate session keys to encrypt further communication. All data exchanged between the upstream and downstream services is encrypted using these symmetric session keys, ensuring the confidentiality and integrity of the information. Additionally, a message authentication code (MAC) is added before data transmission to verify the integrity of the information and protect it from tampering.

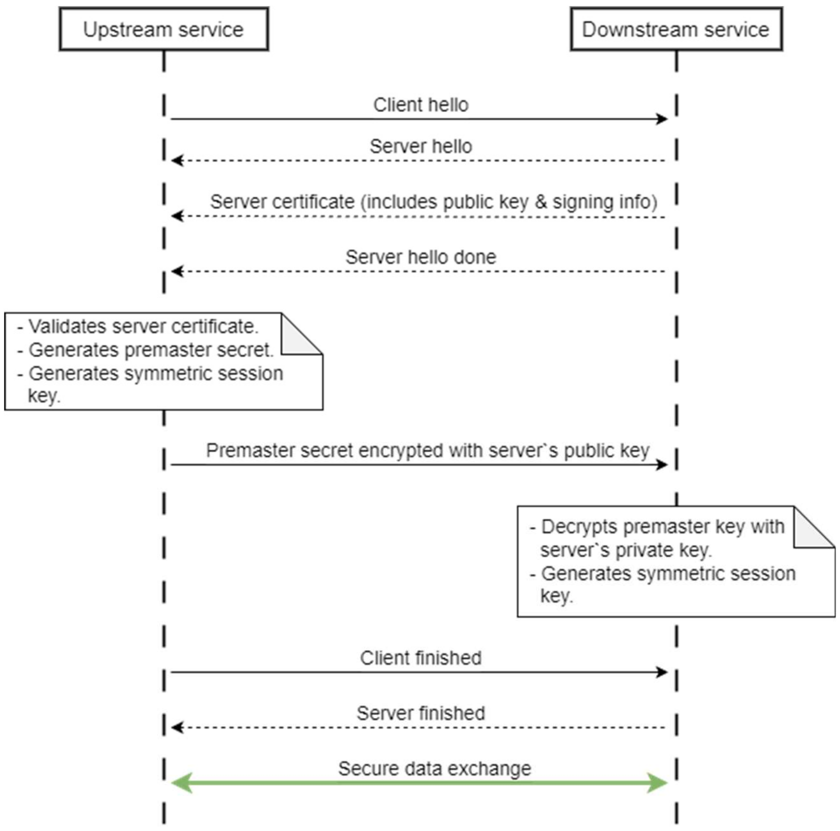

Upstream service
Downstream service
Client hello
Server hello
Server certificate (includes public key & signing info)
Server hello done
- Validates server certificate.
- Generates premaster secret.
- Generates symmetric session key.
Premaster secret encrypted with server`s public key
- Decrypts premaster key with server`s private key.
- Generates symmetric session key.
Client finished
Server finished
Secure data exchange

Figure 6: TLS “handshake” sequence diagram

TLS protects microservices from various security threats, including man-in-the-middle (MITM) attacks. In such attacks, an attacker intercepts communication between two endpoints to eavesdrop on or alter the transmitted data. Implementing TLS significantly reduces the risk of these attacks, as the encryption provided by TLS makes it difficult for attackers to decrypt intercepted data [12, 27].

The integration of TLS in MSA is typically implemented through HTTPS, which incorporates TLS encryption to secure HTTP communications.

Another important aspect is the termination of TLS connections. When using a proxy server, it is essential to understand that TLS protection is point-to-point, meaning the security provided by TLS ends at the proxy server. This requires establishing a new TLS connection between the proxy server and the microservice. This process, known as TLS bridging, can expose data to potential risks if the proxy server is compromised. An alternative approach is TLS tunneling, where a secure tunnel is established directly between microservices, ensuring the data remains encrypted and inaccessible to intermediary servers. TLS tunneling provides an additional layer of security by protecting the data throughout its journey from one microservice to another [28]. A schematic illustration showing the difference between TLS bridging and TLS tunneling is presented in Fig. 7.

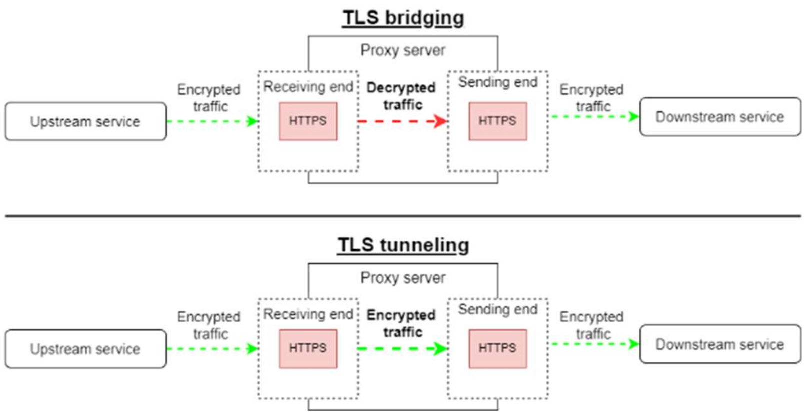


Figure 7: Comparison of TLS bridging and TLS tunneling

Mutual Transport Layer Security (mTLS), unlike one-way TLS, where authentication is verified only by the upstream service, ensures mutual authentication of both parties, guaranteeing that both the upstream and downstream services verify each other's authenticity [12, 28]. The procedure begins with the downstream service providing its certificate to the upstream service, verifying its validity and authenticity. Upon successful verification, the upstream service provides its certificate for similar validation by the downstream service. These additional steps in the handshake process are illustrated in Fig. 8. Communication can proceed only after both parties' certificates have been authenticated. This two-way authentication is particularly valuable in environments where services are distributed across different cloud infrastructures or on-premises data centers [29], mainly when communication occurs over potentially insecure networks, as is standard in hybrid or multi-cloud models.

TLS is inherently designed to protect against MITM attacks by securing communication channels through encryption and ensuring data integrity between services. However, mTLS enhances security by providing additional protection against threats targeted by TLS and attacks such as IP spoofing, where attackers attempt to impersonate trusted entities, and denial-of-service attacks aimed at disrupting service availability [29].

