Best Practices in Implementing a Secure Microservices Architecture

Integrating Microservices Security Considerations into the Engineering of Trustworthy Secure Systems

The permanent and official location for Cloud Security Alliance Application Containers and Microservices research is:
https://cloudsecurityalliance.org/research/working-groups/containerization/

Acknowledgments

Editor:

Anil Karmel

Team Leaders/Authors:

Marina Bregkou
Aradhna Chetal
Mark Yanalitis
Michael Roza

Authors:

Yeu Wen Mak
Vishwas Manral
Ramaswamy Chandramouli
Shachaf Levy
James Turrell
William Gafford
Alex Rebo
Jane Odero Greene
Atul Chaturvedi
Michael Green
Amit Maheshwari
Vrettos Moulos
Ricardo Ferreira

CSA Staff:

Hillary Baron
Marina Bregkou

Reviewers:

Marina Bregkou
Rohit Sansiya
Michael Roza
Vrettos Moulos
Prabath Siriwardena
Suhas Bhat
Carlos Villavicencio Sánchez
Ricardo Ferreira
David Cantrell

Table of Contents

1.0 Overview of the Microservices Architecture

This chapter provides the background for the evolution of the microservices architecture, its advantages compared with the previous architectures and the new challenges posed in terms of configuration and security.

The earliest architecture for application systems is the “monolith” in which the entire application is designed to run as a single process and is hosted on a resource-intensive computing platform called the “server.” Although the application may be structured as different modules, a change in any module requires the recompilation and redeployment of the entire application. Communication between the modules is carried out by local procedure/function calls.

The next evolution of the application architecture is the “service-oriented architecture” (SOA). In SOA, the entire gamut of solutions (e.g. supporting a business process) is broken up into multiple parts or components called services. This approach makes the development, maintenance and deployment of the entire application easier as operations can be limited to a specific service rather than to an entire application. However, for services to work together to deliver the required solution necessitates the use of heavyweight middleware such as an “Enterprise Service Bus” and communication protocols (e.g. web services). Ensuring security of these middleware components is a complex process. Furthermore, the nature of the connections provided by these middleware components requires that the attributes of individual services such as interfaces to be tightly controlled thus creating a tight coupling between services.

The design of a microservices architecture is intended to address the limitations of SOA by enabling the individual microservices to communicate with each other using lightweight protocols such as Representational State Transfer (REST). Furthermore, the individual microservices can be developed in platforms best suited for them, allowing for heterogeneity in addition to independent scalability and deployment due to loose coupling between individual microservices. However, this approach presents new security challenges such as an increased attack surface due to an increase in the number of components and secure service discovery as a result of the dynamic nature of service instance due to location changes.

NIST SP800-180 defines a microservice as “a basic element that results from the architectural decomposition of an application’s components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology.” Microservices are built around capabilities, as opposed to services, and are typically deployed inside Application Containers [NIST SP800-180].

A microservice architecture consists of a set of 1..N microservices operating together as a “system of systems.” This architecture enables development agility and business flexibility by allowing services to independently evolve in response to business or technical needs. However, this architecture also introduces significant challenges, as the move to microservices for most organizations marks a significant shift from single-node system operations to a complex, multi-node "system of systems" distributed architecture. This, in turn brings both the benefits and limitations of distributed systems development into scope, which can be a surprise for development and operations teams accustomed to working with traditional monolithic system architectures. There are direct implications on microservice security as this approach inherits many of the same security challenges faced by distributed systems. These challenges include managing distributed authentication; ensuring the confidentiality and authenticity of each service message; properly merging and monitoring event logs from multiple nodes in a single service; dynamically managing and rotating service credentials across a distributed enterprise; and patching and vulnerability management in a constantly-changing heterogeneous environment which affects the ability to manage the growth of the enterprise-wide attack surface.

The logical and architectural differences between monolithic and microservices architecture are described in section 1.1. An example application to illustrate the differences in building blocks between the two architectures is also provided. Section 1.2 analyzes the SOA landscape and highlights its characteristics, constraints and deficiencies as well as describes how the MSA extends some of the SOA concepts and addresses those deficiencies.

Section 1.3 describes the benefits of microservices architecture and provides specific use cases where it enjoys advantages of SOA. The security and configuration challenges identified for MSA forms the basis for the issues addressed in the rest of this document.

1.1 Service-Oriented Architecture

The concept of SOA has been around for many years. In Erl's [Erl, 2005] definition of SOA, it is established as an architectural pattern that "can be realized through any suitable technology platform." The basis of Service Orientation is not in technology, but in the theory that the separation of concerns, wherein large problems are separated into smaller, distinct, related parts, form services. The SOA paradigm helps to plan, develop, and manage these services in the IT environment. Therefore, we adopt the following SOA definition, as defined by Erl [Erl, 2005]: "SOA represents an open, agile, extensible, federated, composable architecture comprised of autonomous, Quality of Service (QoS)-capable, vendor diverse, interoperable, discoverable, and potentially reusable services."

Let's clarify below the definitions of "Service," "Service Orientation," and "Architecture."

A service is a self-contained piece of business functionality, with a clear purpose [Stojanovic et al., 2004]. Services are modeled after one specific business process such as billing.

Service Orientation, according to [Erl, 2005; Rosen et al., 2012] is a collection of logic integrated in a software component. This component is provided as a service to the component consumers, where the consumer of the service can also be another service. These independent components, when connected, can provide the full support for a specific business process, such as "managing customer account".

Architecture is the description of the subsystems and the components of a software system and the relationship between them.

1.1.1 LANDSCAPE

The successful deployment of the SOA life cycle (modeling, assembly, deployment and management), is done in the context of what is known as SOA governance. SOA governance is the process of establishing the chain of responsibilities and communications, policies, measurements, and control mechanisms that allow people to carry out their responsibilities.

- SOA governance is a process you implement, not a product you buy[1]. [Oracle 2013]
- SOA governance itself has a set of phases: plan, define, enable, and measure.
- SOA defines 4 basic service types:

Business services are coarse-grained services that define core business operations. They are usually represented through XML, Web Services Definition Language (WSDL) or Business Process Execution Language (BPEL).

Enterprise services implement the functionality defined by business services. They rely on application services and infrastructure services to fulfill business requests.

Application services are fine-grained services that are bound to a specific application context. These services can be invoked directly through a dedicated user interface.

Infrastructure services implement non-functional tasks such as authentication, auditing, security, and logging. They can be invoked from either application services or enterprise services.

Characteristics of SOA- Internal architecture

The 10 Principles of the SOA Design[2]:

1. Interoperability - Services should use standards that allow subscribers to use the service. This allows for easier integration.
2. Loose Coupling - Services minimize dependencies on each other.
3. Knowledge Curtain / Service Abstraction - Services hide the logic they encapsulate from the outside world.
4. Resource management/ Service reusability - Logic is divided into services with the intent of maximizing reuse.
5. Service Discovery - Services should and can be discovered (usually in a service registry).
6. Structural Independence / Service Autonomy - Services should have control over the resources the service consumes / relies upon.
7. Service Composition / Composability - Services "break" big tasks into smaller ones.
8. Granularity / Service Statelessness - Ideally, services should be stateless.
9. Service Quality - Services adhere to a Service-Level Agreement between the service provider and the client.
10. High Cohesion - Services should ideally cater to a single task or group similar tasks as part of the same module.

[1] https://www.oracle.com/assets/implement-soa-governance-1970613.pdf
[2] Erl quoted 7 principles, while others today mention 10 as they are listed above as we believe all of those principles should be mentioned.

Constraints in SOA

Since service orientation means different things to different people, the SOA challenges can be viewed through three domain areas: business, technical/engineering, and operations.
There is dramatic variation in the implementation, attributes, descriptions, and datatypes of services; therefore, it remains problematic to effectively manage the services.
In order to explore an initial classification of challenge areas related to service-oriented architecture systems, the table below presents identified challenges for SOA implementation categorized in each of the three domains[3]: [Lund University, 2015] [Beydoun et al, 2013]

Domain	Challenges
Business	Definition of standards
Business	Requirement understanding
Business	Service Composition
Technical/Engineering	Security and Integrity
Technical/Engineering	Service compatibility
Technical/Engineering	Legacy software migration
Technical/Engineering	Real-time communication
Technical/Engineering	Reliability and Testing
Operations	Making services work together
Operations	Governing SOA complexity
Operations	Monitoring services

Table 1: Domain areas for SOA challenges

[3] In SOA Adoption Challenges, [Beydoun at al.], Challenges fall into two domains: Technical and Organizational-Business.

Furthermore, SOA is generally not deployed in the following scenarios because of the security challenges it presents:

1. **Legacy application security:** Legacy capabilities can be externalized with the help of the SOA-based adapter. While doing so is feasible, the designer should factor in the limitations of the capability's existing security model. The proposed SOA adapter might not have insight into the model's proprietary nature.
2. **Loose coupling of services and applications:** SOA security should not violate the overall software design principles, such as the solution component's loose coupling. The service is intended to provide a sustainable interface to a capability, shielding the service consumer from the design and implementation details.
3. **Services that operate across organizational boundaries:** Traditional perimeter security (enterprise security boundary) might be insufficient to mitigate risks presented by transorganizational interactions. A set of compensating controls might be necessary to maintain compliance with enterprise security policies.
4. **Dynamic trust relationships:** SOA participants have to establish mutual trust, possibly including the parties responsible for the hosting and maintenance of the SOA registries. Since these relations are of a dynamic nature, trust is also likely to have a dynamic nature.
5. **Composite services:** Service composition and aggregation are two forms of service association. Service composition, however, might violate the SOA independence principle. Hence, service orchestration and choreography are the preferred service association strategies.
6. **Need to be compliant with a growing list of standards:** SOA standardization is a concern due to the growing number of security standards and regulations. However, this need for compliance is no different from any other capability integration realms.
7. **SOA flexibility:** SOA solutions are intended to be flexible and customizable. It can improve time-to-market in IT supported processes and business solutions. Portfolio gap analysis, transition planning, and architectural governance, is able to provide enterprise architecture with opportunities for change and merge strategic business and IT objectives. By leveraging service-oriented portfolio gap analysis, the enterprise planning cycle strategy can be transformed into a roadmap of specific change initiatives, and provides the opportunity to govern the execution of that resulting roadmap. The SOA lifecycle then drives solution delivery in the context of one or more specific projects in the roadmap. Therefore, the SOA solutions should be customized and extended as appropriate in order to make the business processes relevant, personalized and responsive.[Simplicable, 2011] [The SOA Source Book]

SOA deficiencies:

While SOA is considered a modern abstraction approach, it is still assuming a centrally managed abstraction. Some of the potential deficiencies are aligned to the constraints and to what microservices deliver:

8. Using service bus to achieve SOA reliability, scalability and communications disparity promises comes with certain limitations (e.g. tight coupling, dependent processes) uncharacteristic to microservices.
9. SOA is perceived as having a performance impact; however, this is partly attributed to the protocol rather than to the properly designed service itself. Poorly orchestrated / choreographed services, however, might violate SLA / QSA.

10. Well-established, immutable service interfaces are essential for the service's usability. It is, therefore, expected that the service lifecycle is controlled by the proper governance process.
11. Testing of SOA is likely to be as complex as any other distributed solution, where the consumer objectives, subject to SLA/QSA, are heavily dependent upon service components' ability to uphold the OLA.

In a way, SOA can be seen as a stepping stone that helps implementation of other common architectural patterns (i.e. strangler facades[4], anti-corruption layer[5], etc.), between monolithic to microservices architecture (MSA), when an architect is designing DevSecOps processes and the associated level of freedom. In this way, service boundaries are already clearly defined and services can independently evolve into a microservices architecture.

1.1.2 HOW MICROSERVICES EXTEND SOA

Microservices are an emerging architectural approach that extends SOA practices and overcomes traditional SOA deficiencies. [Haddad, CIOReview]

Not from a computer science perspective, but from an implementation developer perspective, the shift to a Microservices Architecture (MSA) can be extremely radical, although nothing radically new has been introduced in the microservices architecture. Microservices architecture is the logical evolution of SOA and supports modern business use cases.

Similar to SOA, a microservices approach decomposes individual applications into multiple, discrete services. Units of functionality, implemented by the atomic microservices using the single responsibility principle (SRP)[6], are included to form useful business process activities, to create integration flow, and to provide user experience. MSA enriches the collection of SOA patterns with bounded context, bulkhead, circuit breaker patterns.

The additional principles and patterns help teams deliver autonomous, loosely coupled, resilient, and low cohesive solutions. The term that describes this is called Cloud-Native software development[7] (see Section 2).

A microservices approach constrains SOA with additional principles and practices as mentioned above, including: context mapping, loosely coupled/high cohesion, shared nothing architecture, dynamic deployment, and parallel development.

[4] Fowler M., (2004): StranglerFigApplication. https://martinfowler.com/bliki/StranglerFigApplication.html
[5] Kalske (2017), University of Helsinki: Transforming monolithic architecture towards microservice ar-chitecture. https://core.ac.uk/download/pdf/157587910.pdf
[6] Robert C. Martin (2005): https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view
[7] These solutions are engineered to benefit from a cloud-native architecture. One of the four underlying pillars within each cloud application is: Microservices. (Along with Containers, Dynamic Orchestration, and Continuous Delivery).

Below are key differences between SOA and Microservices that show how Microservices extend the SOA functionalities [BMC Software, 2017] [Cerny et al., 2017]:

SOA	**Microservices**
"share-as-much-as-possible" architecture approach	"share-as-little-as-possible" architecture approach
Weights on business functionality reuse	Weights on the concept of "bounded context"
Provides common governance and standards	Affords a more relaxed governance, with a focus on collaboration and freedom of choice
Use of Enterprise Service Bus (ESB) for communication	Uses less elaborate and simple messaging system
Multiple message protocols	Lightweight protocols such as HTTP/REST & AMQP or JMS
Multi-threaded with more overheads to handle I/O	Single-threaded (though not a prerequisite[8]) Event Loop features for non-locking I/O handling
Use of containers is less popular	Containers are convenient for packaging microservices
Required to modify the monolith for some system changes	Creates a new service for some system changes
Maximizes application service reusability	Loosely coupled self-contained services that have their own database in order to be decoupled from other services[9]
Not mainstream support for DevOps / Continuous delivery	Strong focus on DevOps / Continuous Delivery

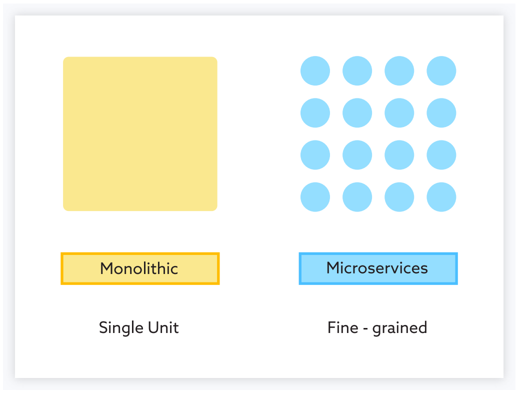
Table 2: SOA vs Microservices

[8] https://www.infoq.com/articles/engstrand-microservice-threading/
[9] https://medium.com/@raycad.seedotech/monolithic-vs-microservice-architecture-e74bd951fc14

1.2 Comparison between Monolithic and Microservices Architecture

Architecturally, a monolithic application is designed as a single computational node. Hence, 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 each providing a service.

Monolithic vs. Microservices



1.2.1 LOGICAL DIFFERENCES

Conceptually a monolithic architecture involves generating an application that should be deployed as a whole, while a microservices-based application contains multiple self-contained, loosely-coupled executables called services or microservices. The individual services can be deployed independently. Any change that is needed to a certain functionality of the overall application will involve at least re-linking and in some instances re-testing of the whole application before being deployed again (as any change requires several testing levels, depending on the application's criticality). However, in the case of microservices, only the relevant service (or services) is (are) modified and redeployed since the independent nature of the services ensures that a change in one service does not logically affect the functionality of any other service. In monolithic applications, scaling in response to increased workload (due to increase in number of users or increase in frequency of application usage) is usually limited to running new instances of the entire application to spread the load more evenly.

Microservices, however, enables a different approach to scaling. The increase in resources can be applied selectively to those services 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. By modular construction, what is meant is that the application may be built from a large number of components and libraries that may have been supplied by different vendors and some components (such as database) may also be distributed across the network.

In such monolithic applications, the design and specification of APIs (Application programming interface) may be similar to that in microservices architecture. However, the difference between such modularly-designed monolithic applications (what is sometimes called a classic modular design) and microservices-based applications is that in the latter, the individual API is network-exposed and hence independently callable and re-usable.

