Draft NIST Special Publication 800-204

Security Strategies for Microservices-based Application Systems

Ramaswamy Chandramouli

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

COMPUTER SECURITY

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

Draft NIST Special Publication 800-204

Security Strategies for Microservices-based Application Systems

Ramaswamy Chandramouli
Computer Security Division
Information Technology Laboratory

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

March 2019

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 800-204
Natl. Inst. Stand. Technol. Spec. Publ. 800-204, 41 Pages (March 2019)
CODEN: NSPUE2

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

Public comment period: *March 25, 2019* through *April 26, 2019*

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

Microservices architecture is increasingly being used to develop application systems since its smaller codebase facilitates faster code development, testing, and deployment as well as optimization of the platform based on the type of microservice, support for independent development teams, and the ability to scale each component independently. Microservices generally communicate with each other using APIs, which requires several core features to support complex interactions between a substantial number of components. These core features include authentication and access management, service discovery, secure communication protocols, security monitoring, availability/resiliency improvement techniques (e.g., circuit breakers), load balancing and throttling, integrity assurance techniques during induction of new services, and handling of session persistence. Additionally, the core features could be bundled or packaged into architectural frameworks such as API gateways and service mesh. The purpose of this document is to analyze the multiple implementation options available for each individual core feature and configuration options in architectural frameworks, develop security strategies that counter threats specific to microservices, and enhance the overall security profile of the microservices-based application.

Keywords

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

Acknowledgements

<TBD>

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: sp800-204-comments@nist.gov.

Table of Contents

EXECUTIVE SUMMARY

The microservices paradigm is being increasingly used for designing and deploying large-scale application systems in both cloud-based and enterprise infrastructures. The resulting application system consists of relatively small, loosely coupled entities or components called microservices that communicate with each other using lightweight communication protocols.

Incentives to design and deploy a microservices-based application system include: (a) agility in development due to relatively small and less complex codebases since each one typically implements a single business function; (b) independence among teams in the development process thanks to the loosely coupled nature of microservices; and (c) availability of deployment tools that provide infrastructure services such as authentication, access control, service discovery and communication, and load balancing.

Despite several facilitating technologies (e.g., orchestration), there are many challenges to be addressed in the development and deployment of a microservices-based application. Network security, reliability, and latency are critical factors since every transaction implemented using this type of system will involve the transmission of messages across a network. Further, the presence of multiple microservices exposes a large attack surface.

The goal of this document is to outline strategies for the secure deployment of a microservices-based application by analyzing the implementation options for core state of practice features, considering the configuration options for architectural frameworks such as API gateway and service mesh, and countering microservices-specific threats.

1. INTRODUCTION, SCOPE, AND TARGET AUDIENCE

Application systems are increasingly developed and deployed using the microservices paradigm due to advantages such as agility, flexibility, scalability, and availability of tools for automating the underlying processes. However, the tremendous increase in the number of components in a microservices-based application system combined with complex network environments comprised of various interaction styles among components call for several core infrastructure features to be implemented either alone or bundled/packaged into architectural frameworks, such as API gateway and service mesh. The objective of this document is to perform an analysis of the implementation options for core features and configuration options for architectural frameworks as well as outline security strategies that counter microservice-specific threats.

1.1 Scope of this document

This document will not discuss the various tools used in the deployment of microservices-based application systems. Discussion of core features and architectural frameworks will be limited to highlighting issues relevant to secure implementation. The core focus is on the methodology to develop security strategies for microservices-based applications through the following three fundamental steps:

- Study of the technology behind microservices-based application systems focusing on design principles, basic building blocks, and associated infrastructure
- Focused review of the threat background specific to the operating environment of microservices
- Analysis of implementation options related to state of practice core features and configuration options related to architectural frameworks for developing security strategies

1.2 Target Audience

The target audience for the security strategies discussed in this document includes:

- Chief Security Officer (CSO) or Chief Technology Officer (CTO) of an IT department in a private enterprise or government agency who wishes to develop enterprise infrastructures to host distributed systems based on microservices architecture
- Application architects who wishes to design a microservices-based application system

1.3 Relationship to other NIST Guidance Documents

This is guidance document focuses on a class of application based on a specific architecture. However, since an essential architectural component—the microservice—can be implemented inside a container, the security guidance and recommendations related to application container technology may also be relevant security strategies for the application architecture discussed in this document. Such guidance includes:

- NIST SP 800-190, Application Container Security Guide
- NIST IR 8176, Security Assurance Requirements for Linux Application Container Deployments

1.4 Methodology and Organization

Since microservices-based application systems encompass diverse technologies (e.g., server virtualization, containers, cloud middleware), the focus here is on core features of this application class and the architectural frameworks that bundle or package them. The threat analysis approach involves taking a macro view of the entire deployment stack of microservices-based application systems and the layer at which these core features are located. The threats specific to those features are identified, and the overall approach for developing security strategies is to analyze the multiple implementations for core features and the architectural frameworks as well as ensure that those implementation options counter microservices-specific threats. The roadmap for the materials used in this methodology is as follows:

- Review of all state of practice core features that form the infrastructure for microservices (Section 2.6)
- Review of the layers in the deployment stack, location of the core features in those layers, and identification of microservices-specific threats (Section 3)
- Analysis of all different implementation options for these core features and outline of security strategies based on these implementation options for core features (Section 4)
- Review of all architectural frameworks that bundle several core features as a single product and outline security strategies based on the configuration options for architectural frameworks (Section 5)

A slightly more detailed summarization of the contents of the various sections in this document is as follows:

- Chapter 2 provides a high-level but expansive overview of microservices-based application systems, starting with a conceptual view followed by design principles, business drivers, building blocks, component interaction styles, state of practice core features, and architectural frameworks
- Chapter 3 provides a stack level view of the threat background and some threats that are specific to the microservices environment
- Chapter 4 contains analysis information pertaining to various state of practice core features for supporting a microservices-based application and outlines the security strategies for implementing the core features based on analysis of implementation options
- Chapter 5 contains analysis information pertaining to architectural frameworks that bundle core features needed in the infrastructure for microservices-based applications and outlines the security strategies for configuring the architectural frameworks

2. MICROSERVICES-BASED APPLICATION SYSTEMS – TECHNOLOGY BACKGROUND

In this section, the technology behind the development and deployment of a microservices-based application system will be described using the underlying design drivers or principles, the artifacts that constitute the building blocks, and the different ways the building blocks can be configured to produce different architectural options. This is not meant to be a comprehensive description of the technology but rather provide sufficient information about components and concepts to facilitate the identification of security threats and the development of secure implementation strategies for a microservices-based application system.

2.1 Microservices – a Conceptual View

A microservices-based application system consists of multiple components (microservices) that communicate with each other through synchronous remote procedure calls or an asynchronous messaging system. Each microservice typically implements one (rarely more) distinct business process or functionality (e.g., storing customer details, storing and displaying product catalog, customer order processing). Each microservice is a mini-application that has its own business logic and various adapters for carrying out functions such as database access and messaging. Some microservices would expose a RESTful API [1] that is consumed by other microservices or by the application's clients [2]. Other microservices might implement a web UI. At runtime, a microservice instance may be configured to run as a process in an application server, in a virtual machine (VM), or in a container.

Though a microservices-based application can be implemented purely as an enterprise application and not as a cloud service, its development is often identified as cloud-native application development with a service-based architecture, application programming interface (API)-driven communications, container-based infrastructure, and a bias for DevOps 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 due to the fact 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.