Additionally, compliance with industry standards and regulations, such as GDPR, PCI DSS, HIPAA, and others, often mandates using mTLS to secure data in transit, highlighting the critical importance of this protocol in protecting clients' sensitive information.

3.3. Service mesh

An additional method for enhancing the security of communication between services in MSA is implementing a service mesh—a specialized infrastructure layer that manages and secures communication between microservices. A service mesh allows for centralized management of security policies, traffic monitoring, and other critical aspects of service interactions, significantly improving the system's overall security.

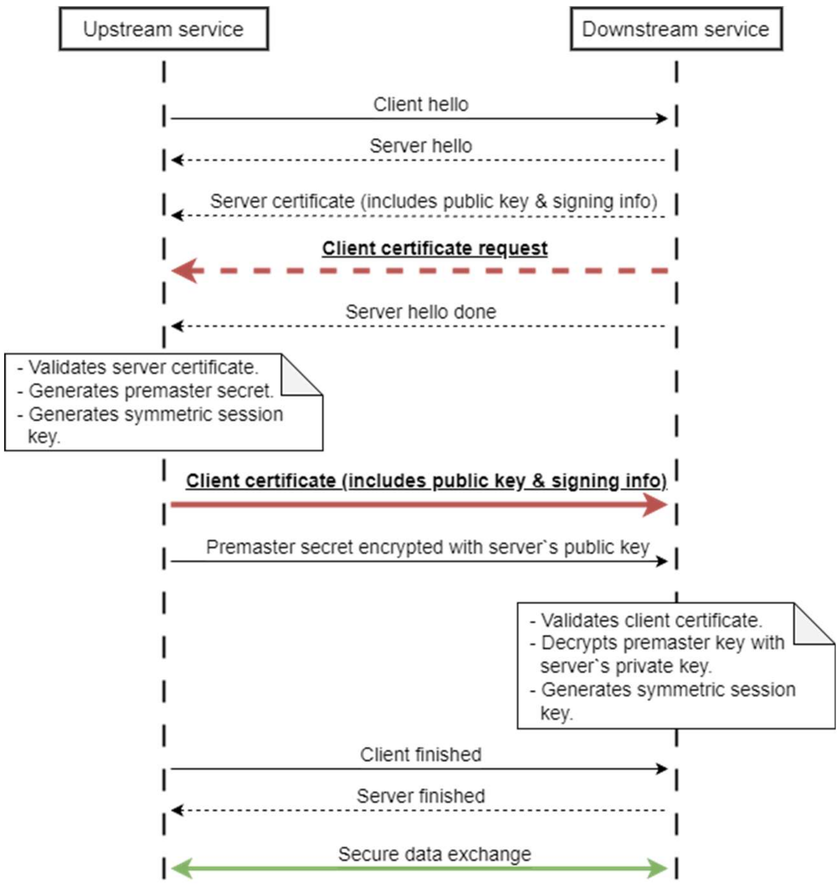
The architecture of a service mesh consists of two primary components: the control plane and the data plane.

The control plane is responsible for managing and configuring the mesh's operation, including implementing security policies and distributing configuration data to the data plane [30]. At this level, it handles certificate management, service discovery, telemetry data aggregation, and more [31, 32].

The data plane, in turn, is composed of sidecar proxies deployed alongside each microservice. These proxies intercept all incoming and outgoing traffic, enforce security policies, and ensure secure communications. This mechanism guarantees consistent communication management between microservices, regardless of the programming languages or frameworks being used [33].

Fig. 9 shows a schematic representation of the service mesh architecture, including the control plane and data plane components.

A service mesh separates the communication layer from the application's business logic, allowing for centralized management of security measures during interactions between services.



Upstream service
Downstream service
Client hello
Server hello
Server certificate (includes public key & signing info)
Client certificate request
Server hello done
- Validates server certificate.
- Generates premaster secret.
- Generates symmetric session key.
Client certificate (includes public key & signing info)
Premaster secret encrypted with server`s public key
- Validates client certificate.
- Decrypts premaster key with server`s private key.
- Generates symmetric session key.
Client finished
Server finished
Secure data exchange

Figure 8: mTLS "handshake" sequence diagram

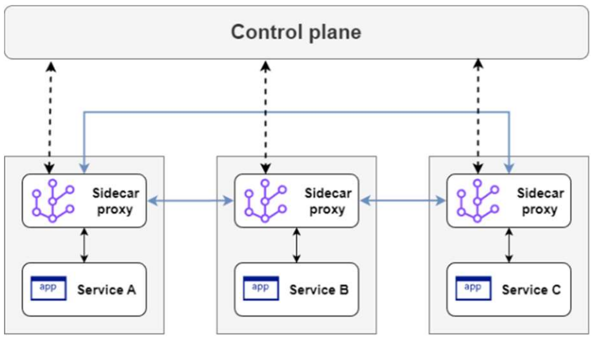


Figure 9: Service mesh architecture diagram

One critical function of a service mesh for ensuring secure communication is the use of mTLS. This is achieved by automatically issuing and managing short-term certificates for each service, which secure the connections [30]. The control plane automates the distribution and renewal of these certificates, maintaining a high level of security without requiring developers' manual intervention [31].

Additionally, a service mesh supports fine-grained authorization policies, which define which services can interact with each other and under what conditions, adding an extra layer of security [33]. This functionality is essential in large-scale deployments, where microservices may have varying levels of sensitivity and access requirements [32].

Traffic management in a service mesh creates a controlled environment for deploying and testing new service versions. With advanced capabilities such as traffic splitting, request mirroring, and canary deployments, a service mesh allows organizations to gradually roll out updates with minimal impact on the production environment. This approach enables the rapid detection and resolution of potential vulnerabilities before they can affect the entire system, significantly reducing the risk of security breaches due to new or insufficiently tested code.

The traffic splitting technology enables the convenient distribution of incoming requests between different versions of a service, facilitating controlled and gradual deployment of updates. This, in turn, minimizes the impact of changes and reduces the risk of introducing errors into the production environment [34].

Request mirroring allows developers to duplicate traffic to a test or monitoring service without affecting the main request flow. This feature is precious for analyzing service behavior under actual traffic conditions, enabling the early detection of potential issues [34].