Another important difference is that using modules in a monolith is a weaker form of isolation since the modules all run in the same process (generally) and developers can more easily create cross-module dependencies, thus weakening cohesion. Comparatively, microservices establish a strong isolation through a network boundary which is also usually reinforced by microservices being maintained by separate development teams, making cross-microservice dependencies more difficult, resulting in cleaner separation of responsibilities. This clean separation in turn is what makes it possible to evolve microservices independently, since monolith module changes still require re-generation/re-testing/re-deployment of the entire monolith's artifacts. [Newman, 2015]

The differences between monolithic and microservices-based application discussed above is summarized in **Table 3** below:

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 is likely to warrant re-deployment of entire application	Only the modified services need to be re-deployed
Scalability involves the Load Balancer increasing or decreasing resources as required by the application that allows balancing at multiple points, not necessarily the ENTIRE app	Each of the individual services can be selectively scaled up by allocating more resources
Component invocations are local	Network-exposed service invocations enable broader distribution and independence[10]
Weaker form of isolation since the modules all run in the same process	Strong isolation through a network boundary

Table 3: Logical Differences between Monolithic and Microservices-based Application

[10] It should be clarified here, that this is simultaneously a benefit AND a drawback, since moving to a microservices architecture is essentially moving to a distributed networked system-of-systems architecture with significant increases in complexity that come with the increases in flexibility. A lot of the MSA best practices are efforts to manage the complexity or accept and work within its limitations.

1.2.1.1 An Example Application to illustrate the logical differences

To illustrate the logical differences discussed above, lets explore an example of a small web-shop or online retail application. The main functions of this application include the following:

- A module that displays the catalog of products offered by the web shop with a picture of the product, product number, product name and the unit price;
- A module for processing customer orders gathering information about the customer (name, address, etc.) and the details of the order (name of the product from the catalog, quantity, unit price, total price, etc.) 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 (the total package to be shipped with the different items in the order and the quantity of each item, the type of shipping (one-day, two-day, etc.), the shipping address, etc.;
- 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 the above web-shop application when it is designed as a monolith and microservices-based are explored in table 4 below:

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.	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 service can put the details of the order to be shipped in a message queue to be picked up asynchronously by the shipping application which has subscribed to the event.
Handling Changes/ Enhancements	The invoicing module needs to have an enhanced functionality to enable customer payments through debit cards – The entire application needs to be recompiled and redeployed after making the necessary changes.	The invoicing function is designed as a separate microservice and hence it is necessary only to recompile that service and to redeploy again.

Scaling the application – allocation of increased resources	The order processing functionality involves longer transaction times compared to shipping or invoicing functions – vertical scaling that involves using servers with more memory/CPUs should be deployed for the entire application.	It is enough to allocate increased resources for hardware (considering the hardware limits) where the order processing microservice is deployed. Also, the number of instances of order processing microservice can be increased for better load balancing.
Development and Deployment Strategy	The development is handled by the development team (which after necessary testing) hands over the task of deployment to an infrastructure team that handles the allocation of suitable resources for deployment.	The complete lifecycle from development to deployment is handled by a single DevSecOps team for each microservice – since a microservice is a relatively small module, with a single functionality and built in a platform (OS, Languages Libraries and IDEs) that is optimal for that functionality.

Table 4: Monolith vs Microservices architecture design for application

1.2.2 ARCHITECTURAL DIFFERENCES

This document takes the view that the difference between SOA and microservices does not concern the architectural style as asserted in [Zimmerman 2016]. Hence the basic architectural principles for MSA and SOA are identical and include the following: decomposition of application into technology-agnostic self-contained modules called Services which embody technical concepts such as Interoperability (services call each other directly without the intervention of a link library), High Cohesion (services cater to only a single task or several tasks which are similar in nature) and Loose Coupling (lack of dependency – a situation where a change in one service does not require a change in any other service). Thus, the architectural concepts discussed for microservices below are directly inherited from SOA.

Since each of the modules operates without coordination with others, the overall system state is unknown to individual nodes. Further, in the absence of any global information or global variable values to go by, the individual nodes make decisions based on locally available information [Tanenbaum at et., 2007].

The differences in architecture between monolithic and microservices application and its implications are summarized in **Table 5** below:

Monolithic Application	Microservices-based Application
Designed as a single computational node. Hence 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.

Table 5: Architectural Differences between Monolithic and Microservices-based Application

An Example Application under Monolithic and Microservices Architectures[11]:

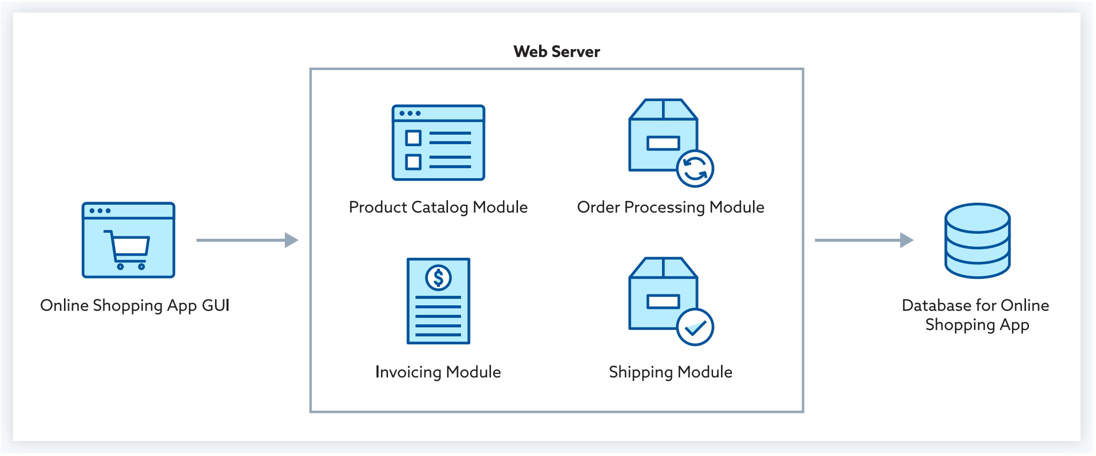


Figure 2 : Online Shopping Application – Monolithic Architecture

[11] When presenting best practices in relation to security as this document does, microservice architecture best practices should have a mention also. One such best practice is that of each microservice managing its own DB independently of all others. Hence, allowing 2..N microservices to share one DB introduces inter-service coupling at the DB layer. [Newman, 2015]

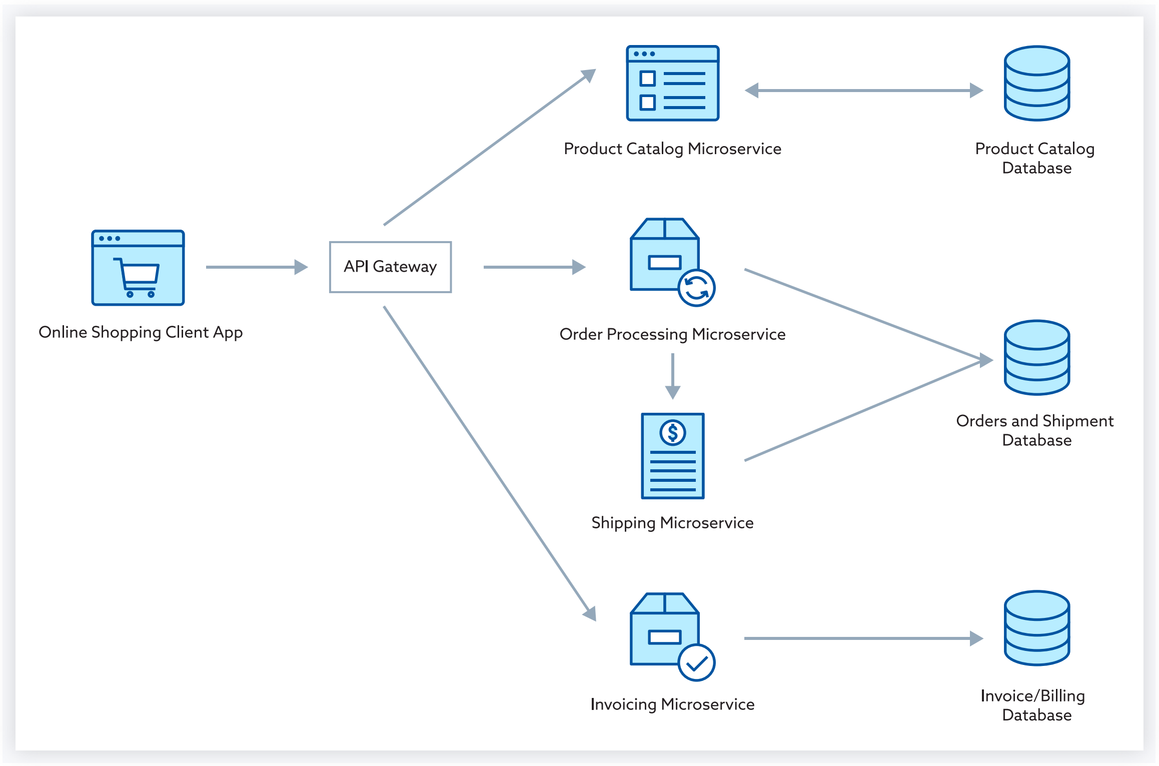

Product Catalog Microservice
Product Catalog Database
Online Shopping Client App
API Gateway
Order Processing Microservice
Orders and Shipment Database
Shipping Microservice
Invoicing Microservice
Invoice/Billing Database

Figure 3 : Online Shopping Application – Microservices Architecture [Yarygina et al., 2018]

1.3 Benefits and Challenges of Microservices

The most important drawback as well as challenge is complexity in the operational environment as a result of many microservices deployed in production, including data consistency (eventual consistent), monitoring, troubleshooting, hardening, securing the container(s), etc.

1.3.1 USE CASES FOR AND AGAINST MSA VS SOA

The following table [SOA Source Book] shows the use cases for and against microservices architecture (MSA) vs service-oriented architecture (SOA). It sets the context for the next two sections describing benefits and challenges of microservices.

Use Cases	MSA Candidate	SOA Candidate
Business Drivers		
Solution resilience (availability, recoverability et al.) is a critical success factor.	MSA can offer an inexpensive means of providing resilience because a microservice that fails can be taken over by parallel instantiations (of the same microservice) that are not dependent on other services.	SOA resilience may be more expensive to provide as SOA applications may have dependencies and interactions required by services choreography.
Solution domain is well established.	Cost of re-architecting may not be worthwhile.	SOA supports existing infrastructure.
Solution is exploratory vs established.	MSA enables rapid development and deployment cycles. It permits rapid failover and self-recovery. These characteristics result in high availability and seamless user interaction.	In comparison, SOA is more suited to a business domain with well-known requirements.
Technical Drivers		
Solution should be broken up into a collection of individual and independent service components.	MSA design revolves around the concept of independent services.	If services cannot be independent, SOA is a better choice.
Solution requires resilience and elastic scalability.	MSA provides resilience (quick recovery) and scalability through parallelism.	SOA provides an approach to achieve these ends using vertically scaled machines which may be more cost-effective.

Solution requires orchestration across services.	MSA relies on smart endpoints but dumb pipes[12] philosophy for low-level choreography.	SOA uses protocols, such as Business Process Execution Language (BPEL) or Web Service Choreography (WS-Choreography), for orchestration.
Operational Drivers		
High rate of churn or update of individual services expected.	If the API definition, service structure, and operational requirements are not stable, MSA may be a better candidate.	High rates of service update will likely be more complicated to support in a traditional SOA infrastructure.
Application development/deployment cycle is short.	MSA offers low overhead and rapid parallel development.	A more complex full SOA may take longer to develop and deploy.
New application (or business area) vs building on top of a legacy application	New application or business area	Building on top of legacy application

Table 6: Use Cases of MSA vs SOA

1.3.2 BENEFITS

As alluded to in Section 1.1.2., the characteristic differences between a Microservices Architecture and SOA bring about advantages beyond those provided by SOA as follows:

- Business benefits
  - Alignment with the business
    - A microservice is based on a single atomic business function realizing atomic business capabilities, owned by a single business unit. Thus, it is able to be updated in concert with changes to the business unit.
  - Agile and Responsive to business needs
    - An entire application does not have to be taken down just to update or scale a component. Each service can be updated or scaled independently. This gives the business the ability to respond faster.

[12] https://martinfowler.com/articles/microservices.html

- Enablement of teams
  - A number of distributed and autonomous teams using the Continuous Integration (CI) and Continuous Deployment (CD) paradigm is able to work on independent services with near-constant integration of software, with automated deployment of the integrated code.
- High Availability / Disaster Recovery
  - The resilience provided through the parallelism of microservices, enabled by their independence, permits rapid failover and self-recovery. These characteristics result in high availability and seamless user interaction.
- Intrinsically interoperable services
  - Services are typically implemented using mandatory well-defined, published interfaces, according to defined service boundaries. Thus, the time and cost associated with composing applications are reduced, enabling faster response to changes in the marketplace or entry into a new market.

- Technical benefits
  - Polyglot approach allows pick and choose best-of-breed technology, tools and platforms.
    - There is no language or technology lock-in. As each service works independently, developers can choose any language or technology to develop it, ensuring its API endpoints return the expected output.
  - Each service in a microservice can be deployed independently.
  - Once the code of a service is written, it can be used in other projects where the same functionality is needed.
  - The microservice architecture enables continuous delivery and continuous integration.
- Operational benefits
  - If one service fails, then its failure does not have a global cascading effect. This helps in debugging as well (without intending to imply, of course, that a Microservices architecture is resilient to failure. In addition, debugging would also become more difficult due to increased complexity. The cascading effect is presented as a drawback below in section 1.3.3). The circuit breaking patterns help in the microservices environment by ensuring that no additional calls are made to the failing service once the circuit breaker is open.
  - Components can be deployed across multiple servers or even multiple data centers.
  - Microservices work very well with container orchestration tools like Kubernetes, DC/OS and Docker Swarm.

1.3.3 DRAWBACKS AND CHALLENGES

Managing a multitude of distributed services at scale is difficult for the following reasons:

- Business challenges
  - Project teams need to easily discover services as potential reuse candidates. These services should provide documentation, test consoles, etc. so re-use is significantly easier than building from scratch
- Technical challenges
  - Interdependencies between services need to be closely monitored. Downtime of services, service outages, service upgrades, etc. can all have cascading downstream

effects and such impacts should be proactively analyzed; otherwise the failure resilience that can be a benefit above becomes a drawback as fault tolerance in microservices is more complex than in a monolithic architecture.

  - Choosing the right service size
    - While breaking the monolith application or creating microservices from scratch, it is very important to choose the right functionality for a service. For example, if the business creates a microservice for each function of a monolith, then developers would end up with lots of small services, which would bring unnecessary complexity.
  - Testing
    - With lots of services and their associated interdependencies, sometimes it becomes challenging to perform end-to-end testing of a microservice.
  - Inter-service communication
    - Inter-service communication can be very costly if not implemented correctly. There are options such as message passing, RPC, etc., and as such, one should be chosen that fits the requirements with the least overhead.
  - Managing databases
    - Databases may be implemented local to a microservice. But, to close a business loop, changes may be required in other related databases which can create challenges such as partitioned databases.
  - Service discovery and documentation
  - Fault tolerance
  - Quality of service
  - Security
  - Request traceability
  - Failure triage
- Operational / Security challenges
  - The Software Development Lifecycle (SDLC) of services should be highly automated. This requires technology, such as deployment automation technologies and CI - CD frameworks, but more importantly, discipline from developer and operations teams. Without responsibility sitting only on the shoulders of Developers and Operators, it is equally important that management is also involved, either to mandate top-down, to support the team, or more importantly, to champion the benefits of security across the board.
- Deployment
  - To deploy a microservice, a distributed environment should be used such as Kubernetes. It is possible to deploy one microservice per Virtual Machines (VM), but this approach would result in significant, unnecessary technical overhead. Kubernetes, by contrast, introduces significant complexity if not used as part of a larger orchestration platform which can bundle a lot of value-added services like monitoring, security, etc.
- Monitoring
  - Monitoring individual services in a microservices environment can be challenging. This challenge is being addressed and tools are being developed to monitor and debug microservices.

Even with the above challenges and drawbacks, deploying microservices makes sense when applications are complex and continuously evolving.

2.0 Microservices Security Architecture for Cloud-Native Applications

Developers are interested in developing new applications that are scalable, portable, resilient and easily updated. The following chapter describes Microservices architectural considerations for cloud-native applications.

2.1 Overall Threat Model and Associated Best Practices

2.1.1 THREATS AND SECURITY CONCERNS

As more organizations shift away from the monolith towards distributed, microservice-based architectures, security concerns are increasing. In a microservices architecture, the attack surface increases significantly and security concerns are exacerbated due to the various network connections and APIs used to establish communication channels between all those components, creating additional methods for attack, interception of in-transit data and manipulation of the data at rest. Microservices-based architectures also expose a lot more of the system's functionality directly to the network which in turn increases the attack surface. The fact that multiple small containers may spread across many different systems/hosts, etc., and that they may have to function cohesively means the threat landscape is significantly increased. This also means each container has to be properly maintained, managed, and secured, all of which is extremely time consuming without proper tools. Furthermore, all controls are implemented by software and services developers, which can lead to inconsistencies and gaps in security controls. In order to achieve consistent enforcement of security controls and enforce development processes along with containers, special tools and techniques are needed. Another issue is that the standardized replicable nature of containers also means that a vulnerability in one microservice can be quickly replicated many times over as the source code is reused.

