Draft 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-draft

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

Draft 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-draft

January 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, 25 pages (January 2020)
CODEN: NSPUE2

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

Public comment period: *January 17, 2020* through *February 14, 2020*

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.

Abstract

The increasing trend in building microservices-based applications calls for addressing security in all aspects of service-to-service interactions due to its unique characteristics. The distributed cross-domain nature of microservices needs secure token service (STS), key management and encryption services for authentication and authorization, as well as 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 only approach that can facilitate specification of these requirements at a level of abstraction such that it can be uniformly, consistently defined, but at the same time, 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

Call for Patent Claims

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the guidance or requirements in this Information Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

a) assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or

b) assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:

   i. under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or
   ii. without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed to: sp800-204a-comments@nist.gov
with Subject: sp800-204a PATENT CLAIMS

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 application bring with them modified/enhanced security requirements.

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

(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, and
(c) the fine-grained nature of microservices calls for the ability to support fine-grained authorization policies.

It has been found that 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 – embed them in the application (microservice) code, couple them to the application code by implementing them as libraries, or by implementing them as service proxies that are independent of application code. The last deployment approach has been found to be the most efficient in terms of scalability and flexibility 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:

- Communication Configuration for 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, and
- Configuration for Cross Origin Resource Sharing (CORS).

Table of Contents

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 has enabled independent and quicker modification and deployment without affecting other components (microservices) of a microservices-based application.
- Scalability – because of the characteristics of the microservices, they can be independently scaled.
- Usability – the use of well-defined Application Programming Interface (APIs) makes integration or onboarding of various microservices easier.
- Availability of tools – increasing availability of automation tools facilitate error-free configuration and deployment.

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

- The more microservices, the more interconnections between these components and more communication links to be protected;
- Components (microservices) can come and go out 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 and;
- 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

- From 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 implementing this Service Mesh can be organized in the following ways with respect to the components of a microservices-based application architecture: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 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 (which would be 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 rely on the underlying platform's isolation guarantees to ensure the application's traffic is only mediated by its dedicated service proxy. 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 itself.

1.3 Target Audience

The target audience for the guidance document for supporting microservices-based applications using the Service Mesh framework will Security solutions architects who want to design a security framework for microservices-based applications. System integrators who build a common infrastructure services framework for different microservices-based applications residing in the enterprise as well as in 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. Hence a read of the characteristics of microservices-based applications and their overall security requirements and strategies form a good background information and 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].
- Chapter 3 introduces Service Mesh, provides a brief description of its components and capabilities and its 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.

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 document 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 set the context for how those requirements are met by functionality provided by the Service Mesh 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; and 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 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 implement this feature is by using a service registry. A service registry consists of a dictionary 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 load on these instances must be evenly distributed to avoid delayed responses or service crashes due to overload.
- Circuit breaker: Large-scale distributed systems, no matter how they're 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 fast when the underlying systems approach their limits. Circuit breaking is the idea of setting a threshold for the failed responses from an instance of a microservice and cut off forwarding requests to that instance when the failure is above the threshold (when the circuit breaker trips). This mitigates the possibility of a cascaded failure, allows time to analyze logs, implement the necessary fix, and push an update for the failing instance.

- 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 be aware 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.

3 Service Mesh - Definitions and Technology Background

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

- **Business Logic** that implements the business functionalities, computations and service composition/integration logic.
- **Network Functions** that take care of the inter-service communication mechanisms (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/supports a business process. The difficulty with the microservice directly performing the network functions is that it uses different libraries depending upon the programming language/development framework it is written/hosted in. With the practical reality of microservices being written in multiple languages such as Java, Node.js, Python, etc., within the same application to optimize the development/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 and 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 System Interconnection (OSI) model (e.g., Transport 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 the 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 usually enabled using a proxy. 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].

It is considered economical to deploy Service Mesh when the number of microservices in the application is of the order of 100s or 1000s. 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. The communication logic relating to a business process such as service call sequence, service composition, etc., has to be built into the microservice code.

3.1 Service Mesh Components & Capabilities

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

- Data plane, and a
- Control plane.

The interconnected set of proxies in a Service Mesh that control the inter-services communication represents its data plane. The data plane provides the ability to forward requests from the applications, and is the data path, hence the name. A data plane may also provide more sophisticated features like health checking, load balancing, circuit breaking, timeouts and retries, authentication, and authorization [5]. The specialized proxy that is created for each service instance (side-car proxy) performs the runtime operations needed for enforcing security (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 control plane is where users specify authentication policies, 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 repository for storing them, the 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 hence should 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].

- Authentication & Authorization – Certificate Generation, Key Management, Whitelist & Blacklist, service-to-service access control
- Secure Service Discovery – Discovery of service endpoints through a dedicated Service Registry
- Secure Communication – mutual TLS, encryption, dynamic route generation, multiple protocol support including protocol translation where required- HTTP1.x, HTTP2, gRPC
- 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 (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/HTTPS to protocols used by microservices such as RPC/gRPC/REST.
- Provide composition of results received from calls to multiple services inside the Service Mesh in response to a single call from the client.
- Load Balancing.