Canary deployments add another layer of safety during testing by directing a limited portion of traffic to the updated service version. At the same time, the majority of users continue to interact with the stable version [32].

Service meshes provide capabilities for collecting and analyzing key metrics such as latency, error rates, and resource usage, contributing to a comprehensive system performance assessment [34]. Distributed tracing serves as a tool for optimizing MSA by enabling the tracking of the complete execution path of requests across numerous services. This allows for identifying bottlenecks in the system's operation and provides valuable insights for further optimization [32].

Service meshes also enable centralized logging from all microservices, consolidating them into a unified monitoring system. This simplifies the diagnostic process, facilitating the rapid detection of errors and precise tracking of the sequence of events leading to failures [31].

Advanced observability and monitoring capabilities are crucial for enhancing system security and stability. For instance, using telemetry data for dynamic system analysis enables administrators to detect anomalous events, architectural flaws, and performance issues in real-time, which is vital for the security and reliability of distributed systems [35].

3.4. Access control mechanisms

In the context of MSA, access control mechanisms are fundamental to ensuring the appropriate level of system security. They govern who can access resources and how within a distributed architecture. Due to the autonomous operation of each microservice, consistent and reliable authorization and authentication procedures are needed. This ensures the secure protection of sensitive information and reduces the risks of unauthorized access, which is critical for maintaining the integrity and security of the entire system.

OAuth 2.0 is a robust and widely recognized authorization framework that is particularly well-suited for ensuring security in the context of MSA. Originally designed to allow users to grant third-party applications limited access to their resources without exposing their credentials, OAuth 2.0 has evolved into the de facto standard for managing access in distributed systems [19]. In MSA, where decentralization is a crucial aspect, the ability of OAuth 2.0 to delegate authorization functions to a central server is precious. This delegation ensures that individual microservices do not need to manage user credentials directly, reducing the risk of security breaches and simplifying operational processes [36].

The OAuth 2.0 framework involves several key components, each performing a specific function in the authorization process: the resource owner, the client, the authorization server, and the server. The resource owner is typically the user with the right to grant access to their resources. The client is the application that requests access to these resources on behalf of the user. The authorization server authenticates the user and issues access tokens to the client. The resource server stores the protected resources and uses the issued access tokens to verify access rights and make decisions about granting access [37, 38].

OAuth 2.0 offers a range of authorization flows, each designed to meet applications' specific needs and provide an appropriate level of security. Understanding the principles and conditions for applying each flow is crucial for ensuring robust protection and optimal system performance.

The Authorization Code Grant (Fig. 10) is the most common method, particularly well-suited for web and mobile applications that securely manage client secrets. The process begins with the user (resource owner) granting access to the application and being redirected to the authorization server. After successful authentication and granting the necessary permissions, the authorization server issues an authorization code. This code is then used to obtain an access token, allowing the application to access resources on the server. This method is highly secure because the exchange of client credentials and the authorization code occurs over secure channels, significantly reducing the risk of data interception [36, 38].

The Implicit Grant (Fig. 11) is primarily used in single-page applications (SPAs) or client-side applications that cannot securely store client secrets. After the user is authenticated and grants the necessary permissions, the authorization server directly issues an access token to the client, which is passed through a URL redirect [36, 39]. Despite the simplicity of this method, its security is limited due to the token being transmitted in plain text via the URL, which restricts its use to applications with lower security requirements.

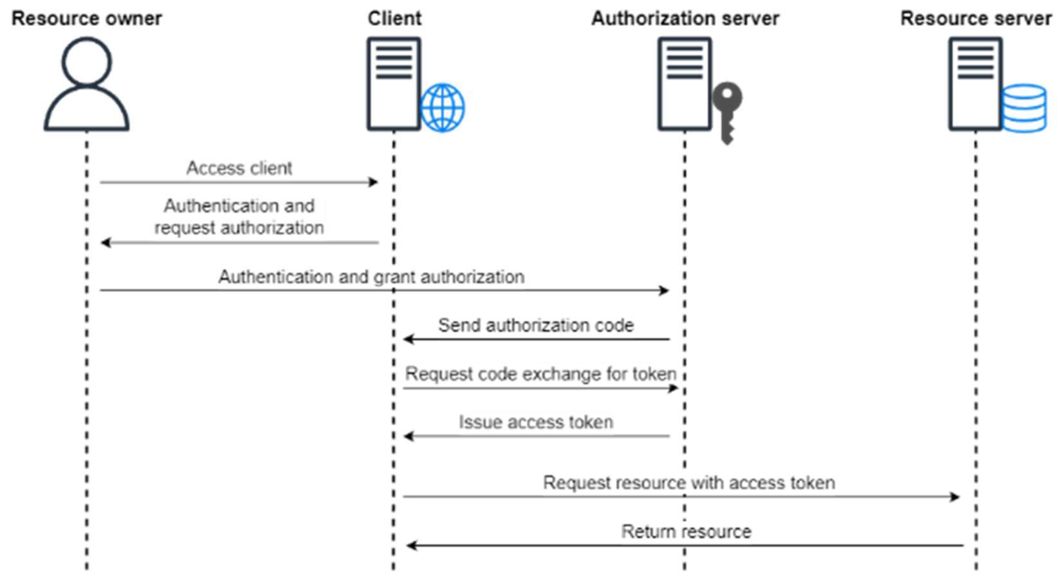

Resource owner
Client
Authorization server
Resource server
Access client
Authentication and request authorization
Authentication and grant authorization
Send authorization code
Request code exchange for token
Issue access token
Request resource with access token
Return resource