With a traditional app/service monolith, the app/service components are typically hosted on one or more servers in a network, which makes it easier to focus on the exposed ports and APIs, to identify an IP address, and to configure a perimeter around it. With microservices, this gets much more complex due to the many exposed ports, API gateways and wider attack surface as the economies of APIs increases, and authentication is also very distributed. This essentially means that running a distributed set of services will require enforcement of security controls in a distributed manner and different sets of stakeholders will have to play their part and be onboard to implement a successful security ecosystem for containers and microservices.

There are some clear security benefits for containers and microservices from a security standpoint, especially given that the applications' components and services are isolated. Microservice security can be implemented at a much more granular level, with controls applied to specific services, APIs, and network communication pathways.

Microservices do provide the ability to implement a defense in depth strategy, but the way security controls are implemented is a huge shift from traditional methods. Within a microservices architecture, there are multiple transactions and interactions. Thus, the security of the app/service components depends on container interactions and knowledge of what they are and when they happen. Visibility into inter service communication becomes challenging or at the very least the scale of the problem increases.

There are several deployment patterns for microservices:

1. Microservices in Virtual Machines (VMs)
2. Microservices in containers
3. Microservices deployed on serverless platforms

2.1.1.1 Threat Model: Microservices in Virtual Machines (VMs)

Threat models for microservices running in virtual machines are well known and some of them are identified below.

- Breach of Process Isolation - VM Escape
  - Major threats to any hypervisor come from rogue VMs. Rogue VMs manage to subvert the isolation function provided by the VMM/hypervisor to hardware resources such as memory pages and storage devices. This can happen for several reasons, such as hypervisor design vulnerabilities, malicious or vulnerable device drivers, etc. Impacts of a rogue VM taking control of the hypervisor include the installation of rootkits or attacks on other VMs on the same virtualized host.
- Breach of Network Isolation
  - Threats to isolation in a virtualized environment include attacks such as IP or MAC address spoofing by a rogue VM and traffic snooping, or the interception of virtual network traffic, intended for a VM on the same virtual network segment.
- Denial of Service
  - Misconfigured or malicious VMs could be consuming a disproportionately high percentage of host resources, resulting in denial-of-service to other VMs on the hypervisor host.

There are many other threats in the VM environment. More details can be found in NIST 800-125 Rev 1.

2.1.1.2 Threat Model / Best Practices: Microservices in Containers

Threat models and associated best practices for microservices running in containers are presented in the following table:

#	Identifier	Weakness	Threat Impact	Mitigations to Consider
1	**Host Physical Security**	Underlying hardware, chipset and BIOS Vulnerabilities	The host OS running the container can be compromised.	Use a trusted platform with secure boot and updated BIOS with patches for vulnerabilities identified. HOST OS trust can be also achieved using a vTPM. This approach also ensures that containers cannot modify the host kernel by loading new modules or exploiting kernel vulnerabilities. Containers can also attest their own state by using the hash extend features that vTPMs provide. vTPMs should be added to each container to ensure the root of trust is maintained throughout the stack. Integrity Measurement Architecture (IMA) could also be considered, since the measurement list cannot be compromised by any attack without being detectable. Other mitigations are to re-instantiate containers periodically at regular intervals of time. This is a common method used in the industry.
2	**Host OS Protection**	OS vulnerabilities can expose containers to a multitude of threats, since multiple containers share	One container can impact the performance or cause a denial of service on other containers.	Use container specific OS rather than Generic OS distributions, keeping OS versions up to date and patched. Disable all unused services, interfaces and minimize Admin account groups. Also use auto patching, auto deployment and no manual changes to be allowed

		the same Operating system Kernel.		in the runtime. In the config files, an attribute should identify the type of OS and its distribution and other features for easy identification of the version, etc.
3	**HOST OS Protection-Container Escape**	Containers run in a shared OS environment and a host Kernel.	Not having application isolation when running in a shared environment can expose data or impact the integrity of app components as a malicious app could interface with other app containers.	Use Kernel Loadable Modules like SELINUX. SELINUX can help isolate containers from the host and each other by applying mandatory access control for users, processes and apps. AppArmor is a Mandatory Access Control (MAC) system which is a kernel (LSM) enhancement to confine programs to a limited set of resources and can be used to enhance the security of the underlying OS. AppArmor's security model is to bind access control attributes to programs rather than to users. AppArmor confinement is provided via profiles loaded into the kernel, typically on boot. Cgroups control the resource allocation and usage. Secomp profiles associated with Containers can restrict the system calls it can make. Linux capabilities can be used to lock down Root in a container. Lightweight OS's can be used with stripped down capabilities which may be more suitable for a container environment. It is essential to have an attribute that identified that SELINUX is turned on and off.

				To enhance security further, it is also recommended to implement kernel patches that address bound checks on userland/kernel copy as well as take advantage of UDEREF, and forward-edge CPI.
4	**HOST OS Protection-Runtime Configurations**	Containers run in a shared OS environment and a host kernel in runtime environment. Thus cross container vulnerabilities or misconfigurations can impact the running containers on the shared OS.	One container's change in config could potentially impact other containers and cause compromise.	Containers should be hardened and use tools such as Seccomp, AppArmor as appropriate. Runtime container Configs/ Parameters are controlled by different API's provided by container Runtime.
5	**Host and Kernel Security**	If an attacker compromises a host system, container isolation and other security controls are invalidated.	Complete compromise of the services /app components running on compromised host.	Host and container engine configs should be hardened (use tools such as Docker Bench Dockscan, Falco, CoreOS Clair, Anchore Engine) with authenticated restricted access. Base systems should be updated and any third-party repos updated as well. Use of minimal container centric host systems will reduce the attack surface. Map trusted app containers on separate machines. Do not run untrusted apps with root privileges.

6	**Container Breakout**	Container Breakout exploits are available in the wild and get manifested in the environment.	Complete compromise of the host.	Create isolated namespaces and limit maximum privileges. To run privileged containers, Root of trust validation from a trusted repo should be validated. Ensure Kernel security and hardening and other OS security hardening techniques to ensure protection, isolation and detection of any container breakouts. Ref: http://gzs715.github.io/pubs/SECNAMESPACE_SEC18.pdf
7	**Container Resource Abuse**	App components could potentially use IPC resources to communicate instead of REST API, and individual services could manipulate the HOST IPC objects and impact other app containers running on the host or potentially cause a denial of service or back queues with lots of messages.	Impact Resources being used by other containers on the host.	Do not allow services to use IPC or shared memory segments. Do not allow containers to see options that will enable sharing namespaces associated with HOST.
8	**Container Resource Abuse**	Host sensitive directories and file systems if mounted	This can jeopardize host security.	Do not allow containers to mount any host sensitive directories in read or write mode as container volumes.

		in read/write mode by containers can potentially give containers the ability to modify files in those containers.		
9	**Container Resource Abuse**	Default options of publishing container ports to the host can increase attack surface.	Widely allow communication to host from containers and cause denial of service (DOS).	Specify the precise interface the port should bind to explicitly. Any traffic to and from container should be restricted to that interface.
10	**Container Resource Abuse**	Design miscalculations or deliberate attacks can cause a denial of service (DOS) by resource abuse. The list of resources to protect are CPU, I/O, Swap, Kernel resources, etc.	This could lead to DOS for the certain services.	Resource limitations should be set bundled with Linux Kernel (use Cgroups). Load testing of services should be done in test before promotion to prod to get a good sense of scalability and performance. Threshold should be set to monitor and alert on the same before the resources are constrained.
11	**Base Image Vulnerabilities**	A container could be performing well but could have an older version of Java or other libraries running in it.	The older versions could lead to compromises.	Having the minimal base image is essential to reduce attack surface. Use software from distributors who guarantee updates, preferably progressive rolling updates. Data should be separated from images.

				Minimal software in images makes it simpler to manage patching of vulnerabilities. Use Runtime scanning tools (e.g. Aquasec, Twistlock, Stack-Rox, etc.) and keep these tools updated with the latest vulner-ability information (e.g. Quay, Clair, etc.)
Container Content Related Threats				
12	**Container Contents from Open Source or Third-party Providers (e.g. Apache web server, Node.Js)**	Malicious components could be part of third- party containers.	This could provide backdoors for the attackers or allow for complete con-trol of apps running on a Host.	It is important to identify what packages are enabled in containers and who built the containers. To that end, enabled packages should be hardened up-front in the CI/CD pipeline to ensure that new patches do not revert the packages to their default configuration. Operators should be prepared to manage their software supply chain to include regularly updated packages in CI/CD pipelines. Ideally, organizations will procure packages and build container images themselves. These containers should be placed in a staging area where they can be scanned using tools, then packaged in container images and promoted to internal repositories. There can be API base integration with the tools to automatically scan when a new file is downloaded into staging area.

				Developers must not be allowed to use unverified images; use images from approved Docker internal registries only. Digital signature validation of the images is critical but is not a replacement for performing current vulnerability assessments.
Container Registries and Access to Images				
13	**Container Registries and Access**	Internally built software and container builds and their security can be compromised if proper access controls are not placed on them.	This could lead to insider threats and mistakenly promoting code to repos which could get deployed to production and impact service availability.	Implement private registries with separate branches for Dev, test and production code with appropriate segregation of duties by roles. Automate scanning of builds before code can be promoted to the prod branch. Ensure that as new updates are applied to the base image, in the next cycle of builds the updates are applied to the running containers in prod.
Security of the Build Process				
14	**App Code Integration with Runtime Libraries in a Container**	Any changes made to app code can impact the integrity of the software stack.	Multiple app code configs and the dependencies on infrastructure can lead to inconsis-	Build once and deploy immutable containers. When code is updated, build a new image and deploy to runtime through the CI/CD pipeline. Automated builds and deployments guarantee configuration uniformity and reduce risk to a great extent. Configuration should be applied to packages in the CI/

		Developers shouldn't be allowed to make changes to the runtime environment.	tent security controls in an environment.	CD pipeline to ensure new deployments do not drift or are deployed with default controls.
Deployment				
15	**Deployment to the Cluster**	Stale images could be deployed if the policies for deployment are not controlled and deployments are not based on policies.	Availability could be impacted as well as integrations of associated services.	Strong security policies should be defined around the context in which an app/service can be deployed and constraints applied to the same (e.g. CIS Benchmarks, POD Security Policies in Kubernetes allows admins to control the running of privileged containers, the capabilities containers can request in runtime, the use of host directories and volumes etc.)
Security of Container Platform				
16	**Container Deployment to Hosts, Host Capacity, which Containers Interact with Each Other, How Do They Discover and Identify Each Other**	Container management environment can be manipulated and impact resources for applications.	Availability could be impacted as well as the security of the whole runtime platform.	APIs are used for automation deployment and container management. Hence, API security is key, as is identity and access management. Use X509 or tokens. Application isolation limits damage from compromised tokens. All communications should be conducted using TLS 1.2 / TLS 1.3 Master Daemons (e.g. etcd) should not be exposed directly to services.

				Runtime container health and security detection is a key security control. Use Linux namespaces and consider rameworks such as IMA.
Network Isolation				
17	**Co-tenant Threats**	Containers for the same app can span multiple clusters, and in this multi- tenant environment one app can impact others for security, e.g. lateral movement using compromised tokens.	This could result in the compromise of multiple apps/services and also the potential for data leakage.	Use network namespaces for isolating collections of containers (PODS), which provides unique IP and port ranges. In order to receive traffic from other PODS, explicit network policy will have to be established as by default PODS can't send traffic from one network namespace to another. SDN overlay will provide a finer grain control on inter container network policies as well as isolation between master and slave nodes. Use of platform controls for egress should be established, e.g. access to databases. Network scanners should be deployed for network policy violation detection.

Table 7: Threat Model for Microservices in Containerized Environments

2.1.1.3 Threat Model / Best Practices for Microservices Running in Serverless Environments

Threats of running microservices in a Container PaaS remain essentially the same as containers. With that said, developers of the microservices don't have to worry about the underlying threats in the Container PaaS environment because they are obfuscated from developers and managed by the service provider. Essentially the developers can't manipulate the underlying container environment, but if the provider has not secured the platform, then malicious entities can take advantage of the same and exploit the services.

Many cloud providers offer serverless compute (e.g. AWS Lambda). Following are some best practices for microservices running in serverless environments.

- Define and Document the Map of the Application
  - With serverless apps being comprised of hundreds of functions, it is important to have a complete picture in order to understand potential risk. Code can reveal business processes which could be sensitive. A map should consider:
    - What data is processed and/or stored in the app?
    - What is the value of the data?
    - What are the services that access the data?
- Perimeter Security at the Function Level
  - The fragmentation of the application into smaller components that are callable, coupled with the use of triggers from diverse sources (such as storage, message queues, and databases) means attackers have more targets and more attack vectors. Using mitigations such as WAF and API Gateway are good, but another serverless security best practice is to apply perimeter security at the function level. Implementing function-level perimeter security generally requires disciplined secure development practices. OWASP Security Champions can be useful for threat modeling and applying mitigations at the function level to mitigate OWASP Top 10, which in turn reinforces the need for well-defined DevSecOps practices.
- Implementing Suitable, Minimal Roles for Each Function
  - Serverless can substantially increase the number of resources that can act and be acted upon. Developers should consider the policies governing the interaction between hundreds of resources, with hundreds of possible permissions in each direction.

    It is important to invest in crafting suitable, minimal roles for each of the functions. Additionally, it is important to ensure that each of the functions executes with the smallest viable set of privileges, so the damage from any holes that slip through is minimal. Roles and functions should be reviewed as often as possible. In serverless, things that were once well-configured can suddenly be sub-optimal, as others might have changed a role, policy, or function that makes some other part of the application vulnerable. Developers should consider emerging technologies that can help craft these policies and issue an alert any time things change.
- Secure Application Dependencies
  - Functions often include dependencies, pulled in from npm (Node.js), PyPI (Python), Maven (Java) or other relevant repositories. Application dependencies are prevalent and frequently vulnerable, with new vulnerabilities disclosed regularly. The nature of serverless makes managing third-party dependencies manually particularly challenging. Furthermore, interdependent code bases can compound this issue (e.g. dependency A in codebase imports dependency B that imports dependency C, which contains a vulnerability). Securing application dependencies requires access to a good database

and automated tools to continuously prevent new vulnerable packages from being used and getting alerted for newly disclosed issues. Additionally, minimizing the impact of vulnerable libraries by ensuring proper segmentation of the app into disparate services, and scrupulously applying the principle of least privilege is critically important because these are wrapping controls to compensate for the likely injection of vulnerable libraries. Segmentation and least privilege help contain the blast radius.

- Important to Detect and Act against Bad Code
    - Serverless deployments, with their diverse triggers and infinite scaling, can mean that the smallest of code errors can quickly turn into a self-inflicted denial-of-service attack from within an application. With a more exposed attack surface, bugs can more easily turn into security liabilities. It is critical that developers are regularly and adequately trained. Code reviews will help as well, but more so will monitoring the code and configuration using tools to test configuration.
- Incorporate Tests for Service Configuration to CI/CD & PROD
    - Serverless deployments invite increased product velocity, increasing the speed at which enterprise applications create value, but they also provide many opportunities to break security. Continuous testing and validation of what configurations were designed and what is actually running in production is required.
- Monitoring the Flow of Information
    - The impact of data flow is unique to serverless due to the significantly larger number of components in applications. It is recommended to monitor the flow of information and data to ensure it is going only between the services for which they are intended, and the right information ends up in the right place.
- Mitigate for Denial-of-Service or Denial-of-Wallet
    - While serverless apps scale easily, scaling is still not infinite. Apps are susceptible to Denial-of-Service attacks if an attacker can saturate resource limits. If the concurrency limits are high enough to avoid this issue, there can be Denial-of-Wallet attacks, in which an attacker overwhelms the app to rack up costs. Enabling massive auto-scaling of the service when using serverless can help mitigate some DoS attacks but opens up DoW attacks. Deploying proper mitigations for DoS attacks, such as API Gateway, in front of web endpoints helps but one must consider DoS and DoW via other triggers, such as Kinesis and S3. Function self-protection can help detect these attacks, minimize their impacts, and dynamically adjust scaling choices to help with mitigation.
- Make FaaS (Function as a Service) Containers Refresh
    - Developers should consider strategies in the lifetime of function instances (e.g. on some platforms, there are APIs to make an instance refresh). It is advisable to consider employing a security solution that can detect things that try to hang around in function instances, like extra code, hidden native process, etc., and then flush those things out of the container. Said security solutions may incur additional complexity or agents running as services.

