NIST Special Publication 800-204A

Building Secure Microservices-based Applications Using Service-Mesh Architecture

Ramaswamy Chandramouli
Zack Butcher

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-204A

C O M P U T E R S E C U R I T Y

National Institute of Standards and Technology
U.S. Department of Commerce

AUTHENTICATED U.S. GOVERNMENT INFORMATION GPO

NIST Special Publication 800-204A

Building Secure Microservices-based Applications Using Service-Mesh Architecture

Ramaswamy Chandramouli
Computer Security Division
Information Technology Laboratory

Zack Butcher
Tetrate
San Francisco, CA

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-204A

May 2020



U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Walter Copan, NIST Director and Under Secretary of Commerce for Standards and Technology

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 *et seq*., Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

National Institute of Standards and Technology Special Publication 800-204A
Natl. Inst. Stand. Technol. Spec. Publ. 800-204A, 27 pages (May 2020)
CODEN: NSPUE2

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-204A

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at https://csrc.nist.gov/publications.

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: sp800-204a-comments@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in Federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

Abstract

The increasing trend in building microservices-based applications calls for addressing security in all aspects of service-to-service interactions due to their unique characteristics. The distributed cross-domain nature of microservices needs secure token service (STS), key management and encryption services for authentication and authorization, and secure communication protocols. The ephemeral nature of clustered containers (by which microservices are implemented) calls for secure service discovery. The availability requirement calls for: (a) resiliency techniques, such as load balancing, circuit breaking, and throttling, and (b) continuous monitoring (for the health of the service). The *service mesh* is the best-known approach that can facilitate specification of these requirements at a level of abstraction such that it can be uniformly and consistently defined while also being effectively implemented without making changes to individual microservice code. The purpose of this document is to provide deployment guidance for proxy-based Service Mesh components that collectively form a robust security infrastructure for supporting microservices-based applications.

Keywords

API gateway; Application Programming Interface (API); circuit breaker; load balancing; microservices; Service Mesh; service proxy.

Acknowledgements

The authors like to express their sincere thanks to Brandon Lum, Senior Software Engineer at IBM Thomas J. Watson Research Center for his extensive, insightful feedback. They also express thanks to Isabel Van Wyk of G2-Inc and Tia Louden of Tetrate Inc. for their detailed editorial review.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

Patent Disclosure Notice

NOTICE: The Information Technology Laboratory (ITL) has requested that holders of patent claims whose use may be required for compliance with the guidance or requirements of this publication disclose such patent claims to ITL. However, holders of patents are not obligated to respond to ITL calls for patents and ITL has not undertaken a patent search in order to identify which, if any, patents may apply to this publication.

As of the date of publication and following call(s) for the identification of patent claims whose use may be required for compliance with the guidance or requirements of this publication, no such patent claims have been identified to ITL.

No representation is made or implied by ITL that licenses are not required to avoid patent infringement in the use of this publication.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

Executive Summary

Microservices-based application architectures are becoming the norm for building cloud-based and large enterprise applications because of their inherent scalability, agility of deployment, and availability of tools. At the same time, the characteristics of microservices-based applications bring with them modified/enhanced security requirements.

A few examples of these characteristics and their security impacts include:

a. The sheer number of microservices results in more interconnections and more communication links to be protected.
b. The ephemeral nature of microservices calls for secure service discovery mechanisms.
c. The fine-grained nature of microservices calls for the ability to support fine-grained authorization policies.

The supporting services (e.g., authentication/authorization, security monitoring, etc.) for a microservices-based application must be tightly coordinated through a dedicated infrastructure, such as the Service Mesh. There are multiple ways of deploying the components of the Service Mesh, including embedding them in the application (microservice) code, coupling them to the application code by implementing them as libraries, or implementing them as service proxies that are independent of application code. This last deployment approach has been found to be the most efficient in terms of scalability and flexibility in many scenarios for implementing the supporting infrastructure for microservices-based applications.

The purpose of this document is to provide deployment guidance for service mesh components in the service proxy-based approach. The Service Mesh deployment recommendations span the following aspects:

- Configuration for communication between service proxies
- Configuration for ingress proxies
- Configuration for access to external services
- Configuration for identity and access management
- Configuration for monitoring capabilities
- Configuration for network resilience
- Configuration for cross-origin resource sharing (CORS)
- Configuration of Permissions for Administrative Operations

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

Table of Contents

Executive Summary v
1 Introduction 1
1.1 Why Service Mesh 1
1.2 Scope 2
1.3 Target Audience 2
1.4 Relationship to other NIST Guidance Documents 2
1.5 Organization of this Document 2
2 Microservices-based Application – Background and Security Requirements 4
2.1 Authentication and Authorization Requirements 4
2.2 Service Discovery 4
2.3 Improving Availability through Network Resilience Techniques 4
2.4 Application Monitoring Requirement 5
3 Service Mesh – Definitions and Technology Background 6
3.1 Service Mesh Components & Capabilities 7
3.1.1 Ingress Controller 8
3.1.2 Egress Controller 8
3.2 Service Mesh as Communication Middleware: What is Different 9
3.3 Service Mesh: State of the Art 10
4 Service Mesh Deployment Recommendations 11
4.1 Communication Configuration for Service Proxies 11
4.2 Configuration for Ingress Proxies 12
4.3 Configuration for Access to External Services 12
4.4 Configuration for Identity and Access Management 12
4.5 Configuration for Monitoring Capabilities 14
4.6 Configuration for Network Resilience Techniques 16
4.7 Configuration for Cross-Origin Resource Sharing (CORS) 16
4.8 Configuration of Permissions for Administrative Operations 16
5 Summary and Conclusions 17
References 18