Figure 10: Authorisation code grant flow

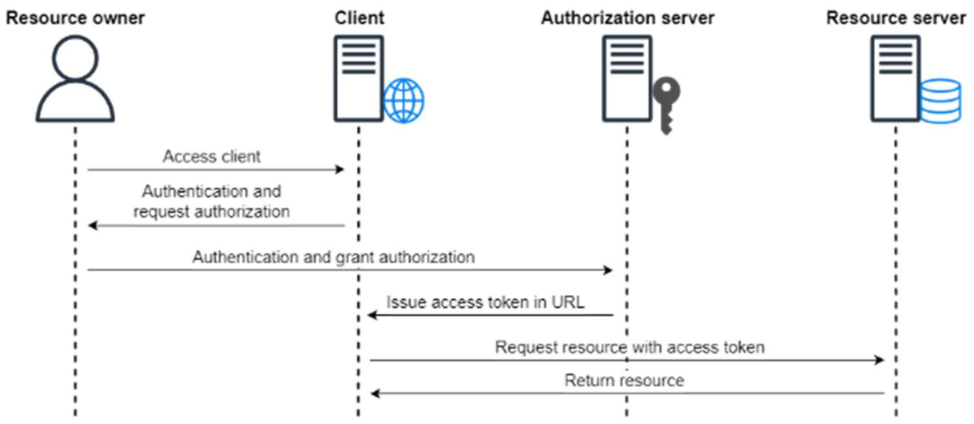

Resource owner
Client
Authorization server
Resource server
Access client
Authentication and request authorization
Authentication and grant authorization
Issue access token in URL
Request resource with access token
Return resource

Figure 11: Implicit grant flow

The Resource Owner Password Credentials Grant (Fig. 12) involves the user directly providing their credentials, such as a username and password, to the application, which then sends them to the authorization server to obtain an access token. This approach is considered outdated and less secure because the user's credentials could be compromised. As a result, it is typically used only in situations where the application is highly trusted and the risk of information leakage is minimal [38, 39].

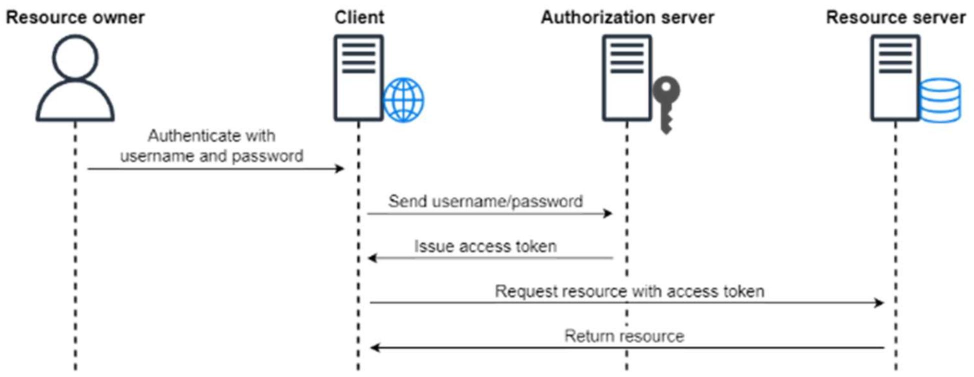


Figure 12: Resource owner password credentials grant flow

The Client Credentials Grant (Fig. 13) is designed to facilitate secure data exchange between services without user involvement. In this method, the application uses its client ID and secret key to authenticate with the authorization server and obtain an access token. This approach is particularly suitable for backend services or APIs that require secure communication without user interaction [38, 39].

In OAuth 2.0, access tokens play a significant role, representing the permissions granted by the resource owner. These tokens are designed explicitly with a short lifespan to minimize the risk of misuse in case they are stolen [40]. To ensure continuous and convenient user access without repeated authentication, OAuth 2.0 supports using refresh tokens. These long-lived tokens allow the client to obtain new access tokens as needed, providing uninterrupted access while maintaining short access token lifespans to enhance security [38, 40]. Secure storage and transmission of refresh tokens are critically important due to their extended lifespan, making it essential to prevent unauthorized access.

Access and refresh tokens are often implemented in JSON Web Tokens (JWT) format. The JWT format is compact and self-contained, directly containing all the necessary information within the token [27]. JWTs are digitally signed, which ensures their integrity and authenticity, providing robust protection against tampering or unauthorized access. This feature is a critical aspect of security, ensuring access and refresh tokens are protected throughout their lifecycle [40].

OAuth 2.0 offers several advantages that align well with the requirements of MSA.

The first advantage is OAuth 2.0's ability to scale across multiple services, providing a unified and standardized authorization mechanism that simplifies management as the system grows [32]. This scalability is crucial in microservices, where new services are frequently added, and the complexity of managing access control increases accordingly.

Additionally, using JWT in OAuth 2.0 reduces latency and complexity by allowing each service to independently verify tokens without contacting the authorization server. In centralized authorization models, services must communicate with a central authorization server to validate tokens, which can introduce delays and create bottlenecks. With JWT, all the necessary information for authorization is embedded within the token, enabling services to verify it locally using the authorization server's public key. This decentralized verification enhances performance and scalability, making it ideal for MSA, where services are designed to operate autonomously [27].

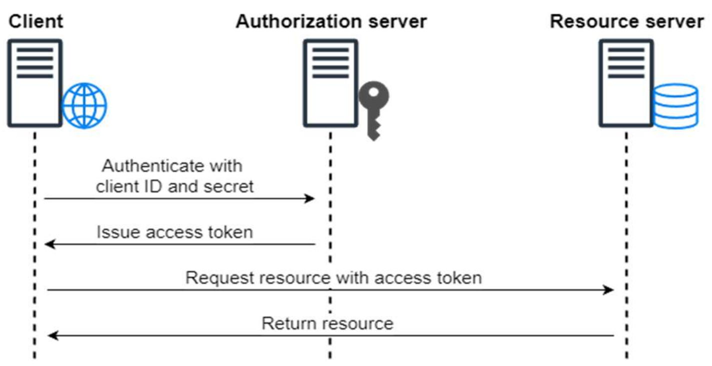


Figure 13: Client Credentials Grant flow

Another key advantage is the separation of authentication from application logic. OAuth 2.0 allows microservices to delegate the authentication process to the authorization server, which can then issue tokens used for authorization across all services [37]. This separation of concerns means developers can focus on building service-specific functionality without worrying about implementing complex authentication and authorization logic. It also simplifies the updating and managing of security policies from a central point, ensuring their consistent application throughout the system [36].

Additionally, OAuth 2.0 supports flexible access management through “scopes”, which define the level of access granted to a client. “scopes” allow for fine-tuning which resources can be accessed and what operations can be performed, which is particularly useful in microservices, where different services may have varying access and security requirements.