Like any facet of cybersecurity, securing serverless applications requires a variety of tactics throughout the entire application development lifecycle and supply chain. Stringent adherence to best practices will help improve the overall security posture. However, proper development is not enough. To achieve ideal protection, it is important to leverage tools that support and provide continuous security assurance, attack prevention and detection, and deception.

The tooling for deploying a single FaaS function is available today; however, the tooling required to coordinate, orchestrate or deploy multiple functions at the same time is lacking. One example is a case in which there are multiple functions that comprise a serverless application requiring deployment in a particular flow or all at once. There are not many tools that can help manage this type of use case. The tooling to ensure zero downtime for serverless applications is not robust either. Integration testing of serverless applications is challenging. As with the FaaS environment, developers depend on external resources to maintain state, and sometimes the testing has to be done in an actual production environment.

2.1.3 AUTHENTICATION

In a monolithic environment, authentication and authorization are handled within the application using a module that validates the incoming requests for required authentication and authorization information, as shown in the following diagram; this module also allows authorized users to define the roles and permissions and assign them to other users in the system to allow them access to the secured resource.

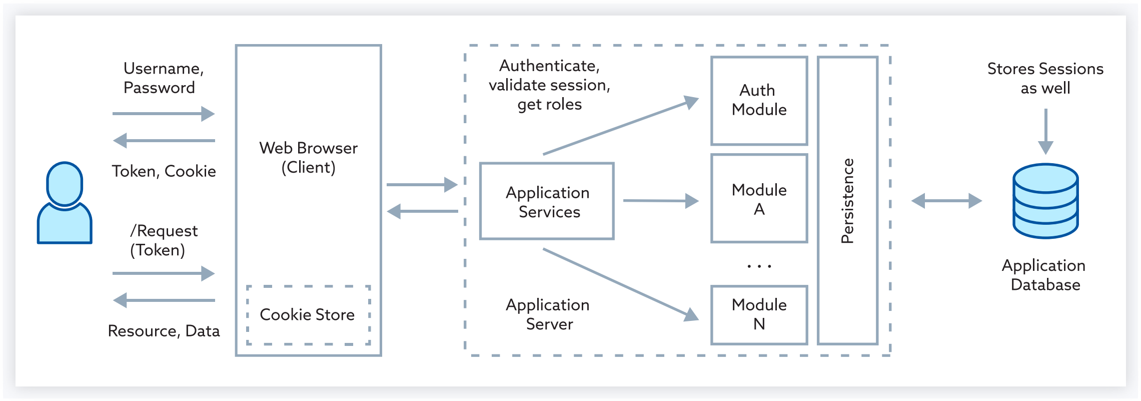


Figure 4: Microservices with Clojure, Packt Publishing

In a microservices-based application, each microservice is deployed in isolation and must not have the responsibility of maintaining a separate user database or session database. Authentication has to be two-fold; first is a user authenticating to a service, and second is application to application authentication which is usually achieved using JWT tokens and using mTLS. Moreover, there is a need for a standard way of authenticating and authorizing users across microservices; Open Policy Agent (OPA) can be used for the same. OPA decouples policy decision-making from policy enforcement. When a microservice needs to make a policy decision, it will query OPA and provide structured data (e.g. JSON) as the input. OPA accepts arbitrary structured data as input. OPA generates policy decisions by evaluating the query input and against policies and data. OPA and Rego are domain-agnostic, so microservices describe almost any kind of invariant in the policies (e.g, which users can access which resources, which subnets egress traffic is allowed to, which clusters a workload must be deployed to, which registries binaries can be downloaded from, which OS capabilities a container can execute with, and which times of day the system can be accessed). Policy decisions are not limited to simple yes/no or allow/deny answers. Like query inputs, the policies can generate arbitrary structured data as output. Ideally, it is essential to separate the authentication and authorization responsibility as a separate Auth service that can own the user database to authenticate and authorize users. In an enterprise context, user authentication across services can be done using enterprise Single Sign On (SSO) using ADFS or SAML 2.0. This also helps in authenticating the user once via the authentication service and then authorizing them to access the resources and related services through other microservices or within each microservice. Appropriate controls must be in place to protect the authentication services as they can be a prime target for stealing authentication credentials or tokens. Proper protection and detection should be in place to be able to detect such compromises.

Different types of tokens can be used for authentication and authorization (e.g. JWT, JSON Web Signature (JWS), JSON Web Encryption (JWE)). An encrypted token can be parsed only after it is decrypted with a key that was used to encrypt the token. Instead of sharing the key across microservices, it is beneficial to send the token to the Auth service directly to decrypt the token and authorize the request on behalf of the service. Performance impact of this can be reduced by caching the prevalidated tokens at each microservice level for a minimum configurable amount of time that can be decided based on the expiry time of the token. Expiration time is an important criteria while working with authentication tokens. Authentication tokens with a very large expiry time should be avoided to reduce the attacks that use stolen tokens. Some examples of authentication token formats can be found at JWT RFC-7519, JWS RFC-7515, and JWE RFC-7516. It is important to ensure the identity of the containers in which services are hosted; communication between two containers should use mTLS which authenticates both endpoints. This approach also helps in mitigating any man-in-the-middle attacks that may lead to token stealing and reusing them for authenticating the service.

2.2 Securing the API

Application Programming Interfaces (APIs) permit the sharing of functionality between applications over a defined interface between applications. If engineered appropriately, APIs expose the internals of the application in a controlled manner. API security involves the application of functional and non-functional software requirements necessary to guarantee the confidentiality, integrity, and availability of inputs and outputs. The terms “fidelity,” “veracity,” and “integrity” are synonyms, when recasted through the lens of the CIA triad, directly addresses security with respect to APIs.

Interface inputs and outputs can be shielded, internal to the applications’ various modules, or exposed outside the application boundary to serve as an interface available to receive and transmit data outward to other externalized interfaces.

All functional and nonfunctional requirements (NFR) should be supported by a common set of software architecture principles that guide the design. Traceability between requirements and the principles guiding the construction of an API is a functional requirement in itself.

API best practices that can serve as the foundation for security guidance include the following:

- Utilize both coarse- and fine-grained access control interface and data access in combination with token-based authorization and session-based access.
- By default, developers should practice service and interface segmentation.
- Developers should favor solutions that are highly partitioned, modular in design, and composed of decoupled services
- Standards-based messaging and encryption protocols for communication should be utilized.

- Interfaces and underlying executing logic need to remain loosely coupled from one another.
- The developer interface experience should be simplified – applications should operate with ease.
- Developers should provide robust error handling, service discoverability, and allow for visible and verifiable function.
- Developers should standardize backend API design decisions by proscribing coding standards for method usage (e.g. favor POST over GET) query parameters, data path, header contents, and declaring acceptable and unacceptable status codes.
- Developers should standardize front-end API design decisions by proscribing the expected service level agreements, service contracts, API mediation use cases (e.g. direct connect or inter-mediated), and acceptable styles (e.g. prefer REST, Accept SOAP with conditions, HATEOS with conditions, discourage RPC, accept Streaming for special uses).
- A versioning scheme and naming convention should be applied.
- Developers should bind API specifications to an agreed set of non-functional requirements and testable acceptance criteria.

2.2.1 SERVICE DISCOVERY

In a microservices architecture, business functions are packaged and deployed as services within containers and communicate with each other using API calls; it is recommended to implement a lightweight message bus that can implement different interaction styles. Other ways in which a service discovery service can be implemented span two dimensions: (a) the way clients access the service registry service and (b) centralized versus distributed service registry.

Enterprises have a choice of having a centralized service registry wherein all services are published and registered at one point. With that said, this approach can potentially become a single point of failure impacting confidentiality, integrity and availability. Compromise of the registry can result in the deployment of malicious services which appear legitimate to other services. Service registries should be protected from unauthorized users via appropriate access controls. Proper detective controls should also be in place for unauthorized changes to the service registry, as a malicious entity could potentially publish services which are not legitimate or approved for use and use them to compromise other services, thus impacting the integrity of the enterprise platforms.

2.2.2 THE API GATEWAY PATTERN

Since some microservices could be performing synchronous or asynchronous communication, it is important to have an API gateway to broker these services supporting assorted communication protocols. The API gateway should be integrated with an identity management service. Appropriate authentication and authorization of services should be conducted before allowing any connectivity or access to services. The API gateway should secure and send all traffic information to a central monitoring or detection system for detecting attacks.

2.3 Authorization and Access Control for Microservices

In order to ensure continued preservation of CIA/IAAA capabilities, an application in microservices is split into multiple processes. Each microservice implements the business logic of one module in the original single application. Therefore, after the application is split, each microservices' access request is to be authenticated and authorized.

The authentication and authorization in the microservices architecture, however, involves scenarios that are more complex. It involves both user-to-service communications as well as machine-to-service (app-to-service and service-to-service) communications [Medium, 2018].

Following are several microservices architectural solutions to solve the issues for authorization and authentication:

- Authorization Handling
- Authorization between Services
- API Access Control
- Firewall
- Secrets Management

2.3.1 AUTHORIZATION HANDLING

The term "authorization" refers to what one can do, for example access, edit or delete permissions to some documents, which takes place after verification passes.
Note that authorization is a separate step from authentication, which is about confirming the identity of who is taking the action.

Authorization is traditionally managed using:

- Rules: Sets of permitted verbs on a set of objects
- Roles: Collections of rules. Users and groups can be associated with, or bound to, multiple roles at the same time. Roles represent a privilege to access a protected resource or an operation.
- Bindings: Associations between users and/or groups with a role
- RBAC: Role Based Access Control. [Red Hat OpenShift]

In order to handle the problems of authorizations in the microservice architecture as mentioned above, the traditional authorization model can be enhanced by adding the concept of "entitlements."

While entitlements are now implemented in Docker as a proof of concept, these mechanisms are not ready to be used in production and are still a work in progress. [LWN.net, 2018]

Labels-based Identity. Labels define which containers can access which resources, requiring a container to have the label of any secret it accesses. Labels allow users to consult a predefined policy and authorize containers to receive secrets. Containers and their labels should be bound together using code integrity tools [Wei et al., 2018] to confirm that a specific container hash is valid. Using cryptography can help create strong identity and authentication on a per-container, per-pod or per-service level. [Unbound, 2018

Client Token with API Gateway. Here, contrary to the basic process of token authentication, the API Gateway is added as the entrance of the external request. This means that all requests go through the API gateway, effectively hiding the microservices. On request, the API gateway translates the original user token into an opaque token that can be resolved only by itself. The API gateway can revoke the user's token when it logs out. It also additionally protects auth token from being decrypted, by hiding it from the client, when they log off.

2.3.2 AUTHORIZATION BETWEEN SERVICES

By executing authorization in every microservice, fine-grained object permissions are possible as well as different user authentication mechanisms for different microservices.

Third-party Application Access
This is performed in three ways:

- API token
- OAuth
- Federation

API Token: The advantage that the API Token presents (instead of using the username/password directly to access the API) is to reduce the risk of exposing the users' password, and to reclaim the token's permissions at any time without needing to change the password. Here, the third party uses an application-issued API Token to access the applications' data. The Token is then generated by the user in the application and provided for use by third-party applications.

OAuth: One of the best approaches is OAuth delegation protocol with a JSON token (JWT - JSON Web Token). JWT is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

The OAuth authentication in microservices is provided by a well-known authentication provider, which simplifies the cumbersome registration operation. Here the microservice plays the role of the client in the OAuth architecture. This way the client/microservice interacts directly with the authorization server when applying for the access token. The authorization server authorizes the client when processing the client's token request and prevents others from forging the client's identity for the authentication code. [Medium, 2018]

The token can be signed with a "private/public key" method. This way, other microservices only need to contain the code for checking the signature and knowing the public key. As the token is sent with the authorization header as a bearer token, it can be evaluated by the microservices. The signature provides that there are no restrictions regarding URLs. Hence, cross-site authorization is also possible, which in return supports single sign-on (SSO) and is quite useful to users. [LeanIX, 2017]

OAUTH2 & OIDC Token: Suited for coarse-grained and functional access control, the OIDC is typically used in front-end services, while the ID token is addressed to the OIDC client. Only the OIDC client can validate the signature and the ID Token cannot be passed to another party. OAUTH2 uses scopes to determine authorization rights, while scope doesn't show who the user is but only the existing operational rights.

Federation: Federated security is a multi-purpose system that serves organizations as well as administrators and users. Federation can be done with same user credentials across multiple services or allow for Single Sign-on (SSO) across the different security domains. [Nordic APIS, 2017]

2.3.3 API ACCESS CONTROL

The API Gateway's best practice here is that of the "least privilege" which restricts access to only resources that a user requires to perform their immediate job functions.

An OAuth/OAuth2 server is commonly used by developers and administrators to obtain tokens for authentication with the API. For security reasons, it is better to ensure that all client-server communications are encrypted in transit with transport layer security (TLS).

Mutual TLS Client Certificate Bound Access Tokens
When mutual TLS is used by the client on the connection to the token endpoint, the authorization server is able to bind the issued access token to the client certificate. Such a binding is accomplished by associating the certificate with the token in a way that can be accessed by the protected resource, such as embedding the certificate hash in the issued access token directly or through token introspection. Binding the access token to the client certificate in that fashion has the benefit of decoupling that binding from the client's authentication with the authorization server. This enables mutual TLS during protected resource access to serve purely as a proof-of-possession mechanism. [IETF, 2018]
Proof-of-possession mitigates against token replay since the adversary would need both the token and the certificate to execute the attack.

API Broker Security Combined with Load Balancing
In order to ensure microservices are addressable and secure at the same time, one best practice is to rely on an API manager or broker to provide security. The API manager/broker combines security with load balancing and ensures the work is partitioned so that it is manageable and service outages are limited. It is important to have all the API brokers and microservices placed in a common subnet for as long as the broker supplies a security token to the calling application. When the broker makes the call to the microservice itself, then the microservices should be placed in their own subnetwork and the only traffic to be allowed is the one from the API broker that accesses the microservices. [TechTarget, 2018]

SPIFEE/SPIRE
SPIFFE (pronounced Spiffy) stands for Secure Production Identity Framework For Everyone. It is an open-source workload identity framework that supports distributed systems deployed in on-premises, private cloud, and public cloud environments. SPIFFE provides a secure identity, in the form of a specially crafted X.509 certificate, to every workload in a modern production environment.

SPIFFE removes the need for application-level authentication and complex network-level ACL configuration.

SPIRE (the SPIFFE Runtime Environment) is a software system that exposes an API (the SPIFFE Workload API) to other running software systems (workloads) so they can retrieve their identity, as well as documents (SVIDs) to prove it to other workloads, at run-time. This proof of identity can then be used as the primary authentication mechanism for a workload when it accesses other systems. SPIRE is an open source and it enables organizations to provision, deploy, and manage SPIFFE identities throughout their heterogeneous production infrastructure.

The Workload API here (similar to other systems' APIs), does not require a calling workload to have any a priori knowledge of its own identity or possess any authentication token when calling the API. This avoids the need to co-deploy any authentication secrets with the workload.

In addition and in comparison to other APIs, however, the SPIRE API runs on and across multiple platforms and can also identify running services at a process level as well as a kernel level - which makes it suitable for use with container schedulers such as Kubernetes.

In SPIRE, all private keys (and corresponding certificates) are short-lived, rotated frequently and automatically in order to minimize the possibility of a key being leaked or compromised. Workloads can request new keys and trust bundles from the Workload API before the corresponding key(s) expire.

Concretely, the workload API allows a workload to retrieve the following:

- Its' identity, described as a SPIFFE ID.
- A private key tied to that ID that can be used to sign data on behalf of the workload.
- A corresponding X.509 certificate is also created, the X509-SVID. This can be used to establish mTLS, sign a JWT token, or otherwise authenticate to other workloads.
- A set of certificates that can be used by the workload to verify the identity of other workloads (a trust bundle.) [SPIRE]

Firewall
In order to have a much smaller attack surface, platforms that allow controlling egress traffic with a firewall are considered. [Apriorit, 2018]

A distributed firewall (a service mesh) with centralized control which allows users more granular control over each and every microservice is also an important component. In this way the developers are able to define with fine granularity which connections are allowed and which are not. This practically means that each service has its' own micro-firewall. Hence, if a service is breached, the rest of them remain secure. [Project Calico]

Security policies that allow fine-grained security of containers and services, protecting services from each other and protecting the orchestrator itself, are a good approach to keep services secure. Operators can turn the high-level security policy definitions into a set of firewall rules for each container, and then verify and maintain the firewall rules for each of those containers. Lastly, operators are able to immediately manage and coordinate all firewall rule changes as containers start, stop, and security policies are updated. [Weaveworks]

2.3.4 SECRETS MANAGEMENT

Secrets are credentials like API tokens, SSH keys, passwords, etc. which a service needs to authenticate and communicate with other services.

