International Journal of Scientific Research and Engineering Development-– Volume 7 Issue 3, May-June 2024
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

API Security in Microservice Architecture: Methods to Prevent Injections

Alexandre Anacleto Libanio Xavier Fernandes
Senior Associate Technology at Publicis Sapient Toronto, Canada
alexandre.fernandes@publicissapient.com

Abstract:
The paper analyzes security threats related to APIs in microservice architecture and proposes methods of their neutralization. The main attention is paid to the problem of injection, which is one of the most critical threats to the stability and security of microservice-based systems. Different types of techniques for preventing tampering, including SQL, command-line, and HTML, are discussed and strategies for preventing them are proposed, including input sanitization, use of parameterized queries, and privilege delimitation. Tools and technologies that can be used to improve API security, including access control systems and the use of secure programming patterns, are also discussed. The analysis results show that the integrated application of the proposed techniques can significantly reduce the risk of successful attacks.
Keywords: software, software, programming, IT, API, modern technologies, SQL.

Introduction

Within microservices architecture, the concept is to decompose large software complexes into many small, independent services that interact with each other using APIs. An analogy can be drawn to the division of a large-scale puzzle into smaller, manageable fragments, where each service specializes in a particular function and communicates with other elements through well-defined interfaces [1].

In turn, the importance of API security in the modern information age cannot be underestimated. APIs, or application programming interfaces, provide critical communication between different software systems, often transferring data across public networks from clients to servers. Compromising these APIs can lead to unauthorized access to personal, financial, or other sensitive information, making security incredibly important throughout the development and operation of RESTful and other types of APIs.

Vulnerabilities in APIs can lead to attackers gaining access to server systems and compromising all functionality available through the API. For example, an improperly secured API can be a target for attacks such as denial of service (DoS), degrading its performance or even disabling it. Additionally, attackers can use an API to delete data or exhaust resources by exceeding usage limits.

More sophisticated attacks can involve the introduction of malicious code, allowing unauthorized operations or even compromising the server entirely. In the context of the proliferation of microservice and serverless architectures, where every function of an enterprise application may depend on an API, securing these interfaces becomes not just important, but a necessary component of enterprise security [2].

1. Types of authorizations

Authorization at the peripheral layer. In basic scenarios, authorization is often performed exclusively at the peripheral layer, e.g., through an API gateway. This gateway allows centralized implementation of authorization mechanisms for all interacting microservices, thereby eliminating the need for individual authentication and access control at the level of each service. According to NIST guidelines, it is desirable to implement mutual authentication methods to prevent anonymous direct connections to internal services from bypassing the API gateway. However, keep in mind that centralizing all authentication decisions at the API gateway can complicate management in a complex ecosystem with multiple roles and access rules. In addition, the API gateway can become a vulnerable single decision center, which violates the principles of echelon security. Ownership of the gateway by operations teams and the lack of ability for development teams to directly make authorization changes slows down processes due to the need for additional collaboration and coordination.

Service-level authorization. Implementing service-level authorization gives each microservice the ability to independently control the execution of access policies. Consider the components of an access control system as categorized by NIST SP 800-162:

- The Policy Administration Point (PAP) provides an interface for creating, managing, testing, and debugging access rules.
- The Policy Decision Point (PDP) is responsible for computing access decisions based on the management policy being applied.
- The Policy Enforcement Point (PEP) enforces the decisions made in response to resource access requests.
- The Policy Information Point (PIP) functions as a source of data or attributes needed to evaluate policies and support the PDP in making informed decisions (Figure 1.).

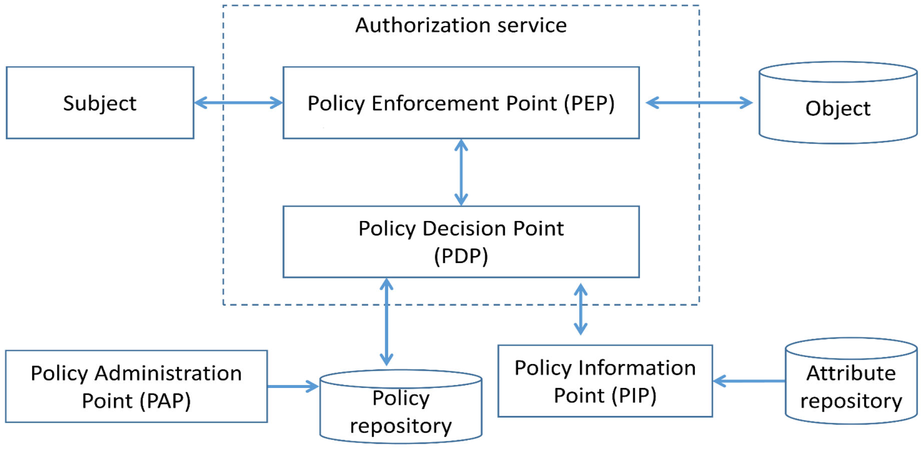
Authorization service
Subject
Policy Enforcement Point (PEP)
Object
Policy Decision Point (PDP)
Policy Administration Point (PAP)
Policy repository
Policy Information Point (PIP)
Attribute repository