Implementing OAuth 2.0 in MSA requires strict adherence to several vital practices to ensure security and proper system functionality.

One critical practice is the regular rotation of secrets used to sign tokens. This strategy significantly reduces the risk of token forgery in case a secret is compromised. It ensures that outdated tokens cannot be reused, which is crucial for maintaining the integrity of the system [40].

Another practical approach is setting a short lifespan for access tokens. Limiting the duration of these tokens significantly reduces the likelihood that an attacker can exploit a stolen token. Using short-lived tokens in combination with refresh tokens achieves a balance between enhanced security and user convenience [37].

Access tokens require strict protection to prevent unauthorized access and misuse. To safeguard these tokens from interception by malicious actors, it is crucial to transmit them only over secure channels, such as HTTPS [40].

Monitoring token activity further enhances security. Implementing systems that can detect unusual token activity, such as repeated failed token validations or token usage from unexpected locations, allows for the rapid identification and response to potential security breaches [38].

OAuth 2.0’s reliance on a centralized authorization server introduces the risk of creating a single point of failure. If the authorization server experiences downtime or becomes unavailable, it can significantly disrupt the system’s ability to perform authorization operations. To mitigate this risk, it is crucial to implement redundancy and failover mechanisms for the authorization server [38]. Additionally, employing load-balancing strategies can evenly distribute traffic across multiple instances of the authorization server, ensuring that no single instance becomes a bottleneck or point of failure.

Finally, integrating OAuth 2.0 with additional access control mechanisms, such as role-based access control (RBAC) or attribute-based access control (ABAC), significantly strengthens the security architecture of microservices. For instance, RBAC can assign specific roles and permissions within individual microservices, while OAuth 2.0 ensures that these permissions are consistently enforced across the entire system [27, 40]. This integration creates a cohesive security system aligning with the organization’s policies.

OpenID Connect (OIDC) is an identity layer built on top of the OAuth 2.0 protocol, providing a standardized solution for simplifying user authentication across numerous services and applications. While OAuth 2.0 primarily handles authorization by allowing third-party clients to access resources on behalf of a user, OIDC extends this model by incorporating authentication capabilities. This ensures reliable verification of user identities and facilitates seamless integration of authentication processes into modern digital ecosystems [19, 32].

OIDC achieves its goals by leveraging familiar OAuth 2.0 authorization methods, such as the Authorization Code Grant, to obtain additional identity information through ID tokens. These ID tokens, typically represented in the JWT format, contain critical data about the authenticated user. They often include user identification details, information about the authentication events, and other attributes such as the user’s email address or profile information [32].

The user authentication process using OIDC begins with redirecting the user to an OpenID Provider (OP), an OAuth 2.0 authorization server enhanced with OIDC support. After successful identification, the OP issues an ID token and a standard access token. The client application and services can use the ID token to verify the user’s identity, access specific fields of its profile, and, if necessary, retrieve additional information from the UserInfo endpoint provided by the OP [36].

3.5. API gateway

The API gateway is a component in MSA that serves as a single entry point for all client requests directed toward microservices. This architectural concept is designed to optimize the interaction of various types of clients with microservices by providing proper management and routing of requests while simultaneously hiding the complexity of the internal system from the client [37]. As a central hub, the API gateway simplifies client interactions with different services. Instead of clients sending multiple requests to various services, the API gateway allows them to combine these into a single request, which is then processed and routed to the appropriate services [41].

In addition to its role in traffic management, the API gateway enhances microservices' security. This technology centralizes critical security functions such as authentication, authorization, and traffic validation (Fig. 14), essential for protecting microservices from unauthorized access and potential threats [42]. By centralizing security operations, the API gateway ensures consistent enforcement of security policies across all microservices, reducing the risk of vulnerabilities arising from inconsistencies in security implementation at the individual microservice level [38].

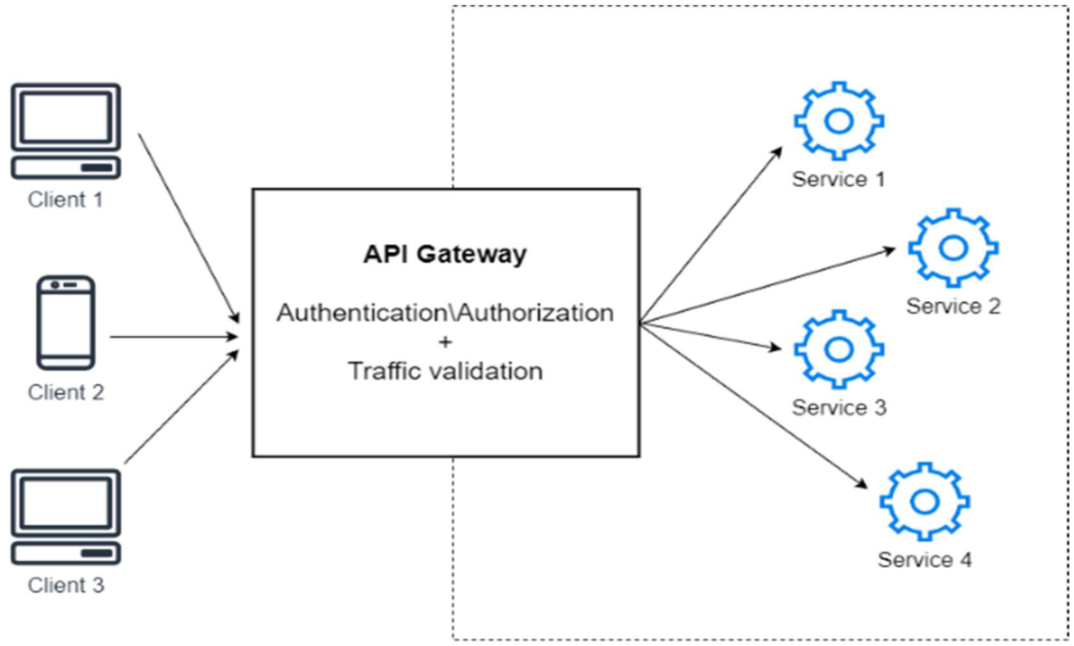


Figure 14: Centralised security management in MSA through API gateway

Centralizing authentication and authorization within the API gateway is a crucial security strategy in MSA. This approach significantly reduces the risk of security breaches by ensuring each request is authenticated and authorized before it reaches the internal services [37].