Authorization for APIs need to be implemented in a distributed manner, which can be challenging, and secrets management and key distribution for distributed applications also introduce another headache for the developer. Putting secrets in the container image expose it to many users and processes and puts it in jeopardy of being misused.

Automated credential management systems lean toward a Security by Design approach to create microservices environments. This gives developers powerful tools (i.e. DevOps Audit Toolkit[13]) to automate secret management immediately while maintaining critical separation of duties.

Developer/Operator best practices for secrets management include:

> - Considering centrally managing secrets and the container access to secrets and view which containers are using those secrets in real time.
> - Injecting secrets into the container at runtime, and ensuring that the secrets are stored in memory and accessible only to the designated container.
> - Ensuring that the secret is delivered encrypted to the container that needs it. [Aquablog, 2019]

In order to centrally manage secrets, a tool for secrets management, encryption as a service, and privileged access management can be used.

Secrets Management guides to securely store secrets include the following:

- **Static Secrets** control who can access them. Secrets should be encrypted prior to writing them to persistent storage so gaining access to the raw storage is not enough to access them.
- **Secret Engines** store, generate or encrypt data. Some secret engines simply store and read data; other secret engines connect to other services and generate dynamic credentials on demand. Data should be encrypted and decrypted without storing them. This allows security teams to define encryption parameters and developers to store encrypted data in a location such as SQL without having to design their own encryption methods.
- **Secret as a Service:** Dynamic Secrets generate database credentials on-demand so that each application or system can obtain its own credentials, and its permissions can be tightly controlled. After creating the dynamic secrets, they are automatically revoked after the lease is up.
- **Database Root Credential Rotation** enables the rotation of the database root credentials for those managed by a system. The longer a secret exists, the higher the chance for it to be compromised. Frequent rotation helps to reduce the secret lifespan and with that the risk of exposure.

[13] Separation of Duties in the DevOps Audit Toolkit: https://www.oreilly.com/library/view/devops-sec/9781491971413/ch05.html

- **Cubbyhole Response Wrapping** is a secure method to distribute secrets by wrapping them where only the expecting client can unwrap. It is not possible to reach into another token's cubbyhole even as the root use opposite to the key/value secrets engine, where there is accessibility to any token for as long as its policy allows it.
- **One-Time SSH Password** makes use of SSH secrets engine to generate a one-time password (OTP) every time a client wants to SSH into a remote host.
- **Build Your Own Certificate Authority** through the use of the PKI secrets engine to generate dynamic X.509 certificates.
- **Direct Application Integration** includes the usage of Consul Template and Envconsul tools to retrieve secrets from the system with no or minimum code changes to the applications.

2.4 Secure Deployment Styles and Strategies in Microservices Architecture

Microservices have four main deployment styles which can be aligned to the Cloud Services Provider (CSP) model. These styles and services are Platform as a Service (PaaS) Serverless (FaaS), Containers (CaaS) and Virtual Machines served by an IaaS model. The picture below depicts the platform spectrum and examples of the same:

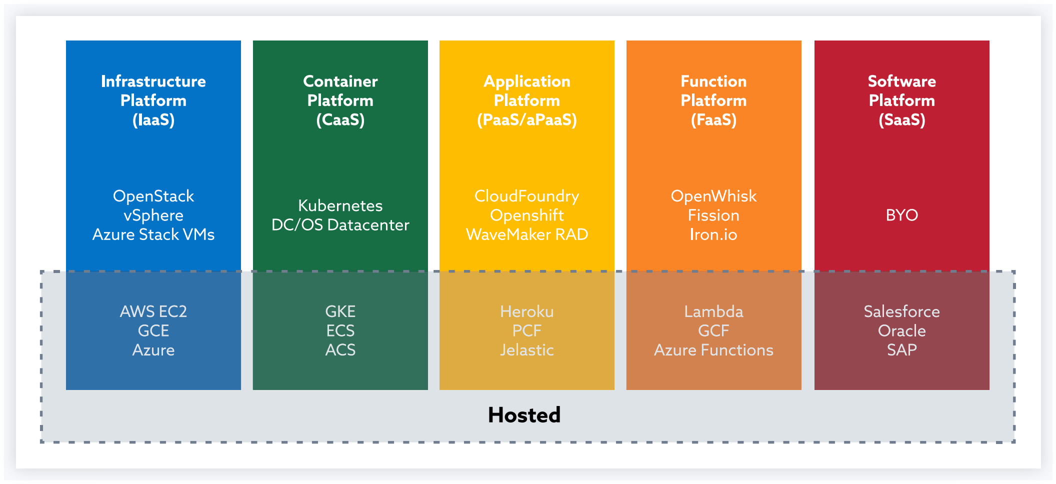


Figure 6: Platform Spectrum of Microservices Deployment Styles
[Picture Reference : http:www.Mesosphere.com]

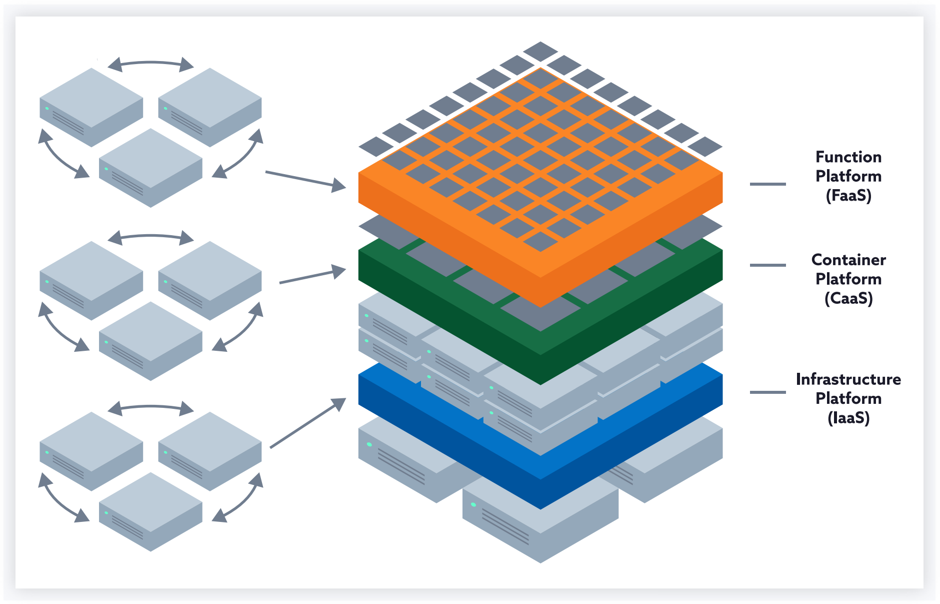


Figure 7: Microservices Deployment Styles and Services
[Picture Reference : http:www.Mesosphere.com]

As with the cloud services model, Microservices have a shared responsibility between customer and CSP depending on the deployment style chosen; this is shown in the microservices responsibility model below. However, customer responsibility and cloud location are not the only decisions involved in the style and strategy of microservices deployment. Other considerations of the deployment are speed, portability and control. Balancing these factors will be key in the style of deployment. Strategy around the deployment will be based heavily on the workload of the microservice and its suitability for the organization to move it to the Public or Private cloud exposing APIs for integration.

Other types of Microservices deployment patterns:

- Multiple instances of microservices per host
- A single instance of a microservice per host
- A single instance of a microservice per VM
- A single instance of a microservice per container
- Microservices deployment platforms
- Serverless deployment

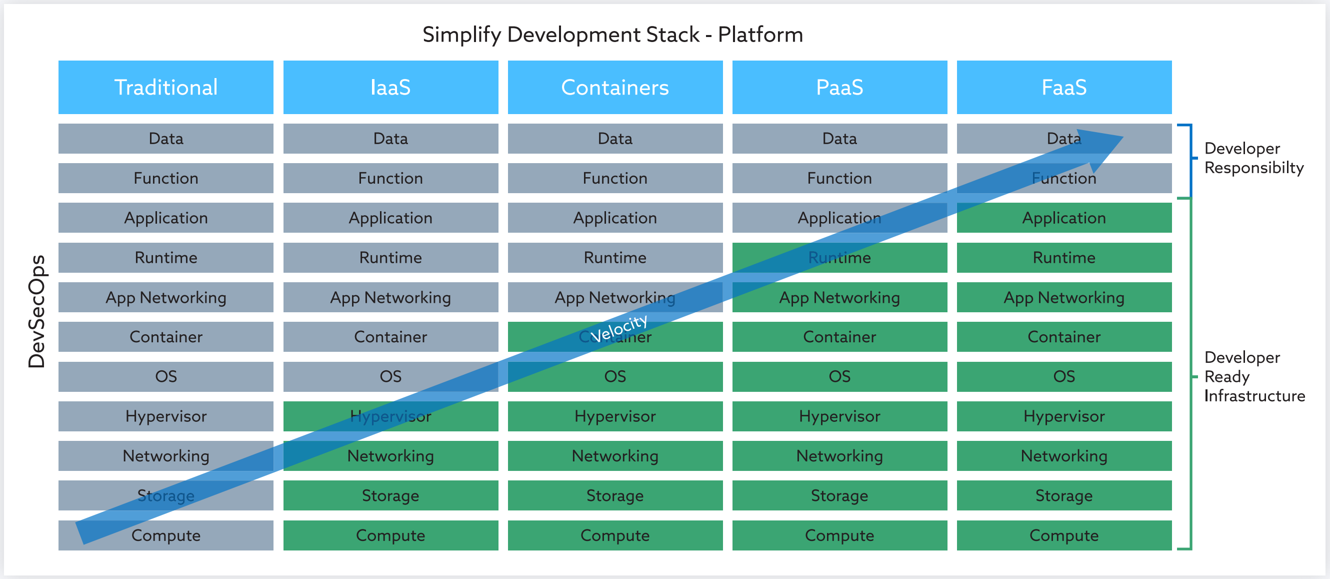


Figure 8: Microservices Responsibility Model
(Reference: https://blogs.vmware.com/vov/files/2018/08/development-stack.png)

2.5 Stateful and Stateless Microservices Security

Stateful Microservices: Applications composed of microservices can contain both stateless and stateful services. It is important to understand the constraints and limitations of implementing stateful services. Stateful microservices make rapid scaling and environment repaving much more difficult because each microservice instance effectively becomes a "snowflake" with its own unique state that should be accounted for by the system as a whole. Security in a dynamic containerized environment relies heavily on the ability to rapidly destroy and recreate containers from a known good image. This implies microservices should be stateless by design; hence, stateless design is a core microservice best practice. Developers should exercise caution before choosing to implement stateful microservices. If a service relies on the state, it should be separated into a dedicated storage layer that is easily accessible.

A microservices-based application may contain stateful services in the form of a relational database management system (RDBMS), Not only Sequential Query Language (NoSQL) databases, and file systems. These microservices are packaged as containers with unique attributes. Technologies supporting creation of a separate persistence layer that is not host-dependent should be used. This helps address the host portability problem and maximize availability with minimal delay on consistency.

Typically, stateful services offload persistence to the host or use highly available cloud data stores to provide a persistence layer. Both approaches introduce complications: offloading to the host makes it difficult to port containers from one host to another, and highly available data stores trade consistency for availability, meaning that developers have to design for eventual consistency in their data model.

Stateless Microservices: One of the key advantages of microservices is the ability to scale rapidly. Like other distributed computing architectures, microservices scale better when they are stateless. Ease of scaling supports service-wide availability.

In a stateless scenario, microservices do not keep a record of state, making it difficult to manage session level security.

As the expiration time is information embedded in the token, it means the expiration of active sessions cannot be revoked. One of the best practices to deal with this concern is to define short expiration times for each token along with managing the confidentiality of the token.

Since the security module of microservice architecture is mostly responsible to renew the token at each request, this should not pose a problem for service consumers.

2.6 Container Storage Interfaces

Databases store data to an underlying storage platform. Application developers may view a persistent storage system as being the database itself. But behind the scenes the database is writing to either block, file, or object storage. A container orchestration system manages application containers. The Container Storage Interface (CSI) is a common interface layer between the container orchestrator and the back end storage system. The idea is that storage providers will need to implement just one plugin and container orchestrators will only need to support CSI.

To make the above happen, it is typical to have a container store data by attaching to storage outside the container. This can be achieved by adding storage and volumes for a container. Adding storage space to a host does not automatically make more storage available to containers running on that host. There are, however, ways to mount and use host storage volumes within a container. Volumes mounted on one container can also be made available to other containers. These are called data volume containers, allowing services to share a volume from a source container with one or more target containers. This approach has some advantages including sharing persistent storage across several containers and providing a layer of abstraction around the bind mount.

Features for mounting storage volumes to containers allow developers to keep containers simple and portable, while making the data accessible outside each container. Because images and containers themselves consume space on the host system, being able to expand disk space on the host is beneficial.

2.7 Runtime Security

Runtime security and policy enforcement is critical in a microservices environment. Runtime protection is another concept of limiting the set of code that can be executed within the container itself. If attackers can access a container, but cannot execute their own code within it, the potential damage is limited.

Traditional VM tools do not support containers and cannot provide detection for containers. There is a new class of container security tools available which help with runtime detection and enforcement of policies. These tools are able to detect container breakouts, network policy enforcement, host OS and Kernel configuration changes and are able to conduct runtime vulnerability scans, etc. Some of the other policies to consider are to establish normal application behavior and then enact security policies based on that (i.e. prevent deployment if a container requires root access), monitor every container for anomalies and the image registry to automatically replace affected images, perform live scans of running containers and hosts and block activity when something deviates, store data on all security events and perform analysis to determine the root cause and prevent future attacks. Some of these tools also provide scanning capabilities for container registries as well as runtime image scans and third-party image scans.

In short, at runtime, it is important to following these best practices for effective security:

- Systems should be able to pull only from trusted image repositories.
- Pulls of unsigned images should be denied.
- Resource usage should be monitored and anomalous usage investigated.
- Containers should not allow direct remote access (i.e. SSH) and instead rely on a versioned artifact store and source control to track changes.
- System calls by a container should be monitored and anomalous calls investigated.
- Containers should have a max time to live and be regularly rotated.
- Read-only containers should be used if possible.
- Privileged users in the container should be mapped to non-privileged users on the host.
- Mandatory Access Control (i.e. SELinux/AppArmor) policies should be enabled and configured.
- Microservice and containers behavior should be analyzed.
- Relate distributed threat indicators and ascertain whether a single container is contaminated or is spread across many containers.
- Intercept and block unauthorized container engine commands.
- Actions for responses should be automated.

3.0 Microservices Secure Development and Governance

For microservices and security to co-exist, a framework and plan for development, governance, and management of microservices must be developed. Developing microservices requires planning out services and their associated integrations, functionality and components. Governance and management of microservices to ensure all functional and security policies are enforced in development as well as at runtime is essential.

3.1 Container Security Best Practices in Microservices

3.1.1 CONTAINER SECURE BUILD AND DEPLOYMENT

The following best practices assume that readers have selected a microservices deployment model leveraging containers and followed the best practices outlined in chapter two of this document.

Servers should be Stateless
Servers should be treated as interchangeable members of a group, particularly those that run customer-facing code. They all perform the same functions, so Site Reliability Engineering (SRE) teams do not have to be concerned about them individually. The only concern should be scalability (i.e. are there enough servers) to support the workload. It is recommended to employ auto-scaling, self-healing and fault tolerant infrastructure so that if one server stops working, it can be automatically replaced by another one. Using servers for specialized functions should be avoided.

Image Security in Build and Runtime
It is important to use container-specific operating systems (OSs) employing the minimum required set of services and software to run said container, therein reducing the OS attack surface. Operators should employ read-only file systems where practicable. Continually updating OSs to mitigate new container vulnerabilities is critical to protect the security of the environment. An example of such an OS is open source Weave Firekube using Weave Ignite.

Developers and Operators should ensure microservices performing similar functions and similar sensitivities are hosted together to segregate them from apps with different sensitivities. Consistent policies and controls should be applied based on the sensitivity of applications running on containers on the same host as well as across the fleet.

Most vulnerability scanning tools are built for VMs; for containers it is critical to use tools that support scanning of containers. Vulnerabilities must be actively addressed to reduce replication of the same vulnerabilities at image/container deployment.

In a container build process, an applications' components are built and placed into an image. The image is a package that contains all the files required to run a container. An app image should include only the executables and libraries required by the app, with all required OS functionality provided by the underlying OS Images. App developers should use layering and master images in read-only mode where practicable so no one can change the master image and introduce vulnerabilities. Changes required to the master image should go through the CI/CD pipeline and security validation processes. The goal should be to simplify the build process for application developers so they do not have to worry about the underlying infrastructure and patching.