2.2 Microservices – Design Principles

The design of a microservice is based on the following drivers [4]:

- Each microservice must be managed, replicated, scaled, upgraded, and deployed independently of other microservices
- Each microservice must have a single function and operate in a bounded context (i.e., have limited responsibility and dependence on other services)

- All microservices should be designed for constant failure and recovery and must therefore be as stateless as possible
- One should reuse existing trusted services (e.g., databases, caches, directories) for state management

These drivers, in turn, result in the following design principles for a microservice:

- Autonomy
- Loose coupling
- Re-use
- Composability
- Fault tolerance
- Discoverability
- APIs alignment with business processes

2.3 Business drivers

Though the business drivers for deployment of microservices-based application systems are only marginally related to the theme of this document, it is useful to identify and state those that are relevant from the point of view of user and organizational behavior [5]:

- Ubiquitous access: users want access to applications from multiple client devices (e.g., browsers, mobile devices)
- Scalability: applications must be highly scalable to maintain availability in the face of increasing number of users and/or increased rate of usage from the existing user base
- Agile development: organizations want frequent updates to quickly respond to organizational (process and structural) changes and market demands

2.4 Building Blocks

Microservices-based applications (e.g., distributed enterprise or web applications [1]) are built using an architectural style or design pattern that is not restricted to any specific technology and is comprised of small independent entities (end points) that communicate with each other using lightweight mechanisms. These end points are implemented using well-defined APIs. There are several types of API endpoints, such as SOAP or REST (HTTP protocol). Each of the small independent entities provides a distinct business capability called a "service" and may have its own data store or repository. Access to these services is provided by various platforms or client types, such as web browsers or mobile devices, using a component called the "client." Together, the component services and the client form the complete microservices-based application system. The services in such a system may be classified as:

- Application-functionality services

- Infrastructure services (called "core features" in this document) implemented alone or bundled into architectural frameworks (e.g., API gateway, service mesh), including authentication and authorization, service registration and discovery, and security monitoring

In a microservices-based application system, each of the multiple, collaborative services can be built using different technologies. This promotes the concept of technical heterogeneity, which means that each service in a microservices-based application system may be written in a different programming language, development platform, or using different data storage technologies. This concept enables developers to choose the right tool or language depending on the type of service. Thus, in a single microservices-based application system, the constituting services may be built using different languages (e.g., Ruby, Golang, Java) and may be hosting different stores (e.g., document datastore, graphical DB, or multimedia DB). Each component service is developed by a team–a microservice or DevOps team—which provides all of the development and operational requirements for that service with a high degree of autonomy regarding development and deployment techniques so long as the service functionality or service contract is agreed upon [6].

Services in microservices are separately deployed on different nodes. The communication between them is transformed from a local function call to a remote call, which would affect system performance due to a high latency of network communication. Thus, a lightweight communication infrastructure is required.

Scaling can be applied selectively on those services that have performance bottlenecks due to insufficient CPU or memory resources, while other services can continue to be run using smaller, less expensive hardware. The functionality associated with such a service may be consumed in different ways for different purposes, thereby promoting reusability and composability. One example includes a customer database service, the contents of which are used both by shipping departments for preparing bills of lading and by accounts receivable or the billing department to send invoices.

2.5 Microservices – Interaction Styles

In monolithic applications, each component (i.e., a procedure or function) invokes another using a language-level call, such as a method or function. In microservices-based applications, each service is typically a process running in its own distinct network node that communicates with other services through an inter-process communication mechanism (IPC) [7]. Additionally, a service is defined using an interface definition language (IDL) (e.g., Swagger), resulting in an artifact called the application programming interface (API). The first step in the development of a service involves writing the interface definition, which is reviewed with client developers and iterated multiple times before the implementation of the service begins. Thus, an API serves as a contract between clients and services.

The choice of the IPC mechanism dictates the nature of the API [7]. The following table provides the nature of API definitions for each IPC mechanism.

Table 1: IPC Mechanisms and API Types

IPC Mechanism	Nature of API Definition
Asynchronous, message-based (e.g., AMQP or STOMP)	Made up of message channels and message types
Synchronous request/response (e.g., HTTP-based REST or Thrift)	Made up of URLs and request and response formats

There can be different types of message formats used in IPC communication: text-based and human-readable, such as JSON or XML, or of a purely machine-readable binary format, such as Apache Avro or Protocol buffers.

The principle of autonomy described earlier may call for each microservice to be a self-contained entity that delivers all of the functions of an application stack. However, for a microservices-based application that provides multiple business process capabilities (e.g., an online shopping application that provides business processes such as ordering, shipping, and invoicing), a component microservice is always dependent, in some fashion, on another microservice (e.g., data). In the context of our example, the shipping microservice is dependent upon "unfulfilled orders" data in the ordering microservice to perform its function of generating a shipping or bill of lading record. Hence, there is always the need to couple microservices while still retaining autonomy. The various approaches to creating the coupling, which are often dictated by business process and IT infrastructure needs, include interaction patterns, messaging patterns, and consumption modes. In this document, the term "interaction pattern" is used, and the primary interaction patterns are as follows.

Request-reply: Two distinct types of requests include queries for the retrieval of information and commands for a state-changing business function [2]. In the first type, a microservice makes a specific request for information or to take some action and functionally waits for a response. The purpose of the request for information is retrieval for presentation purposes. In the second type, one microservice asks another to take some action involving a state-changing business function (e.g., a customer modifying their personal profile or submitting an order). In the request-reply pattern, there is a strong runtime dependency between the two microservices involved, which manifests in the following two ways:

- One microservice can execute its function only when the other microservice is available
- The microservice making the request must ensure that the request has been successfully delivered to the target microservice

Because of the nature of communication in the request-reply protocol, a synchronous communication protocol, such as HTTP, is used. If the microservice is implemented with a REST API, the messages between the microservices become HTTP REST API calls. The REST APIs are often defined using a standardized language, such as RAML (RESTful API Modeling Language), which was developed for microservice interface definition and publication. HTTP is a blocking type of communication wherein the client that initiates a request can continue its task only when it receives a response.

Publish-Subscribe: This pattern is used when microservices need to collaborate for the realization of a complex business process or transaction. This is also called a business domain event-driven approach or domain event subscription approach. In this pattern, a microservices registers itself or subscribes to business domain events (e.g., interested in specific information or being able to handle certain requests), which are published to a message broker through an event-bus interface. These microservices are built using event-driven APIs and use asynchronous messaging protocols, such as Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), and Kafka Messaging, which enable support for notifications and subscriptions. In asynchronous protocols, the message sender does not typically wait for a response but simply sends the message to a message agent (e.g., RabbitMQ queue). One of the use cases for this approach is the propagation of data updates to multiple microservices based on certain events [8].

2.6 Microservices – State of the Practice Core Features

The criticality of the communication infrastructure in a microservices-based application environment calls for several sophisticated capabilities to be provided as core features in many deployments. As already stated, many of these features can be implemented either stand-alone or bundled together in architectural frameworks such as API gateway or service mesh. Even within the API gateway, these features can be implemented through service composition or direct implementation within the code base. These features include but are not limited to authentication, access control, service discovery, load balancing, response caching, application-aware health checks, and monitoring [2]. A brief description of these features [5] includes:

- Authentication and access control – The infrastructure platform can be leveraged to centralize enforcement of authentication and access control for all downstream microservices, eliminating the need to provide authentication and access control for each of the individual services. Authentication and access policy may vary depending on the type of APIs exposed by microservices—some may be public APIs; some may be private APIs; and some may be partner APIs, which are available only for business partners.
- 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 containers, which may be assigned dynamic IP addresses, especially when they are hosted in an IAAS or SAAS cloud service.
  (c) The number of instances associated with a service can vary based on the load using features such as autoscaling.
- Security monitoring and analytics – 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 with analytics capabilities in addition to routine logging features.

An API gateway is generally needed for implementing the following core features:

- Optimized endpoint – This involves several capabilities.

a) *Request and response collapsing*: Most business transactions will involve calls to multiple microservices, often in a pre-determined sequence. An API gateway can simplify the situation for clients by exposing an endpoint that will automatically make all the needed multiple requests (calls) and return a single, aggregated response to the client.
b) *API Transformation*: The API gateway can provide a public interface to the client which is different from the individual APIs it has to consume or the program calls it has to make to cater to a given request. This feature is called API transformation and enables:
   i) Changing the implementation and even the API interface for individual microservices
   ii) Transitioning from an initial, monolithic application to a microservices-based application by enabling continued access to clients through the API gateway while progressively splitting the monolithic application, creating microservice APIs in the background, and changing the API transformation configuration accordingly
c) *Protocol Translation*: Calls from clients to microservices endpoints may be in web protocols, such as HTTPS, while microservices communicate among themselves using synchronous protocols, such as RPC/Thrift, or asynchronous protocols, such as AMQP. The necessary protocol translation in client requests is typically carried out by the API gateway.

- Circuit breaker – This is a feature to set a threshold for the failed responses to an instance of a microservice and cut off proxying requests to that instance when the failure is above the threshold. This avoids the possibility of a cascaded failure, allows time to analyze logs, implement the necessary fix, and push an update for the failing instance.
- 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.
- Rate limiting (throttling) – The rate of requests coming into a service must be limited to ensured 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 since the API gateway 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.