One of the most practical methods for managing authentication and authorization in an API gateway is implementing the OAuth 2.0 protocol. In this process, the API gateway acts as a client within the OAuth 2.0 framework, interacting with the authorization server to authenticate requests and obtain access tokens. The API gateway then validates these tokens before forwarding the request to the appropriate microservice, ensuring that only authorized requests are processed [28].

Ensuring edge security in MSA is crucial for protecting the external perimeter where interactions between external clients and internal services occur. The API gateway plays a vital role in this process by acting as the security layer for all north-south traffic, which refers to the data exchange between external clients and internal microservices [28].

The first function that an API gateway can perform is TLS termination. This process involves decrypting incoming TLS traffic at the API gateway level before passing it to the microservices. Handling decryption at the API gateway reduces the computational load associated with TLS decryption for individual microservices [41]. The API gateway can also inspect the traffic for potential threats, ensuring that only secure information passes through the system [28]. TLS termination also simplifies certificate and encryption management by centralizing these tasks at the API gateway level rather than distributing them across multiple services.

In addition to TLS termination, an API gateway can implement rate-limiting and throttling mechanisms to protect against denial-of-service attacks and other forms of abuse. These mechanisms limit the number of requests that can be made within a specific time frame, preventing the system from being overwhelmed by excessive traffic from any single client [36, 43]. Rate limiting enhances security and helps maintain the performance and availability of services, ensuring that malicious or overly demanding clients do not exhaust resources.

An API gateway can also implement signature-based protection, which helps identify and block requests that match known attack patterns [43]. By recognizing and responding to these patterns, the API gateway adds a critical layer of defense, intercepting potentially malicious traffic before it can reach the backend microservices.

Furthermore, the API gateway is key in isolating internal services from external client applications. By acting as an intermediary, the API gateway ensures that external clients do not interact directly with the microservices, significantly reducing the attack surface and limiting potential threats to access to internal services [12].

The API gateway is a powerful tool for managing and securing MSA, but it also introduces several challenges that organizations must consider to ensure a reliable and scalable system. One of the most significant risks associated with using an API gateway is its potential to become a single point of failure. If the API gateway fails, it can disrupt access to all microservices, effectively leading to a complete system outage [12]. Strategies such as horizontal scaling and redundancy should be implemented to minimize this risk. Deploying multiple instances of the API gateway across different geographic locations ensures service continuity in case one instance fails. Using load balancing to distribute incoming requests among these instances evenly can enhance system reliability and prevent bottlenecks [28].

It is important to note that while the API gateway provides essential security functions, it is not a universal solution for all threats. Organizations should implement additional protective measures at the microservice level, such as fine-grained access controls, detailed logging, and anomaly detection systems to enhance security. Moreover, adopting a defense-in-depth strategy, where security is enforced at multiple levels of the architecture, can offer more comprehensive protection against sophisticated attacks [44]. This multi-layered approach increases the likelihood that other security mechanisms can detect and respond to the threat even if the API gateway is bypassed.

3.6. Securing data at rest

Data at rest refers to information stored on any physical or digital medium, such as databases, files, backup systems, and other storage solutions. In the context of MSA, protecting such data becomes more challenging due to the distributed nature of the system and the placement of data across various services and storage locations. MSA developers need to implement a comprehensive security strategy to ensure the confidentiality of data at rest and protect it from unauthorized access, breaches, and other threats. This strategy may include data encryption, strict access controls, data masking, immutable storage, tokenization, and other practices.

Data encryption is a crucial technique for protecting information at rest. This technique transforms data into an unreadable format using encryption algorithms, such as Advanced Encryption Standard (AES) for large volumes of data and Rivest-Shamir-Adleman (RSA) for secure key exchange [45, 46]. These methods are essential for complying with regulations like GDPR, HIPAA, and PCI DSS and should be a mandatory element of an organization's security protocols. Cloud service providers also support encryption, offering automated solutions that simplify this process and ensure data security regardless of volume or processing speed [45]. Additionally, disk-level encryption can be implemented at the operating system level, such as dm-crypt in Linux, which provides encryption and decryption of data as it is written to or read from storage devices [47].

Access controls provide an additional layer of protection for data at rest. Implementing RBAC and ABAC restricts access to data only to authorized users based on their roles within the organization or specific attributes. Only individuals with the necessary permissions can access sensitive information [45, 48]. Such an approach is crucial in maintaining the confidentiality and integrity of data, especially in distributed environments where data is accessible to multiple users and systems.

Data masking protects sensitive information in MSA, particularly in non-production environments such as development and testing. This method permanently replaces actual data with fictitious but plausible values, protecting sensitive information outside secure production environments. The importance of this approach lies in its ability to maintain the functionality of the data while reducing the risk of unauthorized access or data breaches. This is particularly relevant when data is used for testing, analytics, or similar tasks [45, 46]. For example, a credit card number might be replaced with a similar string of characters that mimics the original data format but has no real value.

Immutable storage involves systems where once-written data cannot be altered or deleted. This approach allows organizations to protect their systems from threats such as ransomware, unauthorized changes, and accidental deletions [45].

Implementing immutable storage can include using write-once-read-many (WORM) mode or blockchain technology to create an unalterable record of transactions. These methods ensure that the data remains unchanged even in the event of malicious access. Additionally, immutable storage aids organizations in meeting regulatory requirements for data preservation and integrity, providing a reliable way to prove that data has not been altered since it was initially stored [48].

Tokenization is another method of protecting sensitive data in microservices, which involves replacing confidential information with non-sensitive equivalents (tokens) that hold no intrinsic value for attackers [45]. The original data is stored securely, and only those with access to the tokenization system can map the tokens back to the original information. Unlike data masking, tokenization is suitable for use in production environments, where maintaining the integrity and security of sensitive data is critical.

Secure storage, regular rotation, and controlled access to cryptographic keys are critical elements of data protection [32, 45]. Using dedicated key vaults or specialized hardware helps prevent key compromise along with the data. Tools like Vault from HashiCorp provide robust key management, granting access only to authorized services [32].