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

1 Introduction

Microservices architecture has become an established approach for building enterprise and cloud-based applications due to the following:

- Agility – The loose coupling and increased modularity of microservices have enabled independent and quicker modification and deployment without affecting other components (microservices) of a microservices-based application.
- Scalability – The characteristics of the microservices allow them to be independently scaled.
- Usability – The use of well-defined application programming interfaces (APIs) makes integration or onboarding of various microservices easier.
- Availability of tools – The increasing availability of automation tools facilitate error-free configuration and deployment.

In spite of the above advantages, the architecture of microservices-based applications has some challenges with modified/enhanced security requirements, such as:

- More microservices lead to more interconnections between these components as well as more communication links to be protected.
- Components (microservices) can come and go dynamically, so the environment needs secure service discovery requirements.
- There is no concept of a network perimeter.
- All microservices must be treated as non-trustworthy.
- The fine-grained nature of microservices requires fine-grained authorizations at each microservice. However, this may require security policies to be centrally defined and the configurations reflecting them to be defined in each microservice to enable uniform, consistent enforcement across all microservices.

1.1 Why Service Mesh

Due to the security requirements for microservices-based applications stated above, the infrastructure that supports the application and that infrastructure's associated services (e.g., security) should be tightly coordinated. One such dedicated infrastructure is the Service Mesh. The code that implements the Service Mesh can be organized in the following ways with respect to the components of a microservices-based application architecture (Each architectural pattern is denoted using the acronym SM-ARx where SM stands for Service Mesh, AR stands for architecture and x is the sequence number):

- SM-AR1: Service Mesh code can be embedded in the microservices application code, making the Service Mesh an integral part of the application development framework.
- SM-AR2: Service Mesh code implemented as libraries and, therefore, applications are coupled to the services provided by the Service Mesh via API calls.
- SM-AR3: Service Mesh functions are implemented in proxies with each proxy deployed in front of a microservice instance and collectively providing infrastructure services for the microservices-based application. These proxies are called "side-car proxies" and can

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

be implemented and operated independently of the application code. Side-car proxies enable heterogeneous platforms (different languages and application development frameworks) to be controlled consistently by adopting the lowest common denominator API—the network.

- SM-AR4: Service Mesh functions are implemented in proxies with a proxy deployed per node (physical host) rather than per microservice instance (such as SM-AR3).

1.2 Scope

For the purpose of this document, the only Service Mesh architecture that will be considered will be SM-AR3, where a dedicated infrastructure layer provides all security functionality to the microservices-based application without any modification to the application service's code. Compared to SM-AR4, SM-AR3 avoids a range of privilege escalation and noisy neighbor problems by deploying one instance of the service proxy per microservice instance and relying on the underlying platform's isolation guarantees to ensure the application's traffic is only mediated by its dedicated service proxy. Compared to SM-AR1 and SM-AR2, SM-AR3 decouples the application lifecycle from the modules that provide the service mesh functionality and avoids the combinatorial explosion of having to maintain the multiple versions of the library across languages, which could potentially happen in SM-AR2. Based on this context, the primary function of Service Mesh from the perspective of this document is to mediate and broker client-to-microservice and microservice-to-microservice communications where the mediating and brokering agents or functional modules do not have tight coupling with the microservice's code.

1.3 Target Audience

The target audience of the guidance document for supporting microservices-based applications using the Service Mesh framework includes security solutions architects who want to design a security framework for microservices-based applications and system integrators who build a common infrastructure services framework for different microservices-based applications residing in the enterprise and the cloud.

1.4 Relationship to other NIST Guidance Documents

This guidance document focuses on building a specific security framework or infrastructure for microservices-based applications. Understanding the characteristics of microservices-based applications and their overall security requirements and strategies is beneficial, and information is provided in the NIST Special Publication (SP) 800-204, *Security Strategies for Microservices-based Application Systems* [1].

1.5 Organization of this Document

The organization of this document is as follows:

- Chapter 2 recaps the security requirements for microservices-based applications by referencing those that were discussed in [1].

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

- Chapter 3 introduces Service Mesh and provides a brief description of its components, capabilities, and unique role as a communication middleware for microservices-based applications.
- Chapter 4 provides detailed deployment recommendations for Service Mesh components spanning configuration areas such as service proxies, ingress proxies, egress proxies, identity and access management, monitoring capabilities, network resilience techniques, and cross-origin resource sharing.
- Chapter 5 provides the summary and conclusions.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

2 Microservices-based Application – Background and Security Requirements

The definition and description of microservices-based application, threats, and security strategies for countering those threats are described in NIST SP 800-204, *Security Strategies for Microservices-based Application Systems* [1]. The purpose of this chapter is to recap and elaborate on the security requirements for this class of application to provide context for how those requirements are met by the functionality of the Service Mesh described in Chapter 3. This facilitates the development of deployment recommendations for Service Mesh components to meet those requirements in Chapter 4.