2.7 Microservices – Architectural Frameworks

The two main architectural frameworks for bundling or packaging core features that primarily ensure reliable, resilient, and secure communication in a microservices-based application are:

- API gateway, augmented with or without micro gateways
- Service mesh

The role of these frameworks in the operating environment of a microservices-based application system are given in Table 2 below [4]:

Table 2: Role of Architectural Frameworks in Microservices Operations

Architectural Framework	Role in the Overall Architecture
API gateway, augmented with or without micro gateways	Used for controlling north-south and east-west traffic (the latter using micro gateways); micro gateways are deployed when microservices are implemented in web/application servers
Service mesh	Deployed for purely east-west traffic when microservices are implemented using containers but can also be used in situations where microservices are housed in VMs or application servers

2.7.1 API Gateway

The API gateway is a popular architectural framework for microservices-based application systems. Unlike a monolithic application where the endpoint may be a single server, a microservices-based application consists of multiple fine-grained endpoints. Direct communication of clients to multiple endpoints results in too many point-to-point connections. Hence, it makes sense to provide a single entry point for all clients to multiple component microservices of the application. This is the underlying objective behind the API gateway architecture. The primary function of the API gateway is to support clients with different form factors (e.g., browser, mobile device) and functional requirements. The core features of the API gateway are request routing, composition, and protocol translation (i.e., translation between web protocols, such as HTTP and WebSocket, and web-unfriendly protocols that are used internally, such as AMQP and Thrift binary RPC). All requests from clients first go through the API gateway, which then routes requests to the appropriate microservice. The API gateway will often handle a request by invoking multiple microservices and aggregating the results.

The multiple APIs or microservices accessible through the API gateway can be specified as part of the input port definition of the gateway (e.g., mobileAPI or MobileService) or be specified dynamically through a deploy operation of the API gateway service with a request parameter that contains the name of the service that should be embedded with the requested service [9]. Thus, the API gateway, located between clients and microservices, represents a pattern wherein a proxy aggregates multiple services. Many API gateway implementations can support APIs written in different languages, such as Jolie, JavaScript, or Java.

Since the API gateway is the entry point for microservices, it should be equipped with the necessary infrastructure services (in addition to its main service of request shaping), such as service discovery, authentication and access control, load balancing, caching, providing custom APIs for each type of client, application-aware health checks, service monitoring, and circuit breakers. These additional features may be implemented in the API gateway in two ways:

- By composing the specific services developed for respective functionality (e.g., service registry for service discovery)
- Implementing these functionalities directly inside the codebase that utilizes the API gateway

Gateway implementations

To prevent the gateway from having too much logic to handle request shaping for different client types, it is divided into multiple gateways [8]. This results in a pattern called backends for frontends (BFF). In BFF, each client type is given its own gateway (e.g., web app BFF, mobile app BFF) as a collection point for service requests. The respective backend is closely aligned with the corresponding front end (client) and is typically developed by the same team.

API management for a microservices-based application can be implemented through either a monolithic API gateway architecture or a distributed API gateway architecture. In the monolithic API gateway architecture, there is only one API gateway that is typically deployed at the edge of the enterprise network (e.g., DMZ) and provides all services to the API at the enterprise level. In the distributed API gateway architecture, there are multiple instances of microgateways, which are deployed closer to microservice APIs [10]. A microgateway is typically a low footprint, scriptable API gateway that can be used to define and enforce customized policies and is therefore suitable for microservices-based applications, which must be protected through service-specific security policies.

The microgateway is typically implemented as a stand-alone container using development platforms such as Node.js. It is different from a sidecar proxy of the service mesh architecture (refer to Section 2.7.2), which is implemented at the API endpoint itself. The security policies in a microgateway are encoded using JSON format and input through a graphical policy management interface. The microgateway should contain policies for both application requests and responses. Since policies and their enforcement are implemented as a container, they are immutable and thus provide a degree of protection against accidental and unintended modifications that may result in security breaches or conflicts, since any security policy update requires redeployment of the microgateway. It is essential that the microgateway deployed for any microservice instance communicate with service registry and monitoring modules to keep track of the operational status of the microservice it is designed to protect.

2.7.2 Service Mesh

A service mesh is a dedicated infrastructure layer that facilitates 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, node 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 OSI model (e.g., TCP/IP) and addresses the service's session layer (Layer 5 of the OSI model) concerns, eliminating the need to address them through application code [11]. A service mesh conceptually has two modules—the data plane and the control plane. The data plane carries the application request traffic between service instances though service-specific proxies. The control plane configures the data plane, provides a point of aggregation for telemetry, and provides APIs for modifying the behavior of the network through various features, such as load balancing, circuit breaking, or rate limiting.

Service meshes create a small proxy server instance for each service within a microservices application. This specialized proxy car is sometimes called a "sidecar proxy" in service mesh parlance [12]. The sidecar proxy forms the data plane, while the runtime operations needed for enforcing security (access control, communication-related) are enabled by injecting policies (e.g., access control policies) into the sidecar proxy from the control plane. This also provides the flexibility to dynamically change policies without modifying the microservices code.

2.8 Comparison with Monolithic Architecture

To fully compare the microservice architecture with the monolithic architecture used for all legacy applications, it is necessary to compare the features of applications developed using these architectural styles as well as provide an example of an application under both architectures for a specific business process. A detailed discussion involving these aspects is provided in Appendix A.

2.9 Comparison with Service-Oriented Architecture (SOA)

The architectural style of microservices shares many similarities with service-oriented architecture (SOA) due to the following common technical concepts [13]:

- Services – The application system provides its various functionalities through self-contained entities or artifacts called services that may have other attributes such as being visible or discoverable, stateless, reusable, composable, or have technological-diversity
- Interoperability – A service can call any other service using artifacts such as an enterprise service bus (ESB) in the case of SOA or through a remote procedural call (RPC) across a network as in the case of a microservices environment
- Loose coupling – There is minimal dependency between services such that the change in one service does not require a change in another service

In spite of the three common technical concepts described above, technical opinion on the relationship between an SOA and microservices environment falls along the following three lines [13]:

- Microservices are a separate architectural style
- Microservices represent one SOA pattern
- Microservice is a refined SOA

The most prevalent opinion is that the differences between SOA and microservices do not concern the architectural style except in its concrete realization, such as development or deployment paradigms and technologies [2].

2.10 Advantages of Microservices

- For large applications, splitting the application into loosely coupled components enables independence between the developer teams assigned to each component. Each team can then optimize by choosing its own development platform, tools, language, middleware, and hardware based on their appropriateness for the component being developed.
- Each of the components can be scaled independently. The targeted allocation of resources results in maximum utilization of resources.
- If components have HTTP RESTful interfaces, implementation can be changed without disruption to the overall function of the application as long as the interface remains the same.
- The relatively smaller codebase involved in each component enables the development team to produce updates more quickly and provide the application with the agility to respond to changes in business processes or market conditions.
- The loose coupling between the components enables containment of the outage of a microservice such that the impact is restricted to that service without a domino effect on other components or other parts of the application.
- When components are linked together using an asynchronous event-handling mechanism, the impact of a component's outage is temporary since the required functions will automatically execute when the component begins running again, thus maintaining the overall integrity of the business process.
- By aligning the service definition to business capabilities (or by basing the decomposition logic for the overall application functionality based on business processes or capabilities), the overall architecture of the microservices-based system is aligned with the organizational structure. This promotes agile response when business processes associated with an organizational unit change and consequently require that associated service to be modified and deployed.

2.11 Disadvantages of Microservices

- Multiple components (microservices) must be monitored instead of one single application. A central console is needed to obtain the status of each component and the overall state of the application. Therefore, an infrastructure must be created with distributed monitoring and centralized viewing capabilities.
- The presence of multiple components creates the availability problem since any component may cease functioning at any time.
- A component may have to call the latest version of another component for some clients and call the previous version of the same component for another set of clients (i.e., version management).
- Running an integration test is more difficult since a test environment is needed wherein all components must be working and communicating with each other.