Monitoring and auditing play a significant role in detecting and responding to security incidents related to stored data. Continuous monitoring of storage system activity and access log analysis enables quick identification of unauthorized access attempts [48].

4. Conclusions

This study comprehensively analyzed the challenges and specific practices associated with ensuring security in MSA. This modern architectural approach offers significant advantages in scalability, flexibility, and the autonomy of service development and deployment. It was found that the implementation and transition to MSA are accompanied by substantial security risks that require careful management to maintain the integrity and confidentiality of distributed systems.

One of the primary challenges identified in this study is the expanded attack surface inherent in MSA. The presence of numerous independent services, each functioning as a separate network endpoint, significantly increases the number of potential entry points for attackers, making implementing robust security measures critically important. To mitigate this risk, the study suggests using service isolation strategies, such as virtualization using VMs, containerization, and micro-segmentation. These strategies allow potential threats to be contained within isolated environments, significantly reducing the risk of attacks spreading between services and providing more reliable system protection. Additionally, it was found that the API gateway plays a crucial role in this context, serving as the first line of defense for all north-south traffic.

Ensuring the security of communication between services is another challenge addressed in this study. The research emphasizes the necessity of securing communication channels using TLS and mTLS protocols, which protect against MITM attacks and ensure that only authorized services interact with each other. A service mesh architecture is also recommended as a robust strategy for managing and securing service-to-service communications. This architecture provides a specialized infrastructure layer that simplifies the consistent application of security policies and guarantees secure interactions between microservices.

Furthermore, this study's analysis of data management challenges in a distributed environment highlights the importance of encrypting data at rest and in transit to protect against threats sufficiently. Implementing strict access control policies like RBAC and ABAC prevents unauthorized access. The study also emphasizes the need for additional data protection methods for data at rest, including data masking, immutable storage, and tokenization. When combined with encryption and access controls, these approaches significantly enhance data security in distributed systems.

The study suggests using the OAuth 2.0 and OpenID Connect frameworks for authentication and authorization. OAuth 2.0 provides a scalable solution for delegating authorization across multiple services, while OpenID Connect adds a layer of user authentication, simplifying identity management in a distributed system. The API gateway enhances these frameworks by centralizing authentication and authorization processes, ensuring that only authenticated users and authorized requests gain access to microservices, thereby reducing the risk of security breaches and simplifying policy management.

In conclusion, while MSA offers significant advantages for modern software development, it demands a carefully coordinated security strategy. Successful implementation of MSA involves not only the integration of cutting-edge technologies but also the adoption of comprehensive security practices specifically designed to address the unique challenges associated with managing distributed systems. As MSA continues to evolve, so do various cyber threats, making it essential for future research to focus on developing innovative tools and methodologies to enhance security in these systems. This will allow organizations to fully leverage the benefits of microservices while minimizing risks to security and reliability.

References

[1] H. Hulak, et al., Dynamic model of guarantee capacity and cyber security management in the critical automated systems, in: 2nd International Conference on Conflict Management in Global Information Networks, vol. 3530 (2022) 102-111.

[2] X. Liu, et al., Research on Microservice Architecture: A Tertiary Study, SSRN Electron. J. (2022). doi: 10.2139/ssrn.4204345.

[3] M. Waseem, P. Liang, M. Shahin, A Systematic Mapping Study on Microservices Architecture in DevOps, J. Syst. Softw. 170 (2020) 110798. doi: 10.1016/j.jss.2020.110798.

[4] F. Auer, et al., From monolithic systems to Microservices: An assessment framework, Inf. Softw. Technol. 137 (2021) 106600. doi: 10.1016/j.infsof. 2021.106600.

[5] P. D. Francesco, I. Malavolta, P. Lago, Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption, in: 2017 IEEE International Conference on Software Architecture (ICSA), IEEE (2017). doi: 10.1109/icsa.2017.24.

[6] G. Blinowski, A. Ojdowska, A. Przybylek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, IEEE Access 10 (2022) 20357–20374. doi: 10.1109/access.2022.3152803.

[7] N. Salaheddin Elgheriani, N. D. Ali Salem Ahme, Microservices vs. Monolithic Architectures [The Differential Structure Between Two Architectures], Minar Int. J. Appl. Sci. Technol. 4(3) (2022) 500–514. doi: 10.47832/2717-8234.12.47.

[8] Y. Dreis, et al., Model to Formation Data Base of Internal Parameters for Assessing the Status of the State Secret Protection, in: Workshop on Cybersecurity Providing in Information and Telecommunication Systems, CPITS, vol. 3654 (2024) 277–289.

[9] D. Berestov, et al., Analysis of Features and prospects of Application of Dynamic Iterative Assessment of Information Security Risks, in: Workshop on Cybersecurity Providing in Information and Telecommunication Systems, CPITS, vol. 2923 (2021) 329–335.

[10] S. Shevchenko, et al., Information Security Risk Management using Cognitive Modeling, in: Workshop on Cybersecurity Providing in Information and Telecommunication Systems II, CPITS-II, vol. 3550 (2023) 297–305.

[11] P. Nkomo, M. Coetzee, Software Development Activities for Secure Microservices, in: Computational Science and Its Applications – ICCSA 2019, Springer International Publishing, Cham (2019) 573–585. doi: 10.1007/978-3-030-24308-1_46.

[12] M. Matias, et al., Enhancing Effectiveness and Security in Microservices Architecture, Procedia Comput. Sci. 239 (2024) 2260–2269. doi: 10.1016/j.procs.2024.06.417.

[13] P. Haindl, P. Kochberger, M. Sveggen, A Systematic Literature Review of Inter-Service Security Threats and Mitigation Strategies in Microservice

Architectures, IEEE Access (2024) 1. doi: 10.1109/access.2024.3406500.

[14] J. Kazanavičius, D. Mažeika, Evaluation of Microservice Communication While Decomposing Monoliths, Comput. Inform. 42(1) (2023) 1–36. doi: 10.31577/cai_2023_1_1.

[15] L. D. S. B. Weerasinghe, I. Perera, Evaluating the Inter-Service Communication on Microservice Architecture, in 2022 7th International Conference on Information Technology Research (ICITR), IEEE (2022). doi: 10.1109/icitr57877.2022.9992918.