3.1.2 Egress Controller

The service proxy of a Service Mesh can be deployed for control of egress traffic (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 an external server. The egress proxy provides the following functions:

- A single set of workloads (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 into external credentials (such as SSO tokens or API keys) without the application directly accessing the external system's credentials.
- Protocol translation from microservice-friendly protocols (such as RPC/gRPC/REST) to web-friendly protocols (HTTP/HTTPS).

3.2 Service Mesh as Communication Middleware: What is Different

Prior to the Service Mesh, in order to provide infrastructure functionality such as service discovery, load balancing, circuit breaking, fault injection, security monitoring, 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. Secondly the application service's code had to be modified to work with or be integrated with these components [8].

In the case of Service Mesh, it does not matter in what technology or programming language the individual microservices are written since it operates at the container level. So if the microservices application developer develops for example, a HTTP server, he or she has complete freedom to choose any language: Java, C++, Rust, Go, NodeJS, Python, etc. It decouples application code from management of service-to-service communication. The application code doesn't need to know about network topology, service discovery, load balancing

and 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 is it different from any other distributed system middleware? The traditional middleware for distributed systems includes Application Delivery Controllers (ADCs), Load Balancers and API Gateways. It has been found that these middleware appliances, apart from heavy cost and operating overheads, 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, we look at the nature of communications in those systems. These systems have clients of various types interfacing with an application that are made of up of a huge number of microservices. The communication traffic between the clients and any application service is called the "North-South" traffic and those between one microservice-to-another is called the "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 only light-weight communication middleware, likea Service Mesh, can provide the acceptable level of performance for a production application.

Though a microservices-based application can be implemented purely as an enterprise application and not as a cloud service, it is often identified as a cloud-native application with a service-based architecture, application programming interface (API)-driven communications, container-based infrastructure, and a bias for DevOps (Combination of Development and Operations) processes such as continuous improvement, agile development, continuous delivery, and 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 on-premises software development and deployment 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 where there are hundreds of services, and each service has tens of instances. However, based on the track record of deployments so far, it has been found that it is most suitable and productive for application platforms with the following configurations:

- Each microservice is implemented as a managed 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 offered by cloud providers and has the necessary deployment and configuration tools found in container management/orchestration environments.

There are now two full functional Service Mesh product suites (with one product in two different architectures) available that are both open-source. The side-car proxies that have been deployed and the common functions supported by them are given in Table 1 below [9]:

Table 1: Open-Source Service Mesh Products and Functions

Service Mesh Product Suite	Native Side-Car Proxy (Proxies)	Platforms	Common Functions Supported
Istio (Google)	Envoy (per pod)	Kubernetes Pods, Docker, VMs	Dynamic routing, Service discovery, Load balancing, TLS termination, HTTP/2 & gRPC proxying, Observability, Policy enforcement
Linkerd 1.X (CNCF)	Netty & Finagle (per node)	AWS, ECS, DC/OS and Docker	
Linkerd 2.X (CNCF)	Envoy, Consul	Kubernetes Pods	

4 Service Mesh Deployment Recommendations

In this chapter, we will look at the deployment options in Service Mesh and provide recommendations that will result in 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 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 or on a specific named service within a given namespace. Access to the control plane of the Service Mesh must always be provided to relay discovery, policy, 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 the service proxy can be extended to implement use case specific policies (e.g., pre-existing or home-grown policy engines).*

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 being known at deploy time. Further, the proxy should atomically swap to new dynamic configuration at runtime while gracefully handling (completing or terminating) outstanding requests under the previous configuration. This is required for timely enforcement of policy changes at runtime without degrading user traffic.*

4.2 Configuration for Ingress Proxies

Recommendation for Ingress Proxies (SM-DR6): *There should be features for configuring traffic routing rules for ingress (standalone) proxies just like service proxies. This is needed because consistent enforcement of 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/private web APIs, legacy applications, and 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-DR7): *Access to external resources/services outside of the mesh should be disabled by default and allowed only by explicit policy. Further, those external resources/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-DR8): *The same availability improvement features such as retries, timeouts etc. that are configured for services inside the Service Mesh must be provided for access to external resources/services. Similarly, there should be features to restrict access (e.g., ability to specify allowed destinations) for these external resources/services.*

Recommendation for Egress Proxies (SM-DR9): *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/services should be mediated by these egress proxies. The egress proxy can implement access and availability policies (SM-DR8). This is useful for working with traditional network-oriented security models (e.g., suppose 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 two main communicating entities of a microservices-based application are: Clients and Microservices. During communication events of any pair (Client-to-Microservice or Microservice-to-Microservice), 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 a client or microservice holds should carry its identity in its Subject Name or Subject Alternative Name fields. This identity can be either: (a) Server Identity (also known as host or domain) or (b) Service Identity (usually service account ID). The recommendations relating to certificate deployment are as follows:

Recommendation for a Universal Identity Domain (SM-DR10): *The identity of a microservice should be consistent and unique: Consistent in that a service should have the same name no matter where it is running, and unique in that across the entire system, the service's name corresponds only to that service; it does not sometimes mean one logical service and in another location mean some different logical service. (i.e., a typical usage of DNS where each service is assigned its own DNS name would satisfy this recommendation.) This is required so that the system policy is manageable – the policy becomes impossible to reason about without consistent names (identities) for services.*

Recommendation for Signing Certificate Deployment (SM-DR11): *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 your existing PKI's root of trust and provided securely to the Service Mesh control plane at startup. This simplifies management of those certificates by an existing PKI as required (e.g., for revocation, or audit).

Recommendation for Identity Certificate Rotation (SM-DR12): *The lifetime of an microservice's identity certificate should be as short as is manageable in your infrastructure – preferably on the order of hours. This helps limit attacks in time: an attacker can only use a credential to impersonate a service until that credential expires, and successively re-stealing a credential increases the difficulty for an attacker.*

Recommendation for the Service Proxy to Cycle Connections on Identity Change (SM-DR13)*: When a service proxy's identity certificate is rotated, the service proxy should gracefully retire existing connections and establish all new connections with the new certificate moving forward. Certificates are only validated during the mTLS handshake, so replacing existing connections when a new certificate is issued is not strictly required; instead this is important for limiting attacks in time.*

Recommendation for Non-Signing Identity Certificates (SM-DR14): *Certificates used to identify microservices should not be signing certificates.*

Recommendation for Secure Naming Service (SM-DR15): *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 set up a secure communication channel but does not specify whether they can 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-DR16): *Each microservice should have its own identity. This allows for access policy at the level of microservice in a given namespace. This is required as 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.*

Recommendation for Authentication Policy Scope (SM-DR17): *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.*

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

Recommendation for Authentication Token (SM-DR18): *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 authenticated identity to build access control decisions.*

4.5 Configuration for Monitoring Capabilities

All proxies (Ingress, Egress, and Service) should have the capability to collect all monitoring data. Monitoring data comes in 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 categories of data mentioned above. Examples are: AppSensor, 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 be event mining, anomaly detection, and service dependency extraction, etc.

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

Recommendation for Logging Events (SM-DR19): *The proxy should log* i*nput validation errors and extra (unexpected) parameters errors, crashes, and core dumps. Common attack detection capability should include Bearer token reuse attack and injection attacks.*

Recommendation for Logging Requests (SM-DR20): *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-DR21)*: Log messages should contain at the minimum runtime/stack information, including function name and line number at which the log entry started and the message.*

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-DR22): *The configuration for gathering metrics using Service Mesh should involve at the minimum the following for external client and microservice calls: (a) Number of client/services requests in a given duration; (b) Number of failed client/service requests by failure code; and (c) Average latency per service as well as*

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, since the tracing function is enabled by a combination of various application services and their associated service proxies, it is called distributed tracing. Because of its use for debugging steady-state problems and in providing data for capacity planning, distributed tracing feature has a direct impact on the application system availability.

Recommendation for Implementing Distributed Tracing (SM-DR23): *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-DR24): *Data pertaining to Retries, Timeouts, Circuit Breaking settings, Canary deployments etc. (in general all configuration 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-DR25): *The Health checking function for service instances should be tightly integrated with 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 only locally (e.g., a service proxy performs health checking on open connections it maintains and makes local load balancing decisions avoiding hosts it deems to be unhealthy).

4.7 Configuration for Cross Origin Resource Sharing (CORS)

Recommendation for Cross Origin Resource Sharing (CORS) (SM-DR26): An edge service (entry point for microservice) may often have to be configured for CORS for communicating with external service such as web UI client service [15]. *The CORS policy for an edge service must be configured using the Service Mesh capability (e.g., VirtualService resource's CorsPolicy configuration in Istio) rather than handling it through microservice application service code.*

5 Summary and Conclusions

Recognizing the increasing adoption of microservices-based applications in cloud and large enterprise environments, the motivation is to identify the 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 can provide secure service discovery, definition, and enforcement of authentication and authorization policies, network resilience features, as well as 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.

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] *Innovation Insight for Service Mesh (2018)*. Available at. https://www.gartner.com/document/3894156?ref=solrAll&refval=237373842

[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*. Availabe at. https://medium.com/glasnostic/comparing-service-meshes-linkerd-vs-istio-c7e0132578a8

[7] Bryant D (2018) *The Importance of Control Planes with Service Meshes.* 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 side-car for your service mesh*. Available at. https://www.abhishek-tiwari.com/a-sidecar-for-your-service-mesh/

[10] Chapter 4. The Three Pillars of Observability. *Distributed Systems Observability O'Reilly (2018)* https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html

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

[12] *Zipkin*. Available at. https://zipkin.io/

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

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

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