2.1 Authentication and Authorization Requirements

Authentication and access policy may vary depending on the type of APIs exposed by microservices—some may be public APIs, private APIs, or partner APIs, which are available only for business partners. There are multiple microservices, and the authentication policies should be defined to provide coverage for all of them. Further, certificate-based authentication requires a public key infrastructure (PKI) for certificate generation/management and key management. Further authorization modules covering resources in all microservices must be built to provide fine-grained authorization in all service requests.

2.2 Service Discovery

In legacy distributed systems, there are multiple services configured to operate at designated locations (IP address and port number). In the microservices-based application, the following scenario exists and calls for a robust service discovery mechanism:

a) There are a substantial number of services and many instances associated with each service with dynamically changing locations.
b) Each of the microservices may be implemented in Virtual Machines (VMs) or as containers, which may be assigned dynamic IP addresses, especially when they are hosted in an Infrastructure as a Service (IAAS) or Software as a Service (SAAS) cloud service.
c) The number of instances associated with a service can vary based on the load fluctuations using features such as autoscaling.

Based on the above characteristics, a feature to discover a service while making a service request is an essential requirement. A common approach to implementing this feature is the use of a service registry. A service registry consists of a directory where new service instances created for the microservices-based application register themselves while service instances going offline are deleted from it.

2.3 Improving Availability through Network Resilience Techniques

- Load balancing: There is a need to have multiple instances of the same service, and the loads on these instances must be evenly distributed to avoid delayed responses or service crashes due to overload.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

- Circuit breaker: Large-scale distributed systems, no matter how they are architected, have one defining characteristic—they provide many opportunities for small, localized failures to escalate into system-wide catastrophic failures. The Service Mesh must be designed to safeguard against these escalations by shedding load and failing quickly when the underlying systems approach their limits. Circuit breaking involves setting a threshold for the failed responses from an instance of a microservice and cutting off forwarding requests to that instance when the failure is above the threshold (e.g., when the circuit breaker trips). This mitigates the possibility of cascading failure and allows for time to analyze logs, implement the necessary fix, and push an update for the failing instance. Thus, circuit breaking is a temporary measure that prevents total disruption to responses for service requests. The service requests will be restored to the instance once the service is responsive.
- Rate limiting (throttling): The rate of requests coming into a microservice must be limited to ensure continued availability of service for all clients.
- Blue/green deployments: When a new version of a microservice is deployed, requests from customers using the old version can be redirected to the new version using the API gateway that can be programmed to maintain awareness of the locations of both versions.
- Canary releases: Only a limited amount of traffic is initially sent to a new version of a microservice since the correctness of its response or performance metric under all operating scenarios is not fully known. Once sufficient data is gathered about its operating characteristics, then all of the requests can be proxied to the new version of the microservice.

2.4 Application Monitoring Requirement

To detect attacks and identify factors for degradation of services (which may impact availability), it is necessary to monitor network traffic into and out of microservices through distributed logging, generation of metrics, performance of analytics, and tracing.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

3 Service Mesh – Definitions and Technology Background

From the description of microservices in the previous chapter, it should be clear that a microservice has two broad functions [2]:

- **Business Logic**, which implements the business functionalities, computations, and service composition/integration logic, and
- **Network Functions**, which take care of the inter-service communication mechanisms (e.g., basic service invocation through a given protocol, apply resiliency and stability patterns, service discovery, etc.) These network functions are built on top of the underlying OS level network stack.

The business logic function must be an integral part of the microservice code since that service is the one that executes or supports a business process. The difficulty with the microservice directly performing the network functions is that it uses different libraries depending on the programming language it is written in or the development framework it is hosted on. With the practical reality of microservices being written in multiple languages (e.g., Java, JavaScript, Python, etc.) within the same application to optimize the development or runtime process, it becomes a tedious task to provide the communication capability for each service node.

A Service Mesh is a dedicated infrastructure layer with a set of deployed infrastructure functions that facilitate service-to-service communication through service discovery, routing and internal load balancing, traffic configuration, encryption, authentication, authorization, metrics, and monitoring. It provides the capability to declaratively define network behavior, microservice instance identity, and traffic flow through policy in an environment of changing network topology due to service instances coming and going offline and continuously being relocated. It can be looked upon as a networking model that sits at a layer of abstraction above the transport layer of the Open Systems Interconnection (OSI) model (e.g., Transmission Control Protocol/Internet Protocol (TCP/IP)) and addresses the service's session layer (Layer 5 of the OSI model) concerns. However, fine-grained authorization may still need to be performed at the microservice level since that is the only entity that has full knowledge of the business logic.