The build creation process is usually managed by developers, and as such, they are responsible for packaging an app for handoff to testing and validation. Security bug bars and quality gates should be employed and enforced in the CI/CD pipeline before a build can be promoted to the next stage. Known as DevSecOps, this paradigm incorporates security from inception, instead of adding it after the application is completed. "Shift Left" security ensures developers address security and quality issues at the earliest stages of the pipeline, with secure, high quality code deployed into production to the farthest right. Having feedback loops at every stage of this pipeline and employing automation and remediation for previous steps decreases the probability of security defects reaching production.

In a Software Development Lifecycle (SDLC) pipeline, this can be achieved using a combination of methods:

- Extending unit tests to ensure the application behaves as expected.
- Validating the security of the supply chain and ensuring that third-party components are assessed and lineaged.
- Making sure the systems are appropriately hardened and tested for misconfigurations and insecure default configurations.
- Adding dynamic and static testing to the pipeline, evaluating the most significant application risks, and enabling visibility and training for developers.
- Adapt and add OWASP Foundation checklists to the pipeline

After image creation, images should be tested and accredited using test automation tools to validate the functionality of the final application. Security teams should certify the images by ensuring all mitigations have been applied that were found in static and dynamic code scans, and ensuring final images are digitally signed. If there are vulnerabilities identified in the application services which do not align with the security bug bars, then the complete build should fail. Ideally, this should be enforced through an automated CI/CD pipeline with enforced security bug bars. While tools are the preferred method, human verification is vital as tools are not infallible and fail to understand context. Context is paramount to assess security defects in code reviews correctly, as a human can triage and accurately calculate the risk to the enterprise. As such, using a code review framework can improve collaboration, reduce security defects, and enhance standardization. There are several types of code reviews; they generally fall into either formal code reviews, (e.g. Fagan inspection, or lightweight code reviews; synchronous, asynchronous, or instant code reviews).

Image Storage
Storing and accessing (IAM permission) images in a central location helps with easy management of the same and also for deployment to multiple environments as needed. Image registries allow developers to easily store images as they are created, tag and catalog images for identification and version control to aid in discovery and reuse, as well as find and download images that others developers may have created. Registries may be private or third-party provided public registries. Registries provide APIs for automation (e.g. automating the push of images which have been tested and have met the security and quality bug bars)

It needs to be noted that third-party public repositories cede control over the supply chain to an external party. This may be acceptable but needs to be thoroughly assessed for risk so an organization can understand and have the opportunity to mitigate the risk they are taking on.

Third-party Image Security
Container image registries are centralized repositories used for sharing images. Container images are made available using community-based public registries such as Docker Hub or enterprise registries such as Quay, Docker Trusted Registry (DTR), AWS Elastic Container Registry (ECR), Google Container Registry, Microsoft Azure Container Registry, etc.

Security of images at rest for most enterprise registries is done by the service provider as part of their software suite. For example, Docker DTR has Docker Security Scanning, which employs active image scanning. In the case of public images, it is recommended to follow best practices such as downloading content from trusted user(s), scanning images before deployment, and using curated official images that have certified content, even if the images are built and consumed locally. In case of enterprise image repositories, it is important to implement an audit process of the registry to ensure that older images and those with older known dependencies are flagged and that interdependencies are captured. These images, if downloaded from a third-party repository, should use a TLS certificate issued by a known Certificate Authority (CA). As a security best practice, it is not recommended to use self-signed certificates or use registries over unencrypted HTTP connections, as doing so may lead to tampering of data in transit. Additionally, image authenticity should be validated and signature verification should be performed before any image is downloaded.

Enterprises typically have a central artifact repository that provides a single source of truth for all components of an application. This allows for management of build artifacts and release candidates in one central location, transparency of component quality and security, management of licenses, modernizes software development in phases like staging and release functionality, and helps scale DevSecOps delivery with high availability and clustering.

Security of the CI/CD Pipeline

A Continuous Integration / Continuous Deployment (CI/CD) pipeline is intended to reduce errors during integration and deployment while increasing release cycles for secure code. While all implementations will be different, adhering to best practices increases an enterprises' security posture. Isolation of CI/CD systems is paramount, as this system has complete access to the codebase and credentials in different stages of development. Isolation can be accomplished using various means and at different implementation layers. For example, CI/CD systems should be deployed to internal, unexposed networks. Access to these systems should be highly restricted with an automated monitoring mechanism in place. Depending on the complexity of the system, the repository of the system should be secured using an automated monitoring mechanism.

Security testing of code is an integral part of the CI/CD pipeline. There should be a gatekeeper process in the pipeline that enables secure testing of code from one stage to another. Apart from this, developers should be enabled to run as much testing locally prior to committing to the shared repository. Early detection is made possible by allowing tests to be run early and often by developers alongside their code changes.

The CI/CD system itself should be containerized. An ephemeral environment possesses various advantages: it executes the security testing on containers which abstracts the host systems and provides standard APIs. This approach ensures that residual side effects from testing are not inherited by subsequent runs of the test cases. It should be noted that stateless microservices make testing much easier; therefore, they are more likely to be verifiably secure when delivered for deployment.

It is critical to ensure that CI/CD pipelines are isolated from the rest of the network and that developers are not able to circumvent the pipeline partially or completely. At the same time, identity and access management of the build pipeline is a critical aspect to implement. Appropriate authentication and access permissions should be set up based on the roles of the Development, Security, Operations and QA / Test teams appropriately.

In a container environment it is very easy to inadvertently replicate vulnerabilities as most application containers use the Master image for builds. Hence the security validation of the master image is of prime importance. In addition, ensuring proper access controls on who can update the master image and proper version control is critical. Restricting access to approved registries for any image downloads is also essential. In enterprise scenarios, it is prudent to create a private container registry, and anything being added to that registry should go through proper security testing and validation. This prevents developers from downloading container images from the internet and replicating vulnerabilities accidentally.

3.2 Microservices Detective Controls

Given that protective controls (controls that are designed to prevent an incident from occurring) such as access, authentication, key storage and encryption to name a few are addressed above, this section will focus primarily though not wholly on detective controls (controls that are designed to identify, characterize, provide an alert to an incident in progress as well resolve incidents).

As opposed to monolithic applications, where components operate and communicate internally and present a single point of failure, Microservice applications operate and communicate both internally and externally. This decomposition or breaking down of the application into individual services, while increasing the application's resiliency, increases the attack surface that needs to be monitored and protected.

Additionally, decomposing an application into separate components, which gives Microservices its advantages in speed, resiliency, flexibility and scalability, also serves to provide security challenges with respect to the orchestrated distributed system. To meet these security challenges, secure development, implementation, operations and ongoing monitoring need to consider Policies, Logging, Monitoring and Alerts as well as Incident Management and Platforms.

3.2.1 POLICIES

Policies are a necessary precursor to logging, monitoring, alerts and incident resolution as they establish a baseline for action. Policies are standards that a monitoring system uses to evaluate behavior, events, and status. Because applications have been broken down into components as microservices, which may or may not be stored in containers, the specific function and behavior of this combination or separation can be well understood and therefore predicted. This makes setting specific operational policies and the monitoring of malicious behavior (deviation from standard behavior) possible. The implementation of any security policies should be defined as part of the configuration artifact generated for execution (configuration files should have the policies configured in them for any tools to enforce them). These policies can include microsegmentation, communication, configurations, interfaces, logging and alerting targets, etc.

Developer and Operators should employ the following best policy practices:

1. Set policies at the OS, network, host, container, and application/microservice levels;
2. Protect services using service-level policies to ensure security regardless of the number of containers, microservices/applications, etc.;
3. Set up a central configuration server from which microservices can load properties files via discovery;

4. Employ a policy management platform at the OS, networking, container, microservice component and application levels. Determine which policies to use, their parameters, and how they are to be enforced and implemented in the management system;
5. Use granular policies to detect images that do not pass security requirements such as any image affected by a CVE of a particular rating;
6. Coordinate image scans with runtime enforcement and remediation capabilities.

3.2.2 LOGGING

One of the most important, if not the most important, tool available to determine what has happened in a system taking advantage of a microservice/container strategy is the logging system. Logging is a critical part of keeping microservices alive and healthy. Logs contain valuable information such as stack traces and information about where data is coming from and can help with reassembly such as in the event of a server crash. SSHing into a machine and looking at STDOUT may work for a single running instance, but in the world of highly available and redundant microservices, there is a need to aggregate all those logs to a common place and query the same based on different criteria. For example, if a single instance is bad, a SRE (site reliability engineer) might want to look at the process ID (PID), hostname, and port. Aggregated logs provide unique insight and context given they originate from services owned by different teams within the organization.

A log file is a record of what occurred (an event, transaction, message). A logging solution provides for storing, recording, analyzing and reporting of these actions as they occur. One of the major challenges to logging microservice/container activity is the ephemeral nature of this distributed architecture. Simply stated, containers and the contained microservices do not exist indefinitely and a well architected logging system needs to consider this.

Developers and Operators should employ the following best logging practices:

1. Tag all microservice request calls with unique IDs so any error can be traced to that call and back to the server/container and or microservice (application) from which the error originated even after destruction;
2. Code error responses with a unique ID so they (container and microservices errors) can more easily be grouped and analyzed;
3. Structure Container and Microservice log data in a standard format (e.g. JSON);
4. Make all log fields in the chosen standard format searchable;
5. Log UTC time to order messages for aggregation, analysis and reporting purposes;
6. Log more data then less to avoid missing important information, but if logging creates excessive demands due to container and microservice data volumes, then developers and operators should be prepared to cut back and/or work with offline storage and analysis tools;
7. View the logs as a stream of data flowing via a dedicated log shipping container;
8. Forward, via a dedicated log shipping container, all logs to a centralized location to make container and microservice activity reporting and monitoring easier;
9. Store logs externally to host due to container microservice unavailability (e.g. if container is destroyed) and storage space resource availability;
10. Employ a container and microservices purposed (fast, handle large data volumes, using visualization and AI) tool to store, aggregate and report log activity.

3.2.3 MONITORING SYSTEM AND ALERTS

Monitoring is used to ensure the system is operating as designed and the data is secure. Again, a major challenge to monitoring and alerting microservice / container activity is the fact that containers and microservices do not exist indefinitely.

Additionally, there could be hundreds to thousands of containers and microservices running at any one time across distributed systems. Monitoring is accomplished via a combination of system policies and logging as described above and presented in dashboards and reports which collect, aggregate, summarize and interpret information such as events or status.

Another major challenge is that traditional tools which were used for monitoring and alerting on VMs are not very effective for containers; hence it is pertinent to implement tools which are container aware and have visibility of the whole ecosystem, can detect malware infections, perform runtime validation of container images, enforce network microsegmentation policies, identify container breakouts, enforce resource policies, etc.

Alerts are messages/notifications generated by monitoring systems that are sent to responsible parties notifying them of events or status that need to be reviewed in order to determine the who, what, where, when and why and mitigating actions if any are necessary.

Developers and Operators should employ the following best Monitoring Systems & Alerts practices:

1. Construct an environment of end-to-end continuous visibility at the host, container and microservice levels;
2. Restrict and monitor container/microservice commands such as create, run, kill and launch or syscalls and seccomp;
3. Identify and monitor key metrics at the host, infrastructure, and microservice levels;
4. Use services registry to hold the location of available microservices during runtime;
5. Use either client-side or server-side discovery to locate available microservices;
6. Monitor clusters when multiple containers and microservices are “in play”;
7. Correlate distributed (multiple hosts, infrastructures and microservice) threats;
8. Implement large scale, fast correlation due to the voluminous, distributed nature of container/microservice information;
9. Analyze container/microservice behavior against baselines to detect malicious activity;
10. Analyze container/microservice volume demands using AI (Predictive Analytics) to correlate behavior, pattern and classification;
11. Implement dashboards that are easy to interpret, and contain all key metrics;
    a. Have the ability to drill down from dashboard metrics to host, infrastructure and microservice logs;
    b. Have the ability to drill up from host, infrastructure and microservice logs to dashboard metrics;
12. Direct alerts immediately to the appropriate responsible according to the alerts assigned to microservice team members using DevSecOps;
13. Create alerts that are clear, actionable, and contain direction for resolution.

3.2.4 INCIDENT MANAGEMENT/FORENSICS AND PLATFORMS

Anyone who has ever dealt with information security knows the prevalence of security incidents and how incredibly stressful they are for everyone involved. At the same time, preparation with tools, techniques for identifying incidents, and the ability to contain and respond to them quickly is just as critical.

Just as technologies evolve, so do the challenges faced by incident response teams and digital forensic investigators. Answers can be found in many places, not just on the hard disks of computers or servers. Cloud storage and services, while highly convenient and cost-effective from a business perspective, are a game changer from an evidence preservation perspective.

There is a definite overlap between the realms of incident response and digital forensics. Knowing how to recognize this overlap and understand when an incident response is turning into a digital forensics investigation, or, conversely, when an investigation requires that an incident response to be triggered, will unquestionably help teams to deliver value to respective organizations. It is important to balance risk in other aspects of information security; hence one should also balance the risk of not completing a digital forensics investigation as a follow-up to an incident.

Containers are designed to be ephemeral, and in fact, multiple container security products play up this property as a security feature. If a container runs for only five minutes, even if an attacker compromises it, they will have access for only five minutes at a time. This property of containers runs contrary to the fundamental forensics need to preserve evidence. Container images that start and stop constantly represent not just moving targets, but targets that frequently cease to exist. However, the majority of container platforms use a copy-on-write file system, which helps forensic investigators tremendously when they are working with a running container. The underlying container image is stored in one location; this image contains the configuration data and applications that form the container image. Any changes made while the container is running will be written to a separate file and can actually be committed into a new image on the fly, without affecting the running container. If a container is believed to be compromised, a forensic investigator can run that newly committed image to explore its contents. It must be noted that such an action results in the creation of a new container from the image, not the exact same copy of the container that was originally committed. This differs from a VM snapshotting approach, since running processes in the target container are not included in the image.

There are several examples of vulnerabilities that would permit malware to escape a container image and access resources on the host machine. Security updates to container management platforms are frequent; as soon as a bug is discovered it is patched. Therefore, if a container is believed to be affected by some sort of malware, or other malicious actor, an investigator may consider simply treating the container as 'just another compromised application' and can image the entire host machine.

Applications broken down into possibly tens if not hundreds of microservices that operate with different life cycles that may interact with one or more other microservices which may or may not also be hosted in containers in the cloud and do not exist indefinitely make the identification, logging, categorization, prioritization, investigation, diagnosis, escalation, work around or resolution and closure of incidents a challenge. This challenge can be most efficiently and effectively addressed with a platform that combines monitoring with artificial intelligence.

Developers and Operators should employ the following Incident and Forensic best practices:

1. Capture all microservice infrastructure activity to produce normative baselines;
2. Capture all images, orchestrators, hosts, containers, events, transactions, and messages;
3. Scan all images and coordinating this runtime policy enforcement and remediation capabilities;
4. Store all information offline for later forensic review;
5. Drill up or down from dashboard metrics to host, infrastructure and microservice logs;
6. Combine the above best practices and apply AI (Bayes Classifiers) to detect incidents before, during and after occurrence;
7. Apply AI to the incident to determine severity, proper routing and delivery of possible remediation scenarios;
8. Employ alert monitoring and notify those responsible for responding to incidents. The teams should mirror the microservices;
9. Ensure DevSecOps automates incident handling as much as possible, enabling the monitoring platform to isolate alerts, metrics and dashboards by microservice teams;
10. Establish an incident management strategy. Site Reliability Engineering (SRE) has an incident response as the basis of reliability which can result in a reduced recovery time while providing support teams structure and familiarity to respond to stressful events;
11. Report after incident resolution to DevSecOps and others as needed and ensure that any fix is actually performed by the right person(s) including policy changes.

3.2.5 THIRD-PARTY INFORMATION EXCHANGE

As part of proper security measures, segmentation based on whitelisting or blacklisting is used extensively; with microservices and their architectural differences, operators can take advantage of the same methods. Microservices can expose public endpoints that can make them vulnerable to Advanced Persistent Threats and other bad actors trying to gain access. The challenge with this type of data is verifying and standardizing it.

Some companies (e.g. IBM X-Force, Facebook ThreatExchange, and AlienVault) provide access to these platforms. Open Threat Exchange (OTX) offers free access to threat intelligence as well. Microservices can leverage third-party community-powered threat data, to microsegment the various services, effectively enabling collaborative defense.

3.2.6 INCLUDING A UNIQUE ID IN THE RESPONSE

There are bound to be times when the microservices' users will be facing an error. It is important to know what caused that error. Hence, developers should code the response the client receives so that it contains a unique ID along with any other useful information about the error. This unique ID could be the same one that is used to correlate the requests. Having a unique ID in the response payload of the request will help the consuming service identify problems more quickly. The parameters of the request-date, time, and other details will help incident responders better understand the problem. It is also recommended to provide access to the data to support engineers via an open-source search and analytics engine.