Figure 1. Authorization at the service level: existing templates [3]

Decentralized pattern. The development team implements PDP and PEP directly at the microservice code level. All-access control rules and attributes required to implement the rule are defined and stored in each microservice (step 1). When a microservice receives a request along with some authorization metadata (e.g., end-user context or the ID of the requested resource), the microservice analyzes it (step 3) to generate an access control policy decision and then applies the authorization (Figure 2.).

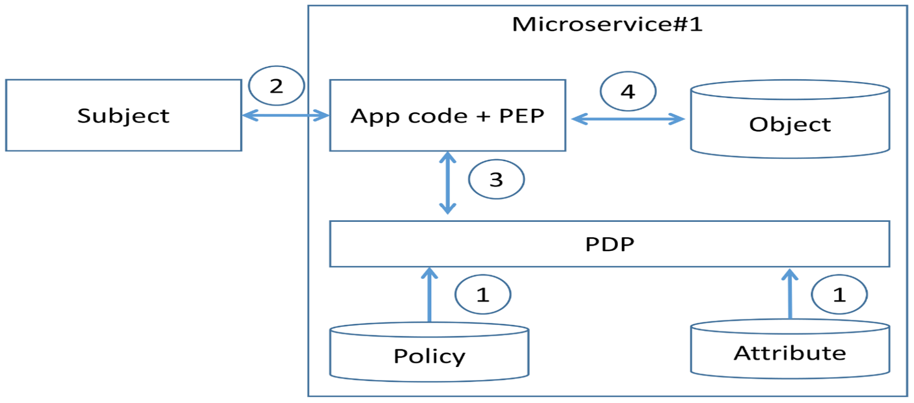

Figure 2. An example of a decentralized pattern [3]

Modern programming platforms provide developers with effective tools for implementing authorization at the microservice level. For example, the Spring Security framework provides facilities to implement authorization checking using details extracted from JWT at the resource server level, which allows strict control over access to functionality.

However, implementing authorization at the source code level entails the need for regular code updates to reflect any changes in authorization policy desired by the development team.

A centralized model with a single point of policy decision-making. This approach is characterized by access control policies being formulated, stored, and analyzed in a centralized manner. In the first step, policies are defined and managed through a policy administration point (PAP). Next, these policies and associated attributes are sent to a centralized policy decision point (PDP). When a user or system contacts the microservice (step three), the microservice initiates a network request to the PDP, which analyzes the rules and attributes associated with the request (step four). Based on the analysis, the PDP makes a judgment, which is then used by the microservice to authorize user or system actions (fifth step).

This model promotes a more robust and centralized management of access policies but also requires accurate and timely updating of policies and attributes to ensure that decisions are relevant and accurate (Figure 3.).

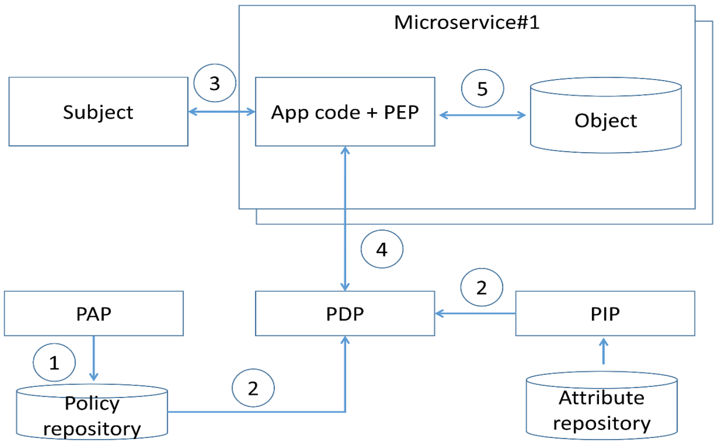

Figure 3. Centralized pattern with single policy decision point [3]

A centralized model with a built-in point of policy decision-making. In this model, access control rules are formulated centrally but evaluated and stored at the microservice level. The initial configuration of policies is done through a policy administration point (PAP). These policies along with the required attributes are passed to the embedded PDP. When a user or system initiates a request to a microservice, the microservice code contacts this embedded PDP, which, analyzing the incoming data, generates an access decision based on the existing rules and attributes (step four). Depending on this decision, the microservice then implements appropriate authorization measures (fifth step).