Alternatively, the Service Mesh can be defined as "a distributed computing middleware that optimizes communications between application services [3]." The service-to-service communication is most effectively enabled using a proxy (see Section 1.1). A Service Mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code without the application needing to be aware [4].

In addition, the Service Mesh can be leveraged to monitor and secure communication. Because it is intercepting and routing all cluster traffic and gathering health metrics, the Service Mesh can learn and intelligently route traffic. Examples of this higher-level functionality include A/B testing, canary deployments, beta channels, automatic retries, circuit breakers, and injecting faults. These features are only possible because the Service Mesh is able to view and learn from the entire cluster's traffic.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

It is considered economical to deploy Service Mesh when the number of microservices in the application is in the order of hundreds or thousands. However, the Service Mesh is not without some drawbacks. Because each microservice requires its own service proxy, the number of runtime instances and the overall attack surface for the application increases. As the functionality built into a service proxy increases, it may become a communication bottleneck.

3.1 Service Mesh Components & Capabilities

A Service Mesh consists of two main architectural layers or components:

- Data plane
- Control plane

The interconnected set of proxies in a Service Mesh that control the inter-services communication represents its data plane. The data plane is the data path and provides the ability to forward requests from the applications. A data plane may also provide more sophisticated features like health checking, load balancing, circuit breaking, timeouts, retries, authentication, and authorization [5]. The specialized proxy that is created for each service instance (i.e., side-car proxy) performs the runtime operations needed for enforcing security (e.g., access control, communication-related), which are enabled by injecting policies (e.g., access control policies) into the proxy from the control plane. This also provides the flexibility to dynamically change policies without modifying the microservice's code.

A control plane is a set of APIs and tools used to control and configure data plane (proxy) behavior across the mesh. The Service Mesh control plane is distinct from the orchestrator's control plane—the former controls the Service Mesh, while the latter controls the cluster. The control plane is where users specify authentication policies and naming information, gather metrics (in general telemetry collection), and configure the data plane as a whole [6]. The intelligence, data, and other artifacts required for implementing all security functions lie in the control plane. These include the software for generating authentication certificates and the repository for storing them, policies for authentication, authorization engine, software for receiving telemetry/monitoring data regarding each microservice and aggregating them, and APIs for modifying the behavior of the network through various features, such as load balancing, circuit breaking, or rate limiting. The control plane of the Service Mesh platform has to be integrated with the orchestration platform (as it gets critical data from the platform, such as service registry) of the microservices-based application and should therefore have the required integration capabilities to be useful. Since the control plane is a critical component of the Service Mesh, it must be highly available and distributed. A control plane can be implemented through configuration files, API calls, and user interfaces [7].

As part of the process of providing the communication, the following functions are supported [1,2]:

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

- Authentication and authorization – Certificate generation, key management, whitelist and blacklist, Single sign-on (SSO) tokens, API keys
- Secure service discovery – Discovery of service endpoints through a dedicated service registry
- Secure communication – Mutual Transport Layer Security (TLS), encryption, dynamic route generation, multiple protocol support, including protocol translation where required (e.g., Hypertext Transfer Protocol (HTTP)1.x, HTTP2, gRPC, etc.)
- Resilience/stability features for communication – Circuit breakers, retries, timeouts, fault injection/handling, load balancing, failover, rate limiting, request shadowing
- Observability/monitoring features – Logging, metrics, distributed tracing

3.1.1 Ingress Controller

The service proxy of a Service Mesh can be deployed for control of ingress traffic (i.e., external traffic coming into microservices application as opposed to microservice-to-microservice communication). In this sense, it realizes the functions of an API gateway. Conceptually, the ingress controller can be looked upon as a side-car proxy for an external client. The ingress controller (sometimes called the front proxy) provides the following functions:

- A common API for all clients shielding the actual API inside the Service Mesh
- Protocol translation from web-friendly protocols, such as HTTP/Hypertext Transfer Protocol Secure (HTTPS), to protocols used by microservices, such as RPC/gRPC/Representational State Transfer (REST)
- Composition of results received from calls to multiple services inside the Service Mesh in response to a single call from the client
- Load balancing
- Public TLS termination

3.1.2 Egress Controller

The service proxy of a Service Mesh can be deployed for control of egress traffic (i.e., internal traffic coming from microservices destined for microservices outside of the mesh). In this sense, it functions as an egress-only gateway. Conceptually, the egress controller can be looked upon as a side-car proxy for one or more external servers. The egress proxy provides the following functions:

- A single set of workloads (e.g., hosts, IP addresses) to whitelist for communication to external networks (e.g., firewalls can be configured to allow only egress proxies to forward traffic out of the local network)
- Credential exchange: Translate from internal (mesh) identity credentials to external credentials (e.g., SSO tokens or API keys) without the application directly accessing the external system's credentials
- Protocol translation from microservice-friendly protocols (e.g., RPC/gRPC/REST) to web-friendly protocols (e.g., HTTP/HTTPS)

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

3.2 Service Mesh as Communication Middleware: What is Different

Prior to the Service Mesh, in order to provide infrastructure functionality (e.g., service discovery, load balancing, circuit breaking, fault injection, security monitoring, authorization, and distributed tracing for distributed systems such as microservices-based applications), a set of components and frameworks that provide these functionalities must be carefully chosen. Some components will only work within certain frameworks, and frameworks themselves are tied to specific languages. Additionally, the application service's code must be modified to work with or be integrated into these components [8].