[16] Microservices Security: Challenges & 7 Ways to Secure Microservices. URL: https://www.aquasec.com/cloud-native-academy/cnapp/microservices-security/

[17] D. Yu, et al., A Survey on Security Issues in Services Communication of Microservices-enabled Fog Applications, Concurr. Comput. 31(22) (2018). doi: 10.1002/cpe.4436.

[18] M. Driss, et al., Microservices in IoT Security: Current Solutions, Research Challenges, and Future Directions, Procedia Comput. Sci. 192 (2021) 2385–2395. doi: 10.1016/j.procs.2021.09.007.

[19] M. G. de Almeida, E. D. Canedo, Authentication and Authorization in Microservices Architecture: A Systematic Literature Review, Appl. Sci. 12(6) (2022) 3023. doi: 10.3390/app12063023.

[20] G. Liu, et al., Microservices: Architecture, Container, and Challenges, in: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE (2020). doi: 10.1109/qrs-c51114.2020.00107.

[21] O. Bentaleb, et al., Containerization Technologies: Taxonomies, Applications and Challenges, J. Supercomput. (2021). doi: 10.1007/s11227-021-03914-1.

[22] What Are Microservices? URL: https://microservices.io/index.html

[23] E. Casalicchio, S. Iannucci, The State-of-the-Art in Container Technologies: Application, Orchestration and Security, Concurr. Comput. 32(17) (2020). doi: 10.1002/cpe.5668.

[24] N. F. Syed, et al., Zero Trust Architecture (ZTA): A Comprehensive Survey, IEEE Access 1 (2022). doi: 10.1109/access.2022.3174679.

[25] D. Klein, Micro-Segmentation: Securing Complex Cloud Environments, Netw. Secur. 2019(3) (2019) 6–10. doi: 10.1016/s1353-4858(19)30034-0.

[26] R. Chandramouli, Guide to Secure Enterprise Network Landscape, National Institute of Standards and Technology, Gaithersburg, MD (2022). doi: 10.6028/nist.sp.800-215.

[27] N. Mateus-Coelho, M. Cruz-Cunha, L. G. Ferreira, Security in Microservices Architectures, Procedia Comput. Sci. 181 (2021) 1225–1236. doi: 10.1016/j.procs.2021.01.320.

[28] P. Siriwardena, N. Dias, Microservices Security in Action, Manning Publications Company (2020).

[29] What is mTLS and How to Implement it with Istio. URL: https://imesh.ai/blog/what-is-mtls-and-how-to-implement-it-with-istio

[30] R. Chandramouli, Z. Butcher, Building Secure Microservices-based Applications using Service-Mesh Architecture, National Institute of Standards and Technology, Gaithersburg, MD (2020). doi: 10.6028/nist.sp.800-204a.

[31] R. Chandramouli, Implementation of DevSecOps for a Microservices-based Application with Service Mesh, National Institute of Standards and Technology (2022). doi: 10.6028/nist.sp.800-204c.

[32] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Incorporated (2021).

[33] M. R. Saleh Sedghpour, C. Klein, J. Tordsson, An Empirical Study of Service Mesh Traffic Management Policies for Microservices, in: ICPE '22: ACM/SPEC International Conference on Performance Engineering, ACM, New York, NY, USA (2022). doi: 10.1145/3489525.3511686.

[34] What is a Service Mesh? URL: https://aws.amazon.com/what-is/service-mesh/

[35] O. V. Talaver, T. A. Vakaliuk, Telemetry to Solve Dynamic Analysis of a Distributed System, J. Edge Comput. (2024). doi: 10.55056/jec.728.

[36] Tai Ramirez, Wai Yan Elsa, A Framework to Build Secure Microservice Architecture, Open Access Theses & Dissertations, 3857 (2023).

[37] C. Richardson, Microservices Patterns: With Examples in Java, Manning Publications Co. LLC (2018).

[38] I. Bazeniuc, A. Zgureanu, Information Security in Microservices Architectures, in: 11th International Conference on "Electronics, Communications and Computing", Technical University of Moldova (2022). doi: 10.52326/ic-ecco.2021/sec.03.

[39] OAuth 2.0 Grant Types. URL: https://docs.vmware.com/en/Single-Sign-On-for-VMware-Tanzu-Application-Service/1.16/sso/GUID-grant-types.html

[40] A. Venčkauskas, et al., Enhancing Microservices Security with Token-Based Access Control Method, Sensors 23(6) (2023) 3363. doi: 10.3390/s23063363.

[41] W. Jin, et al., Secure Edge Computing Management Based on Independent Microservices Providers for Gateway-Centric IoT Networks, IEEE Access 8 (2020) 187975–187990. doi: 10.1109/access.2020.3030297.

[42] C. Fernando, Implementing Observability for Enterprise Software Systems, in: Solution Architecture Patterns for Enterprise, Apress, Berkeley, CA (2022) 231–268. doi: 10.1007/978-1-4842-8948-8_7.

[43] API Gateway Security. URL: https://www.akamai.com/glossary/what-is-api-gateway-security

[44] M. Waseem, et al., Decision Models for Selecting Patterns and Strategies in Microservices Systems and their Evaluation by Practitioners, in: 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE (2022). doi: 10.1109/icse-seip55303.2022.9793911.

[45] P. Atri, Enhancing Big Data Security through Comprehensive Data Protection Measures: A Focus on

Securing Data at Rest and In-Transit, Int. J. Comput. Eng. 5(4) (2024) 44–55. doi: 10.47941/ijce.1920.

[46] M. A. Mohammed, F. S. Abed, A Symmetric-based Framework for Securing Cloud Data at Rest, Turk. J. Electr. Eng. & Comput. Sci. 28(1) (2020) 347–361. doi: 10.3906/elk-1902-114.

[47] L. Daoud, H. Huen, Performance Study of Software-based Encrypting Data at Rest, in: Proceedings of 37th International Conference on Computers and Their Applications, EasyChair, 82 (2022) 122–130. doi: 10.29007/1j1p.

[48] H. A. Abdulghani, et al., A Study on Security and Privacy Guidelines, Countermeasures, Threats: IoT Data at Rest Perspective, Symmetry 11(6) (2019) 774. doi: 10.3390/sym11060774.