This model optimizes the authorization process, speeding up decision-making at the microservice level, and facilitates a more flexible adaptation to changes in security policy (Figure 4.).

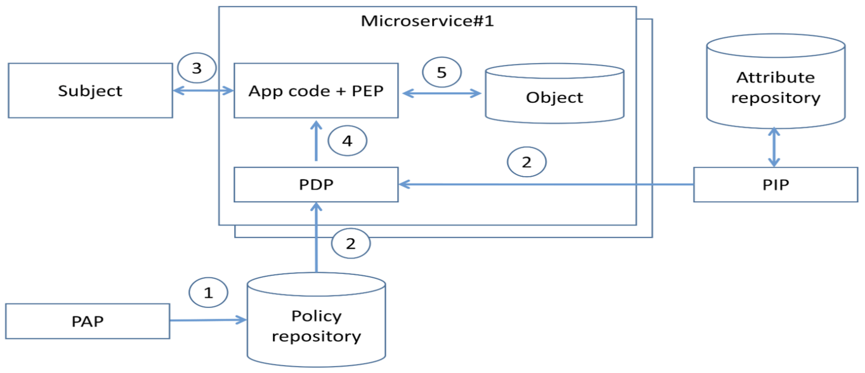

Figure 4. Centralized pattern with embedded policy decision point [3]

In a scenario where PDP is implemented as an integrated library in a microservice or as a standalone application within a service network, it is recommended to embed PDP directly into the microservice. This prevents potential communication failures and delays due to network or node issues. An embedded PDP running on the same node as the microservice typically manages authorization policies by storing them in local memory to reduce external dependencies and response times for authorization operations.

This is different from the "Centralized Single Decision Point Approach" where authorization decisions are centrally stored and managed. In the proposed model, updated authorization policies are stored directly in the microservice, which facilitates their quick update and application. However, the risks associated with caching authorization decisions, which may lead to the use of outdated rules and access control violations, should be considered [3].

2. Ways to protect microservices

Despite the rapid proliferation of microservices architecture, application security often fails to keep pace with its development. There is no one-size-fits-all way to secure microservices. It is important for developers to consider multiple aspects, such as security risks and time-to-market, in an effort to find the best match between security requirements and organizational resources.

API security is a multidimensional field that involves multiple layers of protection measures. Each layer is specifically focused on strengthening API security and strives to provide a robust defense against threats. In managing the security of microservice applications, handling access requests from external users and other services plays a key role. These requests can be managed using specialized authorization mechanisms such as the Open Policy Agent (OPA). It is also possible to use a monolithic API gateway that centrally handles all external requests to microservices. Next, let's look at the most appropriate ways to secure microservices:

1. Applying Shift-Left and DevSecOps strategies: Integrating security throughout the development stages, starting from the earliest stages, and including security experts in development teams helps prevent threats in the initial stages.
2. Develop applications with embedded security: It is important to embed multiple layers of security throughout the design, development, and deployment process by continuously testing and auditing code and infrastructure.
3. Deeply echeloned security: Multiple layers of defense should be in place to prevent attackers from accessing key assets, even if they can penetrate primary barriers.
4. Authentication and Authorization: Implement identity and access management standards and use multi-factor authentication to strengthen security.
5. API Gateway Implementation: API gateways manage all external traffic, providing a single point of authentication and defense against attacks.
6. Container Security: The principle of least privilege and security strategies should guide the deployment and management of containers to minimize risks [4].

3. API security leaks

API leaks can occur due to accidental or intentional disclosure of sensitive data through application programming interfaces. If APIs are not adequately protected, they can become leakage channels for sensitive information including personal data, financial information, and other sensitive data. These leaks can be caused by a variety of reasons, including customization errors, code vulnerabilities, and flaws in security protocols, which can lead to data breaches, identity theft, and financial losses. The existing types of API leaks include the following:

1. Data Leaks: This is the most common type of leak where personal data, passwords, and financial information are made available through APIs.
2. Access leaks: Here, unauthorized users are given access to restricted APIs, allowing them to access sensitive data.
3. Integration Leaks: This occurs when APIs are integrated with other systems without proper security measures.
4. Supply Chain Leaks: These leaks occur when third-party APIs with vulnerabilities are used in the supply chain.