In the case of Service Mesh, the technology or programming language in which the individual microservices are written do not matter since operation occurs at the network level. For example, if the microservices application developer develops an HTTP server, they have complete freedom to choose any language—Java, C++, Rust, Go, JavaScript, Python, etc. Service Mesh decouples application code from the management of service-to-service communication. The application code does not need to know about network topology, service discovery, load balancing, or connection management logic [9]. Features like telemetry, traffic shaping, service discovery, and network policy control can be provided out of the box as well.

Since Service Mesh is defined as a communication middleware, the next question that arises is, "how does it differ from any other distributed system middleware?" The traditional middleware for distributed systems includes application delivery controllers (ADCs), load balancers, and API gateways. In addition to having heavy costs and operating overheads, these middleware appliances are unsuitable in contexts where the application components they serve are in the form of loosely coupled modular microservices since these components require fine-grained capabilities and functionalities, such as dynamic discovery, which are not required by modules of monolithic applications.

In order to understand the unsuitability of traditional communication middleware components for distributed systems and the need for lightweight solutions for microservices-based application systems, consider the nature of communications in those systems. Clients of various types interface with an application made up of a huge number of microservices. The communication traffic between the clients and any application service is called "north-south" traffic, and those between one microservice and another is called "east-west" traffic. Because of the relatively high number of microservices as components in a microservices-based application compared to a monolithic application, the amount of east-west traffic is so high that a lightweight communication middleware, such as the Service Mesh, is needed to provide an acceptable level of performance for a production application.

Though a microservices-based application can be implemented without a container and its associated orchestration service, it is often identified as a cloud-native application with a service-based architecture, API-driven communications, container-based infrastructure, and a bias for DevOps (i.e., development and operations) processes (e.g., continuous improvement, agile development, continuous delivery, collaborative development among developers, quality assurance teams, security professionals, IT operations, and line-of-business stakeholders [3]). Part of the reason for this perspective is that monolithic software development and deployment

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

relies on a server-centric infrastructure with tightly integrated application modules rather than on loosely coupled, services-based architectures with API-based communications.

3.3 Service Mesh: State of the Art

Conceptually, a Service Mesh can be used to provide infrastructure services for all applications based on microservices architecture in which there are hundreds of services and tens of instances in each service. However, based on the Off-the-Shelf implementations available today, Service Mesh is most suitable and productive for application platforms with the following configurations:

- Each microservice is implemented as a container.
- The application makes use of container clusters (for improved availability and performance) that are managed using container orchestration tools.
- The application is hosted through a Container-as-a-Service (CaaS) offered by cloud providers and has the necessary deployment and configuration tools found in container management/orchestration environments.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

4 Service Mesh Deployment Recommendations

This chapter considers the deployment options in Service Mesh and provides recommendations for secure microservices-based application for various application scenarios. Since the primary runtime functions are performed by proxies, the first deployment recommendations are in the context of configuring the various aspects of proxy functions. Each of the deployment recommendations are identified through the symbol SM-DRx, where SM stands for Service Mesh, DR stands for deployment recommendation, and x is the number in the sequence.

4.1 Communication Configuration for Service Proxies

Recommendation for Allowed Traffic into Service Proxies (SM-DR1): *There should be a feature to specify the set of protocols and ports into which a service proxy can accept traffic for its associated service. By default, a service proxy should not allow traffic except as specified by this configuration.*

Recommendation for Reachability of Service Proxies (SM-DR2): *The set of services that a service proxy can reach must be limited. There should be features to limit access based on namespace, a specific named service within a given namespace, or the service's runtime identity. Access to the control plane of the Service Mesh must always be provided to relay discovery, different policies, and telemetry data.*

Recommendation for Protocol Translation Capabilities (SM-DR3): *The service proxy should have built-in capabilities to support clients communicating with different protocols than the target microservice (e.g., convert REST/HTTP requests to gRPC requests or upgrade HTTP/1.1 to HTTP/2). This is required to avoid the need for building a separate server per client protocol, which increases the attack surface.*

Recommendation for User Extensibility (SM-DR4): *The service proxy should have features for defining custom logic in addition to built-in logic for handling network functions. This is required to ensure that the service proxy can be extended to implement use case specific policies (e.g., preexisting or home-grown policy engines). The implementation should provide means to control the risks of extensibility, such as sandboxing, restricting the APIs/runtimes of languages used, or performing pre-analysis that ensures safety (e.g. WASM or eBPF).*

Recommendation for Dynamic Configuration Features for Proxies (SM-DR5): *There should be options to configure proxies dynamically (e.g., event-driven configuration updates) in addition to static configuration. In other words, there should be discovery services for those entities that are expected to be dynamic rather than known at the time of deployment. Further, the proxy should atomically swap to the new dynamic configuration at runtime while efficiently handling (i.e., completing or terminating) outstanding requests under the previous configuration. This is required for timely enforcement of policy changes at runtime without any degradation of user traffic or downtime (i.e., without restarting the service proxy).*

Recommendation for configuring communication between application service to its proxy (SM-DR6): *The application service and its associated proxy should only communicate through a*

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

loopback channel (e.g., localhost IP address, UNIX domain socket, etc.). Further, service proxies should only communicate with each other by setting up a mutual TLS (mTLS) session where every exchanged data packet is encrypted.