3. MICROSERVICES – THREAT BACKGROUND

The threat background for a microservices-based application system should be treated as a continuation of the technology background provided in Section 2. The following approach has been adopted to review the threat background:

- Consider all layers in the deployment stack of a typical microservices-based application and when identifying typical potential threats at each layer
- Identify the distinct set of threats exclusive to microservices-based application systems

3.1 Review of Threat Sources Landscape

Six layers are present in the deployment stack of a typical microservices-based application as suggested in [13]: hardware, virtualization, cloud, communication, service/application, and orchestration. This document considers these layers to be threat sources, and several of the security concerns affiliated with them are described below to provide an overview of the threat background in a microservices-based application. It is important to remember that many of the possible threats are common to other application environments and not specific to a microservices-based application environment.

- Hardware layer – Though hardware flaws, such as Meltdown and Spectre [8], have been reported, such threats are rare. In the context of this document, hardware is assumed to be trusted, and threats from this layer are not considered.
- Virtualization layer: In this layer, threats to microservices or hosting containers originate from compromised hypervisors and the use of malicious or vulnerable container images and VM images. These threats are addressed in other NIST documents and are therefore not discussed here.
- Cloud environment – Since virtualization is the predominant technology used by cloud providers, the same set of threats to the virtualization layer applies. Further, there are potential threats within the networking infrastructure of the cloud provider. For example, hosting all microservices within a single cloud provider may result in fewer network-level security controls for inter-process communication as opposed to controls for communication between external clients and the microservices hosted within the cloud. Security threats within a cloud infrastructure are considered in several other NIST documents and are therefore not addressed here.
- Communication layer – This layer is unique to microservices-based applications due to the sheer number of microservices, adopted design paradigms (loose coupling and API composition), and different interaction styles (synchronous or asynchronous) among them. Many of the core features of microservices pertain to this layer, and the threats to these core features are identified under microservices-specific threats in Section 3.2.
- Service/application layer – In this layer, threats are the results of malicious or faulty code. As this falls under secure application development methodologies, it is outside of the scope of this document.
- Orchestration layer – An orchestration layer may come into play if the microservices implementation involves technologies such as containers. The threats in this layer pertain to the

subversion of automation or configuration features, especially related to scheduling and clustering of servers, containers, or VMs hosting the services, and are therefore beyond the scope of this document.

3.2 Microservices-specific Threats

Most state-of-practice core features refer to the communication layer in the deployment stack of microservices-based applications. Hence, the overall security strategies for microservices-based applications should involve choosing the right implementation options, identifying the architectural frameworks packaging those core features, identifying microservice-specific threats, and providing coverage for countering those threats in the implementation options.

3.2.1 Service Discovery Mechanism Threats

The basic functions in a service discovery mechanism are:

- Service registration and de-registration
- Service discovery

The potential security threats to the service discovery mechanism include:

- Registering malicious nodes within the system, redirecting communication to them, and subsequently compromising service discovery
- Corruption of the service registry database leading to redirection of service requests to wrong services and resulting in denial of services; also, redirection to malicious services resulting in compromise of the entire application system

3.2.2 Botnet Attacks

Unlike monolithic applications, wherein calls to a functional module of the application originate from a local procedure call or through a local data structure (i.e., sockets), calls to an API in a microservices-based application always originate from a program, not a direct client or user invocation), many of them from a remote program across the network. This exposes a microservices API to a multitude of botnets, which can vary based on the type of damage it inflicts (e.g., credential stuffing/abuse, takeover of accounts, page scraping, harvesting data, denial of service).

3.2.3 Cascading Failure

The presence of multiple components in a microservices-based application enhances the probability of a failure of a service. Though the components are designed to be loosely coupled from the point of view of deployment, there is a logical or functional dependency since many business transactions require the execution of multiple services in sequence to deliver the required outputs. Therefore, if a service that is upstream in the processing logic of a business transaction fails, other services that depend upon it may become unresponsive as well. This phenomenon is known as cascading failure.

4. SECURITY STRATEGIES FOR IMPLEMENTATION OF CORE FEATURES AND COUNTERING THREATS

Security strategies for the design and deployment of microservices-based application systems will span the following:

Analysis of implementation options for core features:

a) Identity and access management
b) Service discovery
c) Secure communication protocols
d) Security monitoring
e) Resiliency or availability improvement techniques
f) Integrity assurance improvement techniques

Countering microservices-specific threats:

a) Threats to service discovery mechanism
b) Botnet attacks
c) Cascading failures

Note that service discovery is a core feature in microservices, and analysis of the implementation options will also take into consideration threats to service discovery mechanisms. Similarly, implementation options for resiliency or availability improvement will also address the counter measures for cascading failures. As such, there will not be separate security strategies for these items.

4.1 Strategies for Identity and Access Management

Since microservices are packaged as APIs, the initial form of authentication to microservices involves the use of API keys (cryptographic). Authentication tokens encoded in SAML or through OpenID connect under the OAuth 2.0 framework provide an option for enhancing security [14]. Additionally, a centralized architecture for provisioning and enforcement of access policies governing access to all microservices is required due to the sheer number of services, the implementation of services using APIs, and the need for service composition to support real-world business transactions (e.g., customer order processing and shipping). A standardized, platform-neutral method for conveying authorization decisions through a standardized token (e.g., JSON web tokens (JWT), which are OAuth 2.0 access tokens encoded in JSON format [15]) is also required since each of the microservices may be implemented in a different language or platform framework. Policy provisioning and computation of access decisions require the use of an authorization server.

The disadvantage to implementing access control policies at the access point of each microservice is that additional effort is required to ensure that cross-cutting (common) policies applicable to all microservice APIs are implemented uniformly. Any discrepancy in security policy implementation among APIs will have security implications for the entire microservices-based application. Further, the footprint for implementing access control in each microservices node can result in performance issues in some nodes. Since multiple microservices nodes collaborate to perform a transaction, performance problems associated with any node can quickly cascade across multiple services. The strategies for secure identity and access management to microservices are outlined below.

Security strategies for authentication (MS-SS-1):

- Authentication to microservices APIs that have access to sensitive data should not be done simply by using API keys. Rather, an additional form of authentication should also be used.
- Every API Key that is used in the application should have restrictions specified both for the applications (e.g., mobile app, IP address) and the set of APIs where they can be used.

Security strategies for access management (MS-SS-2):

- Access policies to all APIs and their resources should be defined and provisioned centrally to an access server
- The access server should be capable of supporting fine-grained policies
- Access decisions from the access server should be conveyed to individual and sets of microservices through standardized tokens encoded in a platform-neutral format (e.g., OAuth 2.0 token encoded in JSON format)
- The scope in authorization tokens (extent of permissions and duration) should be carefully controlled; for example, in a request for transaction, the allowed scope should only involve the API endpoints that must be accessed for completing that transaction
- It is preferable to generate tokens for performing authentication instead of passing credentials to the API endpoints since any damage would be limited to the time that the token is valid; authentication tokens should be cryptographically signed or hashed tokens

4.2 Strategies for Service Discovery Mechanism

Microservices may have to be replicated and located anywhere in the enterprise or cloud infrastructure for optimal performance and load balancing reasons. In other words, services could be frequently added or removed and dynamically assigned to any network location. Hence, it is inevitable in a microservices-based application architecture to have a service discovery mechanism, which is typically implemented using the service registry. The service registry service is used by microservices that are coming online to publish their locations in a process called service registration and is also used by microservices seeking to discover registered services. The service registry must therefore be configured with confidentiality, integrity, and availability considerations.

In service-oriented architectures (SOA), service discovery is implemented as part of the centralized enterprise service bus (ESB). However, in microservices architecture—where the business functions are packaged and deployed as services within containers and communicate with each other using API calls—it is necessary to implement a lightweight message bus that can implement all three interaction styles mentioned in Section 2.5. Additionally, alternatives to the ways in which service registry service can be implemented span two dimensions: (a) the way clients access the service registry service and (b) centralized versus distributed service registry. Clients can access the service registry service using two primary methods: client-side discovery pattern and server-side discovery pattern [16].

Analysis of the client-side service discovery pattern

The client-side option consists of building registry-aware clients. The client queries the service registry for the location of all services needed to make requests. It then contacts the target service directly. Though simple, this implementation option for service discovery requires the discovery logic (querying the service registry) to be implemented for each programming language and/or framework that is used for client implementations.