5. Configuration leaks: These occur due to improper API configuration that does not meet the required security standards.

In turn, authentication of data leaks is a critical process because if detected in time, API misuse can be prevented as it authenticates the user before granting access to the data. In the following, the types of authentication will be discussed:

1. Host-based authentication: A widely used method for the Internet of Things and network devices, although it is not recommended for web technologies due to the possibility of spoofing. In this method, verification is done at the server level, and only verified users can access resources.

2. Basic Authentication: One of the simplest forms of API authentication, using the HTTP protocol to send credentials, typically transmitted as plaintext and encoded in base64. By default, this can create vulnerabilities to man-in-the-middle attacks.

3. OAuth: A modern and secure authentication method that uses tokens to verify users. OAuth allows applications to authorize users using tokens issued by the OAuth server, making it a secure and widely used authentication method in modern web applications.

4. OAuth 2.0: The OAuth 2.0 protocol, which is an extended version of the original OAuth standard, is widely used to control access to APIs. This protocol enhances security by restricting API access through the use of standard HTTP services to activate a client application. Examples of the use of such HTTP services include platforms like GitHub and Facebook, which support this protocol for user authentication without requiring direct provisioning of user credentials.

The OAuth 2.0 architecture identifies three main participants: the data owner (the user), the client application, and the API itself. The user owns the data and grants the application permission to access that data through the API. This permission is granted after authentication and authorization are provided through OAuth 2.0.

The use of OAuth 2.0 facilitates the secure transfer of user information between different platforms and devices, making it an ideal choice for authentication and authorization in web applications, mobile applications, and desktop devices. This method not only enhances data security but also provides flexibility in managing access to different applications.

5. SAML.SAML, which stands for Security Assertion Markup Language, is a standard protocol for authentication through a Single Sign-On (SSO) mechanism. This protocol allows the user's identity to be confirmed based on the credentials provided. Once the authentication process is complete and the user's identity is confirmed, the user is granted access to multiple applications or resources. The current version of the SAML protocol is SAML 2.0.

SAML 2.0 facilitates the authentication process by allowing security systems to exchange user credentials. The principle of SAML is similar to the use of an identity card: it confirms that the user is indeed who he or she claims to be, allowing the system to provide him or her with access to the necessary resources based on his or her identity and access rights [5,6].

Conclusion

The conclusion of the research on API security in microservices architecture emphasizes the importance of a comprehensive approach to hacking prevention. The problem remains relevant and requires measures to strengthen security at all levels of the system. Effective prevention is achieved through the use of modern data sanitization techniques, query parameterization, and strict privilege separation. The application of these techniques not only prevents potential threats but also improves the overall security structure of microservice architectures. In addition to technical measures, a conscious and responsible attitude of developers towards the process of creating and maintaining secure APIs plays an important role. The article emphasizes that only multi-layered security and continuous education and training in IT professionals can provide reliable protection against injection attacks in the context of constantly evolving technologies.

References

1. Collaborative Approaches to Strengthening Microservices Security. [Electronic resource] Access mode: https://partnerpens.hashnode.dev/microservices-security.– (access date 04/27/2024).

2. API security. [Electronic resource] Access mode: https://www.postman.com/api-platform/api-security/.– (access date 04/27/2024).
3. Microservices Security Cheat Sheet. [Electronic resource] Access mode: https://cheatsheetseries.owasp.org/cheatsheets/Microservices_Security_Cheat_Sheet.html.– (access date 04/27/2024).
4. Microservices Security: Fundamentals and Best Practices. [Electronic resource] Access mode: https://www.styra.com/blog/microservices-security-fundamentals-and-best-practices/.– (date accessed 04/27/2024).
5. API Security Tutorial. [Electronic resource] Access mode: https://www.wallarm.com/what/api-security-tutorial.– (access date 04/27/2024).
6. Best Practices to Secure Microservices with Spring Security. [Electronic resource] Access mode: https://www.geeksforgeeks.org/best-practices-to-secure-microservices-with-spring-security/.– (date accessed 04/27/2024).
7. API and microservice security. [Electronic resource] Access mode: https://portswigger.net/burp/vulnerability-scanner/api-security-testing/guide-to-api-microservice-security.– (date accessed 04/27/2024).