3.2.7 STRUCTURE LOG DATA

It is almost impossible to have a defined format for log data; some logs might need more fields than others, and those that do not need all those excess fields will be wasting bytes. Microservice architectures address this issue by using different technology stacks, which impacts the log format of each service. One service might use a comma to separate fields, while others use pipes or spaces.

All of this can get complicated. Hence it is important to simplify the parsing of logs by structuring log data into a standard format like JavaScript Object Notation (JSON). JSON allows one to have multiple levels of data so that, when necessary, analysts can get more semantic info in a single log event. Or one can use schema-on-read to format semi structured data on the fly.

Parsing is also more straightforward than dealing with particular log formats. With structured data, the format of logs is standard, even though logs could have different fields.

Contextualizing Logs

Developers and Operators should employ the following log best practices, ensuring logs include:

- Date and time;
- Stack errors. One could pass the exception object as a parameter to the logging library;
- The service name or code, to differentiate which logs are from which microservice;
- The function, class, or file name where the error occurred so that service analysts don't have to guess where the problem is;
- External service interaction names: developers will know which call to the DB was the one with the problem;
- The IP address of the server and client request. This information will make it easy to spot an unhealthy server or identify DOS or DDOS attacks;
- User-agent of the application so that which browsers or users are having issues are identified;
- HTTP code to get more semantics of the error. These codes will be useful to create alerts.
- Contextualizing the request with logs will save time when troubleshooting problems in the system.

3.3 Microservices Messaging Patterns

In a microservices architecture, services are autonomous and communicate over the network to cater to a business need. A collection of such services forms a system, and the consumers often interact with those systems. Therefore, a microservices-based application can be considered a distributed system running multiple services on different network locations. A given service runs on its own process. So, microservices interact using inter-process or inter-service communication styles.

Microservice communications styles are mainly about how services send or receive data from one service to the other. The most common type of communication styles used in microservices are synchronous and asynchronous.

Synchronous Communication and Protocols
In synchronous communication, a client sends a request and waits for a response from the service. Both parties have to keep the connection open until the client receives the response. The execution logic of the client cannot proceed without the response. For example, HTTP is a synchronous protocol. The client sends a request and pauses for a response from the service. The request is sovereign of the client code execution which could be synchronous (thread is blocked) or asynchronous (thread is not blocked, and the response will reach a callback eventually). The important aspect here is the protocol (HTTP/HTTPS) remains synchronous and the customer code can continue its task only when it receives the HTTP server response.

REST doesn't depend on any of the implementation protocols, but the most common implementation is the HTTP application protocol. When a user accesses RESTful resources with the HTTP protocol, the URL of the resource serves as the resource identifier and GET, PUT, DELETE, POST, and HEAD are the standard HTTP operations to be performed on that resource. The REST architecture style is inherently based on synchronous messaging.

Asynchronous Communication and Protocols
In early implementations of the microservices architecture, synchronous communication was embraced as the de-facto inter-service communication style. However, asynchronous communication between microservices is becoming increasingly popular as it makes services more autonomous.

In asynchronous communication, the client does not wait for a response in a timely manner. The client may not receive a response at all, or the response will be received asynchronously via a different channel. This messaging between microservices is implemented using a lightweight and dumb message broker. There is no business logic in the broker and it is a centralized entity with high-availability. There are two main types of asynchronous messaging styles—single receiver and multiple receivers.

a. Single receiver. Each request should be processed by exactly one receiver or service. An example of this statement is the Command pattern.
b. Multiple receivers. Each request can be processed by zero to multiple receivers. This type of communication should be asynchronous. An example of this is the publish/subscribe mechanism used in patterns like Event-driven architecture. This is based on an event-bus interface or message broker when propagating data updates between multiple microservices through events. It is usually implemented through a service bus or similar artifact like Microsoft Azure Service Bus using topics and subscriptions.

A microservice-based application will often use a combination of these communication styles. The most common type is single-receiver communication with a synchronous protocol like HTTP/HTTPS when invoking a regular Web API HTTP service. Microservices also typically use messaging protocols for asynchronous communication between microservices. Microservices can also take advantage of HTTP/2 to improve its security and speed of the workflows; as it is multiplexed, it can leverage parallelism with one single connection.

Other protocols like Advanced Message Queing Protocol (AMQP), a protocol supported by many operating systems and cloud environments, use asynchronous messages. It just sends the message as when sending a message to a RabbitMQ queue or any other message agent.

There are several other brokers, such as ActiveMQ and RabbitMQ, that can offer a set of pub-sub messaging capabilities for simple-to-medium scale asynchronous messaging scenarios. For a fully distributed and scalable asynchronous messaging infrastructure, cloud providers offer systems such as AWS Kinesis, Azure Event Hub, and Google Pub/Sub; some Cloud providers also have managed Kafka.

3.4 Microservices Governance

When Developers move to a microservices architecture, most of the traditional governance concepts are discarded. The governance concept in microservices is interpreted as a decentralized process, which gives individual development teams the freedom to govern their own domain. Decentralized governance is applicable to service development, deployment, and execution processes, but there are other aspects to consider.

Microservices governance comprises the people, processes, and technologies that are coordinated together to implement a real-world solution. Most of these concepts are not new but ones that are already successfully used in SOA governance. They are equally applicable under the microservices architecture.

Service Definition
Any microservice that is developed should have enough information to uniquely identify itself, its functionality, and how a consumer may consume it. It should have a mechanism to specify the service definition, and it should be readily available to the service consumers.

There are several technologies such as OpenAPI, gRPC-Web, and protocol buffer, which help in defining service interfaces. These technologies allow developers to define the service identifiers, service interfaces, and service message models (e.g. service requests and responses). Other service metadata, such as service ownership, service level agreements and their accompanying SLOs (service level objectives) and SLIs (service level indicators or targets), etc., can also be part of the service definition. Service definitions are usually stored in a central repository, consumers can have access to the same and service owners can publish.

Service Lifecycle Management
Microservices have different lifecycle stages (e.g. plan, design, implement, deploy, maintain, and decommission). Given the decentralized and distributed nature of microservices, ownership of these tasks lies with the team who owns each microservice. For practical purposes, it is common to have uniform lifecycle stages for microservices irrespective of the business scope and the technologies that teams use to develop them. Service lifecycle management techniques are centrally applied to a microservices architecture. This includes the deployment lifecycle management, how to perform configuration management, version services, etc. Most of these capabilities are implemented as part of API management or control planes in a service mesh.

Quality of Service (QoS)
There are numerous Quality of Services (QoS) aspects that need to be considered prior to exposing a service to consumers. The service may be exposed as a secured service, which leverages various security protocols and standards (starting from transport layer security, access tokens, etc.). There may be a need to control access to the service using rate-limiting and throttling. Caching

and incorporating various hooks into monitoring and monetization are some other prominent QoS features. Most of these requirements are directly related to the governance of microservices and are centrally controlled.

Development Lifecycle Management
Service lifecycle management is often implemented at the service deployment level (i.e. when a given service has to be deployed across multiple environments). The deployment process addresses the requirements of replicating or moving the same service code across these environments. This also includes various DevSecOps-related deployment methodologies (blue-green, canary, AB testing, etc.) and how to manage different environments such as development (dev), test, quality assurance (QA), staging, pre-production, production (prod), and so on.

API Management
API Gateway and API management has a key role in the realization of several microservices governance aspects. As part of API management, it is important to apply security, service versioning, throttling, caching, monetization, etc. for services during runtime. It is important to understand that most of these capabilities have to be applied centrally for service invocations. Therefore, API gateways should be centrally governed or managed, and application of those capabilities can be either centralized or decentralized. Also, API gateways can be used for external or internal consumers. These capabilities can be equally applicable when microservices talk to each other via an internal API gateway. API management solutions often work hand-in-hand with service registries to discover services as well as to use them as the API repository. This is also quite useful when existing services are used and new APIs are created out of them. One other important aspect is that API management solutions provide a rich capability to discover and consume APIs. So, it is possible to leverage API management to manage all microservices.

Service Observability
When an application interacts with multiple microservices, it is vital to have metrics, tracing, logging, and visualization as well as detection and alerting capabilities for all services so that there is a clear picture of their interactions for supportability and troubleshooting purposes.

All these requirements are consolidated under one concept, called observability. With a microservices architecture, it is likely to have hundreds or thousands of services communicating with each other. The ability to get service metrics, trace messages and service interactions, get service logs, understand runtime dependencies of services, troubleshoot in the event of a failure, and set alerts for anomalies can all be considered under the umbrella of observability.

Service Registry and Discovery
Individual service definitions, the definition of service identifiers, message models, interfaces, etc., are the things that can be done without centralized governance. However, these service definitions should be published into a centralized service registry. The service registry is a centralized component, which also defines a canonical model to describe a service. All the service owners should publish their service definitions in that canonical form to the service registry. Though services are implemented with drastically different technologies (e.g. OpenAPI vs gRPC), there can still be common metadata for such services which should be added to the service registry. It is also critical from a security standpoint to have all microservices registered in a central repository for any additional runtime

metadata needed by security tools. The integrity of the service registry is of prime importance. Corruption of the service registry database could lead to redirection of service requests to wrong services, resulting in denial of services or redirection to malicious services, resulting in the compromise of the entire application.

4.0 Decomposing Monolithic Applications

4.1 Microservices: Use Cases

The following descriptions outline the use cases within microservices.

4.1.1 APPLICATION DECOMPOSITION INTO MICROSERVICES

The approach to breaking apart a monolithic application into smaller parts involves decomposition and refactoring. Decomposition is a strategic work effort to analyze the code base and organize the application into component parts. Refactoring restructures existing computer code without changing its external behavior. Decomposition targets discovery of static dependencies and minimizes their impact, while refactoring software targets improvement of software non-functional attributes. Non-functional requirement examples include resiliency, security, modifiability, or maintainability. Together decomposition and refactoring create a paradigm for re-organizing software code and favorably altering its internal characteristics.

Cases exist where a monolith is a better short-term choice. A monolith might be better suited if there is a need to get a product into the market, or close a capability gap for expediency. From a software development point of view, building a monolith can be done with a microservice refactor in mind. Although it can be interpreted as "build to throw away," transitional architectures have value for they can often uncover requirements that were not previously known until release, or recognition of certain requirements and capabilities sensitive to initial conditions. As long as the initial software architectural quality attributes exemplified by the monolithic applications NFRs (non-functional requirements) carry forward into the microservice refactoring with careful attention to preserve modularity, software construction is a matter of selecting an architectural style and not solely development execution.

NFRs are often referred to as the "-ilities." A more complete treatment can be found in Bass, Clements and Kazman's textbook titled Software Architecture 3rd Edition published by Carnegie Mellon University Software Engineering Institute. Once the software architect and DEV lead receive the NFR's key to application success, the DEV lead should carry those NFRs forward so that later code inspection reveals software constructions manifesting the NFRs. The microservice architecture style addresses problems of scale and modularity. Developers should consider using this style based on an application's current and expected scale needs. Microservices architecture style introduces new, additional architectural quality attributes (NFRs) into a monolith code base refactor such as testability and adaptability.   Developers should not hesitate to ask, "What is the simplest service refactor that could work? What can be peeled off the monolith without affecting external behavior?" Monoliths have to be worth decomposing and refactoring both for business execution reasons as well as technical reasons. Developers should start small to gain experience in both the development, delivery, and ongoing support of microservices.

Encountered or perceived challenges to application decomposition into microservices include the following viewpoints:

Developer's Viewpoint:

1. As an application becomes more fragmented, the developer has less visibility into microservices, creating a challenge to ensure that each component works flawlessly with others. The absence of proper function increases the risk of data spillage, data destruction, and availability issues.
2. A developer writing or maintaining a microservice interfacing with several other microservices may be challenged to ensure interoperability. Reliable test cases are needed. Weak test cases can increase risk of availability and performance issues later in the software development cycle.
3. Monolith decomposition results in the categorization of internal software modules into those that face-forward as part of the customer experience, those that are internal utility or intermediary functions related to inter-process communications, and those that face rearward toward data stores which are either systems of record, or systems of reference. Decomposition will also produce an accounting of interfaces, TCP/IP protocol usage, data store location, the presence of statically declared variables and credentials, and the revelation of multiple programming languages used in conjunction to accomplish particular tasks independent of any particular language used to compile or interpret it. Any one of the discoveries can be a potential source of microservice adoption, or a source of constraint preventing adoption.
4. A Microservice architectural approach is a RESTful method driven style, and not all opportunities fit easily into the approach. SOAP protocol backend connections may be using the WS-Security or WS-Reliability SOAP extensions, and any SOAP interface refactor will require REST to carry forward such security and reliability NFRs into a development style that offers only eventual data consistency. Developers will have to work with other SECOPS teams to ensure that the capabilities offered by the RESTful style can duplicate those capabilities offered by SOAP. Additionally, legacy backend data stores and other applications may not support REST, and that will require the developer to wrap the SOAP endpoint in REST which adds performance overhead in general. In cases where the monolith uses RPC (remote procedure call) and NFS (network file share), usage can be difficult to resolve. For example, the developer will have to assess the use of HTTP/TLS for file transfer to replace NFS, forcing a re-evaluation and potential replacement of the authentication and access methods used by NFS. In the other instance, RPC cannot easily use a TLS transport. The desire to use JSON-RPC likely forces a full rewrite of the module. RPC over SSL/TLS is a difficult problem to tackle. Standards have responded by removing RPC/TLS support and instead favored OS specific tunneling mechanisms to handle remote procedure calls over untrusted networks. Historically, RPC has specific vulnerabilities related to denial of service, access, and authorization.

5. In the microservice style, all developers are now "network programming" developers. Within the confines of a monolith, the bounded context is known and trusted between modules. Microservices change the application context. The development team should write code whose API calls traverse networks offering variable latency and a continuum of round-trip time performance. Developers also need to mitigate RESTful weaknesses not addressed by the style itself: API authentication, cryptographic identity, payload encryption, incorrect use of HTTPS, API Key strength, and the generalized potential of applying logic and security in the wrong place and at the wrong time. To varying degrees, the monolith handled these needs within its own internal processing boundary. As monoliths break apart, the developer will have to either code for the capability or inherit network engineering and security engineering controls that can meet the need.
6. Delivering microservice code restructures the code promotion and testing process. The primary driver of this change is a general need to shift left towards the developer both in terms of container request and delivery, as well as unit testing and end-to-end testing. Promoting software to a container can range from being as straightforward as local static security testing and code quality assessment, followed by secure FTP from a local repository over an authenticated proxy, or as complicated as a promoting code then bootstrapping (self-starting) microservices and containers in the next highest development environment using a pipeline deployment job that includes all the prerequisite secure code testing. Under a monolithic architecture, this is not an overt testing concern. A developer writing or maintaining a microservice interfacing with several other microservices may be challenged to ensure interoperability. Libraries of reliable and repeatable test cases are needed, especially those that can perform end-to-end interface testing in addition to unit testing. Developers also need to write synthetic stub-services to mimic any call out to third-party content and return data or values (known as service virtualization). The absence of robust microservice testing, synthetic data, and API documentation frameworks like SWAGGER increases the risk of code promotion that leads to unforeseen security and performance defects, up to and including inadvertent information disclosure.

Operator's Viewpoint:

1. As an application decomposes into various components, providing a proven code promotion system for use in the development process and through staging and production is an ongoing DevSecOps challenge. The absence of this can result in increased delays in code promotion.
2. Microservices have higher requirements for secure communication between components than those found in legacy applications. Operators can be challenged to provide a flexible, secure network transport between selected microservices. The absence can result in information spillage and system compromise.
3. Operations shares design responsibility to ensure that microservices do not lose state when a container is no longer running. A loss of service state leads to service disruption and data loss. Delivering a reliable and repeatable microservice product is borne out by three different constituents. Operations controls the container life cycle, Security controls the container image fidelity and integrity, and the developer consumes the container service-level and operating-level agreements. Certain non-functional requirements such as resiliency and security are delivered by both the code and the infrastructure. Microservices have a higher requirement for secure communication between components than those found

in legacy applications. Operators can be challenged to provide a flexible, secure network transport between selected microservices. The absence can result in unintended information disclosure and pathways to system compromise.

4. As an application fragments, the operations team loses visibility into microservice application behavior. Visibility challenges create production uncertainty with respect to how each component works in coordination with other microservices and supporting platforms. A monolith can be monitored uniformly and unilaterally. For every peeled or sliced-off service, microservices architectural style exacts a premium – for every one refactored service, it creates two application monitoring points, one micro-application log stream, one (or multiple) container monitoring points, and potentially one I/O data store monitoring point. The landscape can be further divisible into inner API RESTful microservices tied to back-end systems of reference or record, and outer API RESTful microservices that are part of delivering the customer experience. Modern application performance platforms require a service mesh to support telemetry over multiple microservices and containers. Service mesh requires a foundational backstop composed of container-specific security, policy-based network security, and centralization of application performance logging capable of merging with traditional security and information management platforms. Event management and machine-readable logging in an expanding microservice architecture evolves into big data analytic needs. Without a scalable event management platform, the risk of data spillage, destruction, and outright loss due to the inability to retain logging information over time increases.