4.2 Configuration for Ingress Proxies

Recommendation for Ingress Proxies (SM-DR7): *There should be features for configuring traffic routing rules for ingress (standalone) proxies just like service proxies. This is needed because consistent enforcement of routing policy is required all the way to the edge of the application deployment.*

4.3 Configuration for Access to External Services

Certain services in the microservices-based application may have to access some public or private web APIs, third-party services, legacy applications, or applications in different virtualized infrastructures, such as in VMs or different clusters (than the one in which the Service Mesh runs). To provide the same security assurance for access to these resources, the following recommendations are provided:

Recommendation for Restricting Access to External Resources (SM-DR8): *Access to external resources or services outside of the mesh should be disabled by default and only allowed by an explicit policy that restricts access to specified destinations. Additionally, those external resources or services should be modeled as services in the Service Mesh itself (e.g., by including them in the Service Mesh's service discovery mechanism).*

Recommendation for Secure Access to External Resources (SM-DR9): *The same availability improvement features (e.g., retries, timeouts, circuit breakers, etc.) that are configured for services inside of the Service Mesh must be provided for access to external resources and services.*

Recommendation for Egress Proxies (SM-DR10): *There should be features for configuring traffic routing rules for egress (standalone) proxies just like service and ingress proxies. When deployed, access to external resources and services should be mediated by these egress proxies. The egress proxy should implement access and availability policies (SM-DR8). This is useful for working with traditional network-oriented security models. For example, suppose that outbound traffic to the internet is only allowed from a specific IP in the network; an egress proxy can be configured to run with that address while proxying traffic for a range of services in the mesh.*

4.4 Configuration for Identity and Access Management

The three main communicating entities of a microservices-based application are clients, microservices, and external services. During the communication events of any pair (i.e., client-to-microservice, microservice-to-microservice, or microservice-to-egress-service), both entities need to have distinct identities and perform mutual authentication. Since mutual TLS (mTLS) is the de facto mechanism for doing this, the authentication certificate that a client or microservice holds should carry its identity in its "subject name" or "subject alternative name" fields. This

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

identity can either be a server identity (also known as a host or domain) or a service identity (usually service account ID). The recommendations relating to certificate deployment are as follows:

Recommendation for a Universal Identity Domain (SM-DR11): *The identity of all instances of a microservice should be consistent and unique—consistent in that a service should have the same name regardless of where it is running and unique in that across the entire system, the service's name corresponds only to that service. This does not mean different logical services in different locations; a typical usage of Domain Name System (DNS) where each service is assigned its own DNS name would satisfy this recommendation. Consistent names (identities) for services are required so that the system policy is manageable.*

Recommendation for Signing Certificate Deployment (SM-DR12): *The Service Mesh control plane's certificate management system should have its ability to generate self-signed certificates disabled. This functionality is frequently used to bootstrap an initial signing certificate for all other identity certificates in the Service Mesh. Instead, the signing certificate used by the mesh's control plane should always be rooted in the enterprise's existing PKI's root of trust and provided securely to the Service Mesh control plane at startup. This simplifies the management of those certificates by an existing PKI (e.g., for revocation or audit). Further, we recommend that separate intermediate signing certificates should be generated for different domains to simplify rotation and enable fine-grained revocation.*

Recommendation for Identity Certificate Rotation (SM-DR13): *The lifetime of a microservice's identity certificate should be as short as is manageable within the infrastructure—preferably on the order of hours. This helps limit attacks since an attacker can only use a credential to impersonate a service until that credential expires, and successfully re-stealing a credential increases the difficulty for an attacker.*

Recommendation for the Service Proxy to Cycle Connections on Identity Change (SM-DR14): *When a service proxy's identity certificate is rotated, the service proxy should efficiently retire existing connections and establish all new connections with the new certificate. Certificates are only validated during the mTLS handshake, so replacing existing connections when a new certificate is issued is not strictly required; rather, this is important for limiting attacks in time.*

Recommendation for Non-Signing Identity Certificates (SM-DR15): *Certificates used to identify microservices should not be signing certificates.*

Recommendation for Workload Authentication before Certificate Issuance (SM-DR16)*: The Service Mesh control plane's certificate management system should perform authentication of a service instance before issuing it an identity certificate. In many systems, this can be achieved by attesting the instance against the system's orchestration engine, and by using other local proofs (e.g. secrets retrieved from an HSM).*

The same care should be taken in provisioning the signing certificate for the Service Mesh control plane's certificate management system. That signing certificate should be retrievable only by the Service Mesh control plane and only after some form of attestation has been done

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

against it.