Analysis of the service-side service discovery pattern

The service-side discovery has two implementations: one pattern delegates the discovery logic to a dedicated router service set at a fixed location, while the other utilizes a server in front of each microservice with the functionality of a dynamic DNS-resolver. In the dedicated router option, the client makes all service requests to this dedicated router service, which in turn queries the service registry for the location of the client-requested service and forwards that request to the discovered location. This removes the tight coupling between an application service and an infrastructure service such as the service registry service. In the DNS resolver pattern, each microservice completes its own service discovery using its built-in DNS resolver to query the service registry. The DNS resolver maintains a table of available service instances and their endpoint locations (i.e., IP addresses). To keep the table up to date, the asynchronous, nonblocking DNS resolver queries the service registry regularly—perhaps every few seconds—using DNS SRV records for service discovery. Since the service discovery function through the DNS resolver runs as a background task, the endpoints (URLs) for all peer microservices are instantly available when a service instance needs to make a request [2].

A good strategy would be to use a combination of the service-side service discovery pattern and the client-side service discovery pattern [16]. The former can be used for providing access to all public APIs, while the latter can allow clients to access all cluster-internal interactions.

Centralized versus distributed service registry

In a centralized service registry implementation, all services wishing to publish their service register at a single point, and all services seeking these services use the single registry to discover them. The security disadvantage of this pattern is the single point of failure [17]. However, data consistency will not be an issue. In the decentralized service registry, there may be multiple service registry instances, and services can register with any of the instances. In the short term, the disadvantage is that there will be data inconsistency between the various service registries. Eventually, consistency among these various instances of service registry is achieved either through broadcasting from one instance to all others or by propagation from one node to all others via attached data in a process called piggybacking.

Regardless of the pattern used for service discovery, secure deployment of service discovery functions should meet the following service registry configuration requirements.

Security strategies for service registry configuration (MS-SS-3)

- Service registry capabilities should be provided through a cluster of servers with a configuration that can perform frequent replication.

- Service registry clusters should be in a dedicated network where no other application service is run.
- Communication between an application service and a service registry should occur through a secure communication protocol such as HTTPS or TLS.
- Service registry should have validation checks to ensure that only legitimate services are performing the registration, refresh operations, and database queries to discover services.
- The bounded context and loose coupling principle for microservices should be observed for the service registration/deregistration functions. In other words, the application service should not have tight coupling with an infrastructure service, such as a service registry service, and service self-registration/deregistration patterns should be avoided. When an application service crashes or is running but unable to handle requests, its inability to perform deregistration affects the integrity of the whole process. Therefore, registration/deregistration of an application service should be enabled using a third-party registration pattern, and the application service should be restricted to querying the service registry for service location information as described under the client-side discovery pattern.
- If a third-party registration pattern is implemented, registration/deregistration should only take place after a health check on the application service is performed.
- Distributed service registry should be deployed for large microservices application, and care should be taken to maintain data consistency among multiple service registry instances.

4.3 Strategies for Secure Communication Protocols

Secure communication between clients and services (north-south traffic) and between services (east-west traffic) is critical for the operation of a microservices-based application. It is a good practice to build security features into infrastructure rather than application code, and several technologies have evolved with that objective.

However, certain strategies for security services—such as authentication or the establishment of secure connections—can be handled at the individual microservices nodes. For example, in the fabric model, each microservice instance has the capability to function as an SSL client and SSL server (i.e., each microservice is an SSL/TLS endpoint). Thus, a secure SSL/TLS connection is possible for interservice or inter-process communication from an overall application perspective. These connections can be created dynamically (i.e., before each interservice request) or be created as a keep-alive connection. In the keep-alive connection scheme, a "service A" creates a connection after a full SSL/TLS handshake—the first time an instance of that service makes a request to an instance of a "service B." However, neither service instances terminate the connection after a response returns for that request from service B. Rather, the same connection is reused in future requests. The advantage of this scheme is that the costly overhead involved in performing the initial SSL/TLS handshake can be avoided during each request, and an existing connection can be reused for thousands of following interservice requests. Thus, a permanent secure interservice network connection is available for all instances of requests.

<u>Security strategies for secure communication (MS-SS-4)</u>

- Clients should not be configured to call its target services directly but rather to point to the single gateway URL

- Client-to-API-gateway communication should take place after mutual authentication and be encrypted (e.g., using mTLS protocol)
- Frequently interacting services should create keep-alive TLS connections

4.4 Strategies for Security Monitoring

Compared to monitoring a monolithic application which runs in a server (or some replicas for load balancing), a microservices-based system must monitor a large number of services, each running in different servers possibly hosted on heterogeneous application platforms. Further, any meaningful transaction in the system will involve at least two or more services.

Security strategies for security monitoring (MS-SS-5)

- An analytics engine analyzes the dependencies among the services and identifies nodes (services) and paths (network) that are bottlenecks
- A central dashboard displays the status of various services and the network segments that link them

4.5 Availability/Resiliency Improvement Strategies

In microservices-based applications, targeted efforts that improve the availability or resiliency of certain critical services are needed to enhance the overall security profile of the application. Some technologies that are commonly deployed include:

- Circuit breaker function
- Load balancing
- Rate limiting (throttling)

4.5.1 Analysis of Circuit Breaker implementation options

A common strategy for preventing or minimizing cascading failures involves the use of circuit breakers, which prohibits the delivery of data to the component (microservice) that is failing beyond a specified threshold. This is also known as the fail fast principle. Since the errant service is quickly taken offline, incidences of cascading failures are minimized while the errant component's logs are analyzed, required fixes are performed, and microservices are updated. There are three options for deploying circuit breakers [9]: directly inside the client, on the side of services, or in proxies that operate between clients and services.

Client-side circuit breaker option: In this option, each client has a separate circuit breaker for each external service that the client calls. When the circuit breaker in a client has decided to cut off calls to a service (called "open state" with respect to that service), no message will be sent to the service, and communication traffic in the network is subsequently reduced. Moreover, the circuit breaker functionality need not be implemented in the microservice, which frees valuable resources for efficient implementation of that service. However, locating the circuit breaker in the client carries two disadvantages from a security point of view. First, a great deal of trust must be placed in the client that the circuit breaker code executes properly. Second, the overall integrity of the operation is at risk since knowledge of the unavailability of the service is very much local to the client, a status that is determined based on the frequency of calls from that client to the service rather than on the combined response status received by all clients against that service.

<u>Server-side circuit breaker option</u>: In this option, an internal circuit breaker in the microservice processes all client invocations and decides whether it should be allowed to invoke the service or not. The security advantages of this option are that clients need not be trusted to implement the circuit breaker function, and since the service has a global picture of the frequency of all invocations from all clients, it can throttle requests to a level which it can conveniently handle (e.g., temporarily lighten the load).

<u>Proxy circuit breaker option</u>: In this option, circuit breakers are deployed in a proxy service, located between clients and microservices, which handles all incoming and outgoing messages. Within this, there may be two options: one proxy for each target microservice or a single proxy for multiple services (usually implemented in API gateway) that includes both client-side circuit breakers and service-side circuit breakers existing within that proxy. The security advantage of this option is that neither the client code nor the services code needs to be modified, which avoids trust and integrity assurance issues associated with both these categories of code as well as the circuit breaker function. This option also provides additional protections such as making clients more resilient to faulty services, and shielding services from cases in which a single client sends too many requests [9], resulting in some type of denial of service to other clients that use that service.

<u>Security strategies for implementing circuit breakers (MS-SS-6)</u>

- A proxy circuit breaker option should be deployed to limit the trusted component to the proxy. This avoids the need to place the trust on the clients and microservices (e.g., setting thresholds and cutting off requests based on the set threshold) since they are multiple components.

4.5.2 Strategies for Load Balancing

Load balancing is an integral functional module in all microservices-based applications, and its main purpose is to distribute loads to services. A service name is associated with a namespace that supports multiple instances of the same service. In other words, many instances of the same service would use the same namespace [17]. To balance the service load, the load balancer chooses one service instance in the request namespace using an algorithm such as the round-robin algorithm—a circular pattern to assign the request to a service instance.

<u>Security strategies for load balancing (MS-SS-7)</u>

- All programs supporting the load balancing function should be decoupled from individual service requests. For example, the program that performs health checks on services to determine the load balancing pool should run asynchronously in the background.
- When a DNS resolver is deployed in front of a source microservice to provide a table of available target microservice instances, it should work in tandem with the health check program to present a single list to the calling microservice.