Architect's Viewpoint:

1. Architects will be challenged to find a balance between the costs and benefits of rebuilding to a microservice architecture and then orchestrating those microservices. The absence of balance can result in either cost overruns, or an application that does not fully benefit from a microservice architecture. Architecture means different things to different people. In general, an experienced and diversely skilled software project manager, senior developer, or business process analyst capable of stepping back and filling the communication gap between the business stakeholder and the multiple technical teams can fulfill the "architect" role. But all architects have to remain close to the code. The architect should write code, either for educational or leisurely purposes but not necessarily write software as a vocation. Application decomposition is a software domain experience; as a result "the architect" is really functioning as a software architect working between Agile development crews, crew program managers and scrum masters, technical teams, and the business stakeholder. Not every information technology architect, whether it be a solution, infrastructure, security, data, or network architect can function in this role. Application decomposition and refactor does not always need a software architect to work as a boundary spanner between groups, but it helps to elevate the work above the next sprint horizon, maintains focus on the non-functional requirements, and keeps the developer leads and product managers focused on the work and on the big picture. Software architects that do not play a role in software testing and QA processes have no visibility into whether the code satisfies the agreed-upon architectural quality attributes (i.e. NFRs). While developers control the fulfillment of functional requirements, architects control and influence the fulfillment of non-functional requirements. If not empowered in this role, assessment of whether NFRs such as security, resiliency, modifiability, modularity, and availability falls back to the Agile team or the testing organization, neither of which may be compelled to take NFRs into account.

4.1.2 NEW APPLICATION DEVELOPMENT USING A MICROSERVICES ARCHITECTURE

Developers may need to decompose a monolith but then create a new application using microservices architecture. A microservices architecture allows multiple presentation components to integrate with business logic and data stores while allowing high-velocity changes in a continuous integration and continuous deployment (CICD) environment. The first step to decomposing a monolith is adherence to key software architecture tenets germane to microservice construction. The DEV lead or senior developer will work with the DEV team to survey the code base for business capabilities that can be isolated and peeled away from the monolith's edges. The team looks for business functions, data needs, and boundary context that can be rendered in code as a RESTful microservice endpoints while remaining an integrated and narrowly focused interface, independently deployable, loosely coupled, and language agnostic. The REST endpoint and controlling API provides a façade for executing business logic. The core microservice principles include the following:

1. Each service runs its own process, and the service delivery is one service per container;
2. Optimize as-built services for one and only one business function per service. In other words, a microservice should have one and only one reason to change.
3. Inter-service communication occurs via message brokers, distributed loosely coupled data stores, and RESTful APIs. Caution should be exercised with data stores, for stores work counter to loose coupling.
4. Microservices can be expected to evolve at differing rates. The system can evolve, but software architecture principles guide the development over time.
5. Microservices rely upon distinct per-service high availability and clustering decisions. Not all services need to scale; some will require auto-scaling, and it is unlikely that a common scaling policy will fit all microservice profiles.

Portions of a monolith can be peeled or sliced off by dividing out service capability from a business point of view. The internal technical representation should not change the external business behavior. Developers should examine the code base for delineated or well-defined interfaces without many inter-dependencies, or alternatively look for areas of the code base that are organized around a specific language or specific type of data store. When examining data stores, it must be determined if the data is tightly-coupled to the monolith code base or loosely coupled such that the monolith can function in the presence of variable data consistency and data latency. Counter-intuitively, developers should analyze the code base for bottlenecks that can benefit from a refactor as long as the refactor can deploy, scale, and fail independently. Last, an assessment must be made as to whether future feature sets are better cast as microservice enhancements rather than upgrades of the monolithic code base. Depending upon the size of the monolith, the program manager and dev lead may have to allow for co-existence between legacy monolithic applications and new microservice style application. Only so many points can fit into a sprint, and sprints into an epic. Regrettably, constraints in scope, cost, time, risk, and money all lead to circumstances where a hard cut-over to the new microservice application carries too much risk. A parallel implementation may present less business disruption potential.

Encountered or perceived challenges to new application development using a microservices architecture include the following viewpoints:

Developer's Viewpoint:

1. In a microservices architecture, each microservice needs to be autonomous. This can be a challenge for a developer accustomed to tightly-bound monolithic applications. Each microservice needs to be autonomous, which means that a fault or outage in its process domain cannot and does not affect the function of any other microservices. Developers accustomed to working within a tightly-bound monolithic application may delay the release of their feature set due to unknowns or the need to coordinate testing or capability delivery with other allied security and operations teams. Delay affects Agile crew velocity up to and including story refactor. Often times this happens when Agile crews have to promote code while also integrating or implementing with support from waterfall infrastructure and security processes that have straight-line inbox/outbox processing. The recommended practice to avoid these foreseeable delays is to have a "robust Sprint 0" added to sprint planning such that all the necessary support waterfall processes and service level agreements are known ahead of time so that expected support for risk assessment, code review, API cryptography keys, containers delivery, test environments, service accounts, and TLS certificate deployments are part of the sprint planning process. Otherwise, features will finish but block on release as they wait for a waterfall request process to complete.
2. When consuming a microservice, either a synchronous or an asynchronous protocol may be used depending on several factors of the microservice. For developers, this may provide a challenge to keep track of how specific services operate. To support scaling and availability requirements, microservices frequently run in multiples across several systems. Developers experience challenges writing services in a manner such that multiple coexisting copies can run at the same time. The absence of service coexistence and concurrence increases data loss and data destruction risk. Keeping track services consumption behavior is a common developer challenge following production promotion and software release. Prior to releasing microservice code, teams should know the end-to-end service performance profile for synchronous or asynchronous messaging and the protocols used for communication and how container lifecycle governance affects microservices. Most importantly, the institution needs to have a means to assess end-to-end application performance monitoring and a means to allow developers to build their own dashboards and access their logs to debug application problems as they surface.

Operator's Viewpoint:

1. Preferably before delivering microservices into the production environment, a robust service mesh exists to handle sidecar functions necessary for the proper discovery of microservices as well as provide other needed capabilities such as proxy, load-balancing, encryption, authorization and access, and circuit breaker support. Service mesh integrates within the computer container cluster and facilitates service-to-service communication independent of the microservice business logic. Otherwise, these IT functions have to be provided by a host of discrete appliances where managing the application run-time is much more difficult and opaque.

2. Mature network policy routing, key life cycle management, container security, and event logging backstop service mesh. Developers will need access to their application logs. Industry practice is to log off-platform so that if the entire container or cluster hosting the microservice becomes lost or unresponsive, an adjacent logging infrastructure exists on an internal private VLAN that allows developers to parse machine-generated data to understand the root cause of the failure. Although this is cast as a developer need, it is a required practice for any mature security organization. In instances of catastrophic failure, adjacent logging infrastructure functions the same as an airliner's flight data recorder and black box.

Architect's Viewpoint:

1. An architect functioning in the software architecture role has to operate at two or sometimes three different levels in the institution. Software architects have to provide direct support to the developer leads and program managers, requiring them to be close to the code. Software architects also have to function at a higher level in the institution to coordinate solution delivery containing features and functionality that are not germane to the developers' workspace, but essential for a functional platform upon which to deploy microservice. The architect in the microservice workspace has "two speeds": one geared to providing support to multiple Agile development teams and working ahead of them to sort out future challenges, prototype capability, and render in UML upcoming integrations between the code and the infrastructure, and another speed to communicate with upper management and business stakeholders through the use of traditional business-technology alignment viewpoints.
2. Ensuring a microservice architecture provides means for authentication, and authorization of access to a microservice can be a challenge due to the variety of IAM solutions and their compatibility with the hosting architecture of the microservice. As microservices each provide smaller, usually quicker services that are accessed more often, they require authentication and authorization functionalities that are also as efficient in nature. As the number of microservices scales, the amount of time just authenticating and authorizing the API caller increases.
3. Presently, few standardized methods to render architectural viewpoints that can scale down to the code, while also being able to scale up to the technical or business alignment point of view, exist. Most often, the resulting artifact mix is a combination of UML and proprietary viewpoints rendered to meet the needs and culture of the business constituency. It is common for architects to have a stable set of standard UML templates to convey sequence, class, and component diagrams. One such example is to use UML to understand how cryptography and identity and access management (IAM) interact and validate service authorization. But when an architect tries to understand the technical interaction between platforms supporting the code, UML begins to scale out in favor of network-centric representations. In a business setting, the visual representations and knowledge domain are entirely consistent with the underlying microservice interaction despite being very far from the development team. Institutions who employ architects expect architects to have different modes of operation, know when to change modes, and be successful in each mode, hence the need for two-speed architects engaged in the microservice architectural style.

4.2 Microservices: Features

The following descriptions outline the features within microservices.

4.2.1 MICROSERVICE INTEGRITY VALIDATION

A number of architectural patterns layer upon one another to provide defense in depth given that the API is not capable of providing for all the necessary control needed to ensure its integrity. API integrity is not restricted to just the cryptographic capability used to assign identity to API interfaces; integrity applies to other fundamentals in API construction to ensure that the data being passed are of high integrity, fidelity and veracity. Cryptography is joined by identity and access control and traditional network pattern adjuncts such as gateway proxy, offloading, aggregation, translation, and routing. A specific challenge exists when essential security and network policy backstops do not exist. The microservice diaspora inherits backplane control weaknesses, for it is not able to make up for the absence with just defensive software programming alone.

APIs expose the underlying implementation of the application. The REST API is standard and generic. A REST API offers predictable entry points that can be used for many different functions. In the first generation of API enabled applications, the GET/fetch of data was typically one send followed by a corresponding request-reply. Modern REST API development in the API-based application context is now many fetches occurring simultaneously, with corresponding raw data and parameters returning to the application. The client is the rendering agent and controls the majority of the client behavior and maintenance of user state while the server functions as proxy capability fronting the API. Software languages do offer specific security models whose code interacts with server-side settings (The Angular JS Framework security model is an example), but by and large the modern client handles requests and data responses by applying its own security model (document object model [DOM] and content security policy [CSP], for example).

The developer has accountability to produce secure API code, of high quality, free of anti-pattern usage, and free of defects. However, it is the engineer and operations staff that have to provide and be capable of supporting the security and policy controls that the code cannot. The developer, operator, and architect have to work together to ensure that the core software architecture non-functional attributes of microservices (autonomy, ubiquity, loose coupling, reusability, composability, fault tolerance, discoverability, and business process alignment) are neither compromised nor constrained by layered security and network policy controls.

Encountered or perceived challenges to microservice integrity validation include the following viewpoints:

<u>Developer's Viewpoint:</u>

1. Weak code leads to security vulnerabilities. It is generally less expensive to resolve coding defects early in the development process than after merge or release. Process immaturity, developer unpreparedness and unawareness are root causes of insecure software. Vulnerability disclosure post-release creates data disclosure risk at a point in time where it

is most expensive and time-consuming to patch. Unprepared developers present a unique challenge. First and foremost, the developer needs the tooling, defensive programming skill, foreknowledge or just-in-time training to set up their integrated development environment (IDE) to construct secure API-based code. To mitigate this particular challenge, key aspects of the IDE tool set include basic elements, plugins, and extensions such as

a. Specific training in defensive software programming practices in common languages like Javascript, C, C++, C# (.NET), Java, Kotlin (Android), Swift (IOS), R, Python, SQL, and noSQL.
b. Specific training on Top-10 secure coding topics such as encoding, managing sensitive transactions, managing thread and memory resources, encapsulation, escaping untrusted responses, handling exceptions, input validation, when to serialize and deserialize, managing session state, credential usage and storage, handling parameter queries, protecting data at rest and in transit, and encryption.
c. A Linter - a code-highlighting software adjunct that draws attention to formatting errors and discrepancies with industry-accepted coding standards and conventions. A linter can pinpoint logic errors as the developer creates code.
d. A Doc generator – A Doc Generator will create coding documentation as the code materializes. Doc generators can operate as a background IDE task that the developer does not have to return to later; code documentation is an invaluable output for the next developer that has to maintain the code.
e. An Open API design/documentation plugin or extension – The developer needs to use an Open API (OAS) compliant specification. API specification is absolutely necessary if the service will be advertised and consumed by other developers. For example, Swagger Hub is both an API design and documentation toolset that allows the developer to build the API specification as it is being constructed or tested. Alternatives include YAML and RAML for describing, producing, visualizing, and consuming REST APIs.
f. A static code testing plugin or extension – As part of the Shift-Left movement, the developer should have access to high quality plugins or extensions that can execute static code analysis prior to committing the final version to the remote code repository and issuing a pull request to have the code inspected by the DEV lead, or prior to entry into the continuous integration pipeline for branch merge. Many open source and commercial offerings exist.
g. A repository perspective or view – Modern IDEs offer direct integration with source code repositories as well as integration with job automation frameworks, such as Selenium and Jenkins.
h. Credential and Crypto Key management – The developer should never place static and/or clear-text credentials into promoted code. The developer can integrate SSH and TLS into the IDE so that interactions with remote code repositories can assure confidentiality. More robust key ring, key chain crypto archiving, and hardware security module integration involve platforms outside of the developer's control. Each IDE has its own ways of managing keys and it is common for a wide variety of command line interaction methods. However, the principles applied are the same even though the interactions are syntactically different from one IDE to another.
i. A Unit testing and End-to-End testing framework – Depending upon the software language, there is a wide variety of testing frameworks. Developers should use one that possesses good test-retest reliability and repeatability, and is matched for the language

used. A developer in Shift-Left will have to build, run, and report out their own unit tests. In the case of Microservices, higher level testing will require end-to-end testing because the sum of the customer experience is the combination of many distributed semi-independent parts. As a general rule, a developer should have 90-95% unit test case coverage. Anything lower does not inspire confidence and higher numbers become costly. It should be noted that coverage implies tested coverage, not code quality. Other techniques reported earlier cover code quality. Code quality and test coverage are different metrics.

   j. A Service-to-Container Bootstrap framework – Preferably this framework can initiate as a plugin, pluggable module, includable library, or extension capable of fully automating (essentially self-starting) a microservice and all its requisite support infrastructure. Bootstrapping is an essential and complicated development topic but will not be covered in this text.

2. Authentication and access control: 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. The developer needs to work with the platform owner for LDAP, Kerberos, SAML, Oath, and TLS certificate management to ensure that APIs can attest to their identity, work within their assigned authentication and authorization boundary, and validate the identity of other services in the service communications mesh. Without this control plane capability, data disclosure risk becomes unknowable and the operating environment becomes untrustable.
3. Ensuring end-to-end transaction confidentiality: When working with TLS, the CPU intensive processing occurs most during the handshake phase. The simplest way to reduce this cost is to enable persistent HTTP connections because a TLS handshake occurs only during the creation of the HTTP connection. By using this mechanism, the cost of the handshake is spread over multiple requests. Alternatively, even after reaching the persistent maximum number of requests or timeout, the TLS session ID can be used to avoid another costly handshake.
4. The developer should use the strongest cryptographic capability available. The generic profile is a key length of 2048-bits (1024 if not developing for a regulated industry), SHA512, HMAC-MD5 as a message authentication code, AES-256 or TDEA (3DES), and TLS 1.3 with a hard-stop to prevent renegotiation to lower values. Developers have three challenges: Crypto attacks always get better over time – seldom worse. Beyond the use of TLS, finer details of cryptography might be knowledge deficits, and by and large in mature security organizations the security engineering team owns the cryptographic policy, life cycle and standards, and as such, take crypto use out of the developer's hands completely. To understand how a Crypto-security team might approach cryptographic management usage see NIST SP 800 -130 *Framework for Designing Cryptographic Key Management Section 6 Cryptographic Keys* and Metadata for a complete treatment of the topic.

<u>Operator's Viewpoint:</u>

1. The operator's contribution to microservice integrity is the provision of repeatable and reliable container image repository and container lifecycle (create, store, use, share, archive, destroy) that is well-integrated into both the microservice bootstrap framework used by the developer, consistent and complementary to the microservice versioning process used by

development teams, and robustly tied to the continuous integration tool chain (commonly it is Jenkins driving the multitude of bootstrap jobs in need of coordination). Without this level of proximity to the developer environments, two vexing challenges materialize: one, the delivery of containers in the bootstrap process is highly variable in terms of image contents leading to potential runtime version or patching conflicts, and the overall management of the executing environment running inside an operating system is not fully immutable, creating containers that have to persist for one reason or another. A particular challenge of a container environment that is not fully immutable is the need to allow remote interactive login to running