Recommendation for Secure Naming Service (SM-DR17): *If the certificate used for mTLS carries server identity, then the Service Mesh should provide a secure naming service that maps the server identity to the microservice name that is provided by the secure discovery service or DNS. This requirement is needed to ensure that the server is the authorized location for the microservices and to protect against network hijacking.*

If the certificate used for mTLS carries the service identity, no additional secure naming service is required. This also ensures that when the microservice is ported to a different network domain (different cluster or different cloud location), the identity and associated access control policies need not be defined again for the new location.

Setting up certificates for microservices based on service identity enables two communicating services to establish a secure communication channel but does not specify whether they are allowed to communicate at all in the place. To specify this, a feature to define policies for allowed inbound and outbound traffic for each microservice node is required.

Recommendation for Granular Identity (SM-DR18): *Each microservice should have its own identity, and all instances of this service should present the same identity at runtime. This allows for access policy at the level of microservice in a given namespace. This is required since common microservice runtimes default to issuing identities per namespace rather than per service so that all services in the same namespace present the same runtime identity unless otherwise specified. In addition, labels can be used to augment the service name (identity) to enable granular logging configuration and to support granular authorization policies.*

Recommendation for Authentication Policy Scope (SM-DR19): *The feature to specify the policy scope for authentication should have the following minimal options: (a) all microservices in all namespaces, (b) all microservices in a particular namespace, and (c) a specific microservice in a given namespace (using the runtime identity referred to in SM-DR17).*

The above described approach enables authentication using static parameters (i.e., service identity and pre-defined policies). An additional requirement is the ability to incorporate some contextual information (such as the user invoking the microservice) in some scenarios. A mechanism for this is to use tokens encoded in platform-neutral format (e.g., JavaScript Object Notation (JSON) Web Tokens or JWT). The requirements for the token are:

Recommendation for Authentication Token (SM-DR20): *Tokens should be digitally signed and encrypted so that claims included in them have the assurance of authenticity since these claims can be used to augment or be part of an authenticated identity to build access control decisions. Further, these tokens must only be passed by loopback device (to ensure that there is no network path involved) or through an encrypted channel.*

4.5 Configuration for Monitoring Capabilities

All proxies (ingress, egress, and service) should have the capability to collect all monitoring

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

data. Monitoring data is classified into three categories: logging, metrics, and traces in software systems, as identified by Peter Bourgon [10]. This capability is realized by enabling integration support in the Service Mesh for specialized tools that can generate one or more of the categories of data mentioned above. Examples include AppSensor and Fluentd for event logging, Prometheus for aggregatable metrics, and Jaeger [11] and Zipkin [12] for distributed tracing. This allows Service Mesh deployment teams to minimize the effort of building the pipelines for data collection and focus on data analytics. Depending on the use case context, example could include event mining, anomaly detection, or service dependency extraction.

The *logging* data should, at a minimum, record the following events to detect some common attacks:

Recommendation for Logging Events (SM-DR21): *The proxy should log input validation errors and extra (unexpected) parameters errors, crashes, and core dumps. Common attack detection capabilities should include bearer token reuse attack and injection attacks.*

Recommendation for Logging Requests (SM-DR22): *The proxy should log at least the Common Log Format fields for irregular requests (e.g., non-200 responses when using HTTP). Logging for successful requests (e.g., 200 responses) tends to be of little value when metrics are available.*

Recommendation for Log Message Content (SM-DR22): *Log messages should contain, at a minimum, the event date/time, microservice name or identity, request trace id, message, and other contextual information (e.g., requesting user identity and URL). However, logging should take care to mask sensitive information, for example Bearer tokens.*

On the *metrics* side, a baseline for normal, uncompromised behavior in terms of the outcome of business logic decisions, contact attempts, and other behavior should be created. To enable this:

Recommendations for Mandatory Metrics (SM-DR23): *The configuration for gathering metrics using Service Mesh for external client and microservice calls should involve, at a minimum, the following: (a) the number of client/service requests in a given duration, (b) the number of failed client/service requests by failure code, and (c) the average latency per service as well as the average total latency per complete request lifecycle (ideally as a histogram; also by failure code).*

A *trace* is a representation of a series of causally related distributed events that encode the end-to-end request flow through a distributed system [13]. A trace can provide visibility into both the path traversed by a request (various microservices involved) and the structure of a request (forking logic and the nature of the request—synchronous or asynchronous). In the context of a microservices-based application and Service Mesh, it is called distributed tracing since the tracing function is enabled by a combination of various application services and their associated service proxies. Because of its use for debugging steady-state problems and in providing data for capacity planning, the distributed tracing feature has a direct impact on the application system availability.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

Recommendation for Implementing Distributed Tracing (SM-DR24): *When configuring the service proxies for implementing distributed tracing, care should be taken to ensure that the application services are instrumented to forward the headers for communication packets they received.*

It is important to note that in order to carry out the above recommendation, a minimal amount of instrumenting the application service is necessary, unlike other infrastructure functions provided under Service Mesh architecture [14].