4.5.3 Rate Limiting (Throttling)

The goal of rate limiting is to ensure that a service is not oversubscribed. That is, when one client increases the rate of requests, the service continues its response to other clients. This is achieved by setting a limit on how often a client can call a service within a defined window of time. When the limit is exceeded, the client—rather than receiving an application-related response—receives a notification that the allowed rate has been exceeded as well as additional data regarding the limit number and the time at which the limit counter will be reset for the requestor to resume receiving responses. Closely related to the concept of rate limiting is quota management or conditional rate limiting where limits are determined based on application requirements rather than infrastructure limitations or requirements.

Security strategies for rate limiting (MS-SS-8)

- Quotas or limits for application usage should be based on both infrastructure and application-related requirements.
- Limits should be determined based on well-defined API usage plans.

4.6 Integrity Assurance strategies

Integrity assurance requirements in the context of microservices-based applications arise under two contexts:

- When new versions of microservices are inducted into the system
- For supporting session persistence during transaction

Monitored induction of new releases: Whenever a newer version of a microservice is released, its induction must be a gradual process since (a) all clients may not be ready to use the new version, and (b) the behavior of the new version for all scenarios and use cases may not meet the business process expectation despite extensive testing. To address this situation, a technique called canary release is often adopted [4]. Under this technique, only a limited number of requests are routed to the new version after it is brought online, and the rest are routed to the existing operational version. After a period of observation provides assurance that the new version meets performance and integrity metrics, all of the requests are routed to the new version.

Security (integrity assurance) strategies for the induction of new versions of microservices (MS-SS-9):

- The traffic to both the existing version and the new version of the service should be routed through a central node, such as an API gateway, to monitor the total number of calls to the service.
- Security monitoring should cover nodes hosting both the existing and newer versions.
- Usage monitoring of the existing version should steadily increase traffic to the new version.
- The performance and functional correctness of the new version should be factors in increasing traffic to the new version.
- Client preference for the version (existing or new) should be taken into consideration while designing a canary release technique.

Session persistence: It is critical to send all requests in a client session to the same upstream microservice instance since clients execute a complete transaction through multiple requests to a specific service, and the target of all requests should be to the same upstream service instance in that session. This requirement is called session persistence. A situation that could potentially break this requirement is one wherein the microservice stores its state locally, and the load balancer handling individual requests forwards a request from an in-progress user session to a different microservice server or instance. One of the methods for implementing session persistence is sticky cookie. In this method, there is a mechanism to add a session cookie to the first response from the upstream microservice group to a given client, identifying (in an encoded fashion) the server that generated the response. Subsequent requests from the client include the cookie value, and the same mechanism uses it to route the request to the same upstream server [18].

Security (integrity assurance) strategies for handling session persistence (MS-SS-10):

- The session information for a client must be stored securely
- The artifact used for conveying the binding server information must be protected

4.7 Countering Botnet Attacks

Though it is impossible to protect against all types of botnets, microservice APIs must be provided with detection and prevention capabilities against credential-stuffing and credential abuse attacks. This is especially critical for those applications where each of the microservices are independently callable and carry their own sets of credentials. Credential abuse attacks can be detected using offline threat analysis or run-time solutions [19]. Detection of botnet attacks is provided by a dedicated bot manager product or as an add-on feature in web application firewalls (WAF).

Security strategies for preventing credential abuse and stuffing attacks (MS-SS-11):

- A run-time prevention strategy for credential abuse is preferable to offline strategy. A threshold for a designated time interval from a given location (e.g., IP address) for the number of login attempts should be established; if the threshold is exceeded, prevention measures must be triggered.
- A credential-stuffing detection solution has the capability to check user logins against the stolen credential database and warn legitimate users that their credentials have been stolen.

5. SECURITY STRATEGIES FOR ARCHITECTURAL FRAMEWORKS IN MICROSERVICES

The two main architectural frameworks considered in this document for microservices-based application systems are the API gateway and service mesh. The primary security considerations in the implementation of the API gateway involve choosing the right platform for hosting it, proper integration and configuration with enterprise-wide authentication and authorization frameworks, and securely leveraging the traffic flowing through it for security monitoring and analysis.

Security strategy for API gateway implementation (MS-SS-12):

- API gateway platform requirements: Since some microservices have multiple communication styles (i.e., synchronous and asynchronous), it is imperative that the API gateway that serves as the entry point for these services should support multiple communication protocols, and a high-performance webserver and reverse proxy should support its basic functional capabilities.
- Integrate API gateway with an identity management application to provision credentials before activating the API.
- When identity management is invoked through the API gateway, connectors should be provided for integrating with identity providers (IdPs).
- The API gateway should have a connector to an artifact that can generate an access token for the client request (e.g., OAuth 2.0 Authorization Server).
- Securely channel all traffic information to a monitoring and/or analytics application for detecting attacks (e.g., denial of service, malicious actions) and unearthing explanations for degrading performance.

Implementing a service mesh can help ensure that proper configuration parameters associated with various security policies are defined correctly in the control plane so that the intent of the security policies are met, and the service mesh alone does not introduce new vulnerabilities.

Security strategy for service mesh implementation (MS-SS-13):

- Provide policy support for designating a specific communication protocol between pairs of services and specifying the traffic load between pairs of services based on application requirements.
- Default configuration should always enable access control policies for all services.
- Avoid configurations that may lead to privilege escalation (e.g., the service role permissions and binding of the service role to service user accounts).

Appendix A: Differences between Monolithic Application and Microservices-based Application

A.1 Design and Deployment Differences

Conceptually, a monolithic architecture of an application involves generating one huge artifact that must be deployed in its entirety, while a microservices-based application contains multiple self-contained, loosely-coupled executables called services or microservices. The individual services can be deployed independently. In monolithic applications, any change to a certain functionality of the overall application will involve recompilation and, in some instances, re-testing of the whole application before being deployed again. However, in the case of microservices, only the relevant service is modified and redeployed since the independent nature of the services ensures that a change in one does not logically affect the functionality of another. In monolithic applications, any increase in workload due to an increase in the number of users or the frequency of application usage will involve allocating resources to the whole application, whereas in microservices, the increase in resources can be selectively applied to those whose performance is less than desirable, thus providing flexibility in scalability efforts.

Some monolithic applications may be constructed modularly but may not have semantic or logical modularity. Modular construction refers to how an application may be built from a large number of components and libraries that may have been supplied by different vendors, and some components (e.g., database) may be distributed across the network [17]. In such monolithic applications, the design and specification of APIs may be similar to that in a microservices architecture. However, the difference between such modularly designed monolithic applications (sometimes called a classic modular design) and a microservices-based application is that in the latter, the individual API is network-exposed and therefore independently callable and re-usable.

The differences between monolithic and microservices-based applications is summarized in Table A.1 below:

Table A.1: Logical Differences between Monolithic and Microservices-based Application

Monolithic Application	Microservices-based Application
Must be deployed as a whole	Independent or selective deployment of services
Change in a small part of the application requires re-deployment of the entire application	Only the modified services need to be re-deployed
Scalability involves allocating resources to the application as a whole	Each of the individual services can be selectively scaled up by allocating more resources
API calls are local	Network-exposed APIs enable independent invocation and re-usability

A.1.1 An Example Application to illustrate the design and deployment differences

The following example of a small, online retail application illustrates the design and deployment differences discussed above. The main functions of this application are:

- A module that displays the catalog of products offered by the retailer with pictures of the products, product numbers, product names, and the unit prices
- A module for processing customer orders by gathering information about the customer (e.g., name, address) and the details of the order (e.g., name of the product from the catalog, quantity, unit price) as well as creating a bin containing all the items ordered in that session
- A module for preparing the order for shipping, specifying the total bill of lading (i.e., the total package to be shipped, quantity of each item in the order, shipping preferences, shipping address)
- A module for invoicing the customer with a built-in feature for making payments by credit card or bank account

The differences in the design of this online retail application as a monolithic versus microservices-based are given in table below.

Table A.2: Differences in Application Construct between Monolithic and Microservices-based Application

Application Construct	Monolith	Microservices-based
Communication between functional modules	All communications are in the form of procedure calls or some internal data structures (e.g., socket). The module handling the order processing makes a procedural call to the module handling the shipping function and waits for successful completion (blocking type synchronous communication).	The shipping functionality and the order processing functionality are each designed as independent services. Communication takes place as an API call across the network using a web protocol. The order processing microservice can either (a) make a request-response call to the shipping microservice and wait for a response or (b) put the details of the order to be shipped in a message queue to be picked up asynchronously by the shipping microservice, which has subscribed to the event.
Handling changes or enhancements (e.g., invoicing module needs to be changed to accept debit cards)	The entire application must be recompiled and redeployed after making the necessary changes.	The invoicing function is designed as a separate microservice, so that service can simply recompiled and redeployed.
Scaling the application, allocation of increased	The order processing functionality involves longer	It is enough to allocate increased resources for

resources (e.g., order processing module needs to be allocated more resources to handle a larger load)	transaction times compared to shipping or invoicing functions. Vertical scaling that involves using servers with more memory or CPUs must be deployed for the entire application.	hardware where the order processing microservice is deployed. Also, the number of instances of order-processing microservices can be increased for better load balancing.
Development and deployment strategy	Development is handled by the development team which, after necessary testing by the QA team, transfers the task of deployment to an infrastructure team that oversees the allocation of suitable resources for deployment.	The complete lifecycle—from development to deployment—is handled by a single DevOps team for each microservice since it is a relatively small module with a single functionality and built-in a platform (e.g., OS, languages libraries) that is optimal for that functionality.

A.2 Run-time Differences

A monolithic application runs as a single computational node such that the node is aware of the overall system or application state. In a microservices environment, the application is designed as a set of multiple nodes that each provide a service. Since they operate without the need to coordinate with others, the overall system state is unknown to individual nodes. In the absence of any global information or global variable values, the individual nodes make decisions based on locally available information. The independence of the nodes means that failure of one node does not affect other nodes. Unlike monolithic applications where services may share database connections or a data repository, a microservice architecture may deploy a pattern wherein each service has its own data repository. In many situations, interaction between services may require a distributed transaction which, if not designed properly, may affect the integrity of the databases.

The runtime differences between monolithic and microservices applications and their implications are summarized in Table A.2 below.

Table A.3: Architectural Differences between Monolithic and Microservices-based Application

Monolithic Application	Microservices-based Application
Runs as a single computational node; overall state information fully known	Designed as a set of multiple nodes, each providing a service; overall system state is unknown to individual nodes
Designed to make use of global information or values of global variables	Individual nodes make decisions based on locally available information
Failure of the node means crash of the application	Failure of one node should not affect other nodes

Figure A.1: Online Shopping Application – Monolithic Architecture

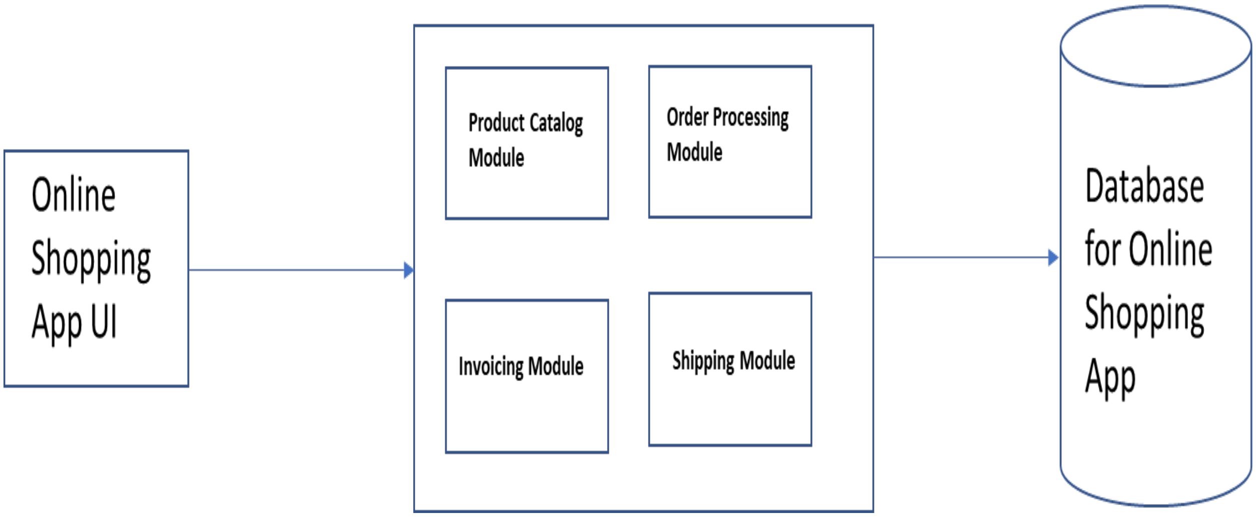

Figure A.1: Online Shopping Application – Monolithic Architecture

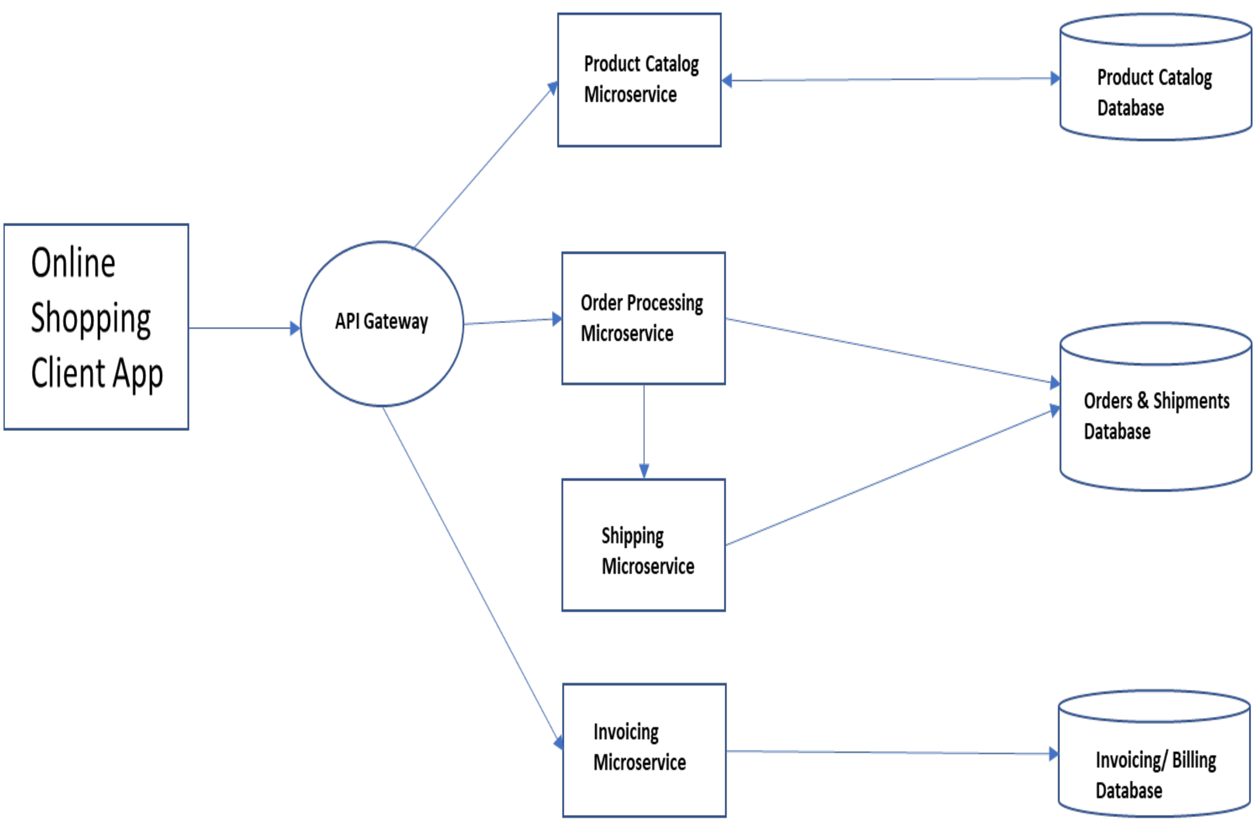

Figure A.2: Online Shopping Application – Microservices Architecture

Appendix B: Traceability of Security Strategies to Microservices Architectural Features