4.6 Configuration for Network Resilience Techniques

Recommendation for Storing Data for Implementation of Network Resilience (SM-DR25): *Data pertaining to retries, timeouts, circuit breaking settings, or canary deployments (in general, all control plane configuration data) should be stored in robust data stores, such as Key/Value stores.*

Recommendation for Implementation of Health Checking of Service Instances (SM-DR26): *The health checking function for service instances should be tightly integrated with the service discovery function to maintain the integrity of the information used for load balancing.*

This health data can be reported to a central health checking service (or a central service discovery system), but it may also be used locally only (e.g., a service proxy performs health checking on open connections it maintains, makes local load balancing decisions, and avoids hosts it deems to be unhealthy).

4.7 Configuration for Cross-Origin Resource Sharing (CORS)

Recommendation for Cross-Origin Resource Sharing (CORS) (SM-DR27): *An edge service (i.e., entry point for microservice) may often have to be configured for CORS for communicating with external services, such as a web UI client service [15]. The CORS policy for an edge service should be configured using the Service Mesh capability (e.g., VirtualService resource's CorsPolicy configuration in Istio) rather than handling it through the microservice application service code.*

4.8 Configuration of Permissions for Administrative Operations

Recommendation for Access Control for Administrative Operations (SM-DR28): *There should be granular access control permissions for all administrative operations in a service mesh (e.g., policy specifications, configuration parameters for service resiliency parameters, canary deployments, retries, etc.) that are specifiable at the level of all services in a namespace, a named service within a namespace, etc. Generally, the interface for exercising this functionality may not be part of the service mesh itself but part of an installation software or orchestration software used for configuring the application service cluster.*

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

5 Summary and Conclusions

The increasing adoption of microservices-based applications in cloud and large enterprise environments has prompted the identification of an infrastructure that provides a comprehensive, consistent, and coordinated set of support services. The Service Mesh is one such infrastructure, and the state of practice for deployment of Service Mesh components is a proxy-based approach that can provide all support services without any change to the application code.

The proxy-based Service Mesh can be engineered to build and integrate support service components that provide secure service discovery, definition, and enforcement of authentication and authorization policies, network resilience features, and performance and security monitoring capabilities. The primary contribution of this document is to provide detailed deployment guidance for each of the Service Mesh components spanning the areas listed above.

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

References

[1] Chandramouli R (2019) Security Strategies for Microservices-based Application Systems. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-204. https://doi.org/10.6028/NIST.SP.800-204

[2] Indrasiri K (2017) *Service Mesh for Microservices*. Available at https://medium.com/microservices-in-practice/service-mesh-for-microservices-2953109a3c9a

[3] Lerner A, Thomas A (2018) *Innovation Insight for Service Mesh*. Available at https://www.gartner.com/en/documents/3894156/innovation-insight-for-service-mesh

[4] Morgan W (2017) *What's a service mesh? And why do I need one?* Available at https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/#_ga=2.39876518.581209135.1575405453-1411561046.1575405453

[5] Mishra A (2017) *Smart Networking with Consul and Service Meshes.* Available at. https://www.hashicorp.com/blog/smart-networking-with-consul-and-service-meshes/

[6] Schiesser M (2019) *Comparing Service Meshes: Linkerd vs. Istio*. Available at https://medium.com/glasnostic/comparing-service-meshes-linkerd-vs-istio-c7e0132578a8

[7] Bryant D (2018) *The Importance of Control Planes with Service Meshes and Front Proxies.* Available at https://blog.getambassador.io/the-importance-of-control-planes-with-service-meshes-and-front-proxies-665f90c80b3d

[8] Kirschner E (2017) *Proxy Based Service Mesh*. Available at https://medium.com/@entzik/proxy-based-service-mesh-96cd4b74c198

[9] Tiwari A (2017) *A sidecar for your service mesh*. Available at https://www.abhishek-tiwari.com/a-sidecar-for-your-service-mesh/

[10] Sridharan C (2018) The Three Pillars of Observability. *Distributed Systems Observability* (O'Reilly Media, Inc., Sebastopol, CA), Chapter 4. Available at https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html

[11] Jaeger Project (2020) *Jaeger: open source, end-to-end distributed tracing*. Available at https://www.jaegertracing.io/

[12] Zipkin Project (2020) *Zipkin*. Available at https://zipkin.io/

[13] Leong A (2019) *A guide to distributed tracing with Linkerd*. Available at https://linkerd.io/2019/10/07/a-guide-to-distributed-tracing-with-linkerd/

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A

[14] Stafford GA (2019) *Kubernetes-based Microservice Observability with Istio Service Mesh: Part 1* (ITNEXT). Available at https://itnext.io/kubernetes-based-microservice-observability-with-istio-service-mesh-part-1-bed3dd0fac0b

[15] Stafford GA (2019) *Istio Observability with Go, gRPC, and Protocol Buffers-based Microservices* (ITNEXT). Available at. https://itnext.io/istio-observability-with-go-grpc-and-protocol-buffers-based-microservices-d09e34c1255a

This publication is available free of charge from: https://doi.org/10.6028/NIST.SP.800-204A