Security Strategy Identifier	Security Strategy	Microservices Core Feature/ Architectural framework
MS-SS-1	• Authentication to microservice APIs that have access to sensitive data should not be done simply by using API keys; an additional form of authentication should also be used. • Every API Key that is used in the application should have restrictions specified both for applications (e.g., mobile app, IP address) and the set of APIs where they can be used	Authentication
MS-SS-2	• Access policies to all APIs and their resources should be defined and provisioned centrally to an access cover • The access server should be capable of supporting fine-grained policies • Access decisions from the access server should be conveyed to individual and sets of microservices through standardized tokens encoded in a platform-neutral format (e.g., OAuth 2.0 token encoded in JSON format) • The scope in authorization tokens (i.e., extent of permissions and duration) should be carefully controlled; for example, in a request for a transaction, the allowed scope should only involve the API endpoints that must be accessed to complete that transaction • It is preferable to generate tokens for performing authentication instead of passing credentials to the API endpoints since potential damage will be limited to the time that the token is valid instead of the long-term damage due to compromised credentials; authentication tokens should be cryptographically signed or hashed	Access management

Security Strategy Identifier	Security Strategy	Microservices Core Feature/ Architectural framework
MS-SS-3	• Service registry capability should be provided through a cluster of servers with a configuration that can perform frequent replication • Service registry clusters should be in a dedicated network where no other application services are run • Communication between an application service and a service registry should be through a secure communication protocol, such as HTTPS/TLS • Service registry should have validation checks to ensure that only legitimate services are performing the registration and refresh operations or querying its database to discover services • The bounded context and loose coupling principle for microservices should be observed for the service registration/deregistration function; the application service should not have tight coupling with an infrastructure service, such as service registry service, and the service self-registration/deregistration pattern should be avoided. Moreover, when an application service crashes or is running but not in a position to handle requests, it cannot perform deregistration, thus affecting the integrity of the whole process. Registration or deregistration of an application service should be enabled using a third-party registration pattern, and the application service should be restricted to simply querying the service registry for service location information as described in the client-side discovery pattern. • If third-party registration pattern is implemented, registration/deregistration should only take place after performing a health check on the application service • Distributed service registry should be deployed for large microservices applications, and care should be taken to maintain data consistency among multiple service registry instances	Service registry configuration
MS-SS-4	• Clients should not be configured to call their target services directly but rather be configured to point to the single gateway URL • Client to API gateway communication should take place after mutual authentication and be encrypted (e.g., using mTLS protocol) • Frequently interacting services should create keep-alive TLS connections	Secure communication

Security Strategy Identifier	Security Strategy	Microservices Core Feature/ Architectural framework
MS-SS-5	• Analytics engine that analyzes dependencies among the services and identifies nodes (services) and paths (network) that are the bottlenecks • A central dashboard that displays the status of various services and the network segments linking them	Security monitoring
MS-SS-6	• Proxy circuit breaker option should be deployed to limit the trusted component to be the proxy, which avoids the need to place the trust on the clients and microservices (setting thresholds and cutting off requests based on the set threshold) since they are multiple components	Implementing circuit breaker
MS-SS-7	• The load balancing function should be decoupled from individual service requests; for example, the program that performs health checks on the services to determine the load balancing pool should run asynchronously in the background • When a DNS resolver is deployed in front of a source microservice to provide a table of available target microservice instances, it should work in tandem with the health check program to present a single list to the calling microservice	Implementing load balancing
MS-SS-8	• Quotas or limits for application usage should be based on both infrastructure and application-related requirements • Limits should be determined based on well-defined API usage plans	Rate limiting (throttling)
MS-SS-9	• Traffic to both the existing version and the new version of the service should be routed through a central node, such as an API gateway, to monitor the total number of calls to the service • Security monitoring should cover nodes hosting both the existing and newer versions	Induction of new versions of microservice
MS-SS-10	• Session information for a client must be stored securely • The artifact used for conveying the binding server information must be protected	Handling session persistence
MS-SS-11	• A run-time prevention strategy for credential abuse is preferable to an offline strategy; a threshold for a designated time interval from a given location (e.g., IP address) for the number of login attempts should be set up, and prevention measures must be triggered if the threshold is exceeded • A credential-stuffing detection solution with the capability to check user logins against the stolen credential database and warn the legitimate users that their credentials have been stolen	Preventing credential abuse and stuffing attacks

Security Strategy Identifier	Security Strategy	Microservices Core Feature/ Architectural framework
MS-SS-12	• Channel all traffic information to a monitoring and/or analytics application for detecting attacks (e.g., denial of service, malicious threats) through unusual usage patterns or deteriorating response times • Integrate API gateway with an identity management application to provision credentials before activating the API • API gateway platform requirements: since some microservices have multiple communication styles (i.e., synchronous and asynchronous), it is imperative that the API gateway which serves as the entry point for these services support multiple communication styles; a high-performance webserver and reverse proxy should support its basic functional capabilities • When identity management is invoked through an API gateway, connectors should be provided for integrating with IdPs • API gateway should have a connector to an artifact that can generate an access token for the client request (e.g., OAuth 2.0 Authorization Server)	API gateway configuration
MS-SS-13	• Policy support should be enabled for: (a) designating a specific communication protocol between pairs of services and (b) specifying the traffic load between pairs of services based on application requirements • Default configuration should always be to enable access control policies for all services • Avoid configurations that may lead to privilege escalation (e.g., the service role permissions and binding of the service role to service user accounts)	Service mesh configuration

Appendix C: References

[1] Sill, A. (2016). The design and architecture of microservices. IEEE Cloud Computing.

[2] Richardson, C. and Smith, F. (2016). Microservices: From design to deployment. NGINX Inc.

[3] TechTarget. (n.d.). Comparing two schools of application development: Traditional vs. Cloud-Native. Retrieved from https://searchcloudcomputing.techtarget.com/PaaS/Comparing-Two-Schools-of-Application-Development-Traditional-vs-Cloud-Native.

[4] Richardson, C. (2015). Building microservices: Using an API gateway. Retrieved from https://www.nginx.com/blog/building-microservices-using-an-api-gateway/.

[5] Palladino, M. (2016). Microservices and API gateway, Part 1: Why an API gateway? Retrieved from www.nginx.conf. Accessed March 2019.

[6] Jander, K., Braubach, L., and Pokahr, A. (2018). Defense in-depth and role authentication for microservice systems. Proceedings of the 9th International Conference on Ambient Systems, Networks and Technologies.

[7] Richardson, C. (2015). Building microservices: Inter-process communication in a microservices architecture. Retrieved from https://www.nginx.com/blog/building-microservices-inter-process-communication/.

[8] Guidelines for Adopting Frontend Architectures and Patterns in Microservices-Based Systems, Proceedings of 11th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), Paderborn, Germany, Sept 4-8, 2017.

[9] Montesi, F. and Weber, J. (2016). Circuit breakers, discovery, and API gateways in microservices.

[10] O'Neill, M. and Malinverno, P. (2018). Critical capabilities for full life cycle API management. Gartner Report 334223.

[11] Calcote, L. (2018). The enterprise path to service mesh architectures. O'Reilly.

[12] Twistlock. (n.d.) Securing the service mesh: Understanding the value of service meshes, why Istio is rising in popularity, and exploring official Twistlock compliance checks for Istio. Retrieved from https://www.twistlock.com/resources/securing-service-mesh-istio-compliance-checks/.

[13] Yarygina, T. and Bagge, A.H. (2018). Overcoming security challenges in microservice architecture. Proceedings of 2018 IEEE Symposium on Service-Oriented System Engineering.

[14] OpenID. (n.d.). Welcome to OpenID Connect. Retrieved from https://openid.net/connect/. Accessed March 2019.

[15] Internet Engineering Task Force (IETF). (2012). The OAuth 2.0 authorization framework. Retrieved from https://tools.ietf.org/html/rfc6749. Accessed March 2019.

[16] Circuit Breakers, Discovery, and API Gateways in Microservices

[17] T. Yarygina and A.H. Bagge, "Overcoming Security Challenges in Microservice Architecture", Proceedings of 2018 IEEE Symposium on Service-Oriented System Engineering, 2018.

[18] NGINX. (n.d.). High-performance load balancing: Scale out your applications with NGINX and NGINX Plus. Retrieved from https://www.nginx.com/products/nginx/load-balancing/#session-persistence.

[19] Katz, O. (2017). Improving credential abuse threat mitigation. Retrieved from https://blogs.akamai.com/2017/01/improving-credential-abuse-threat-mitigation.html. Accessed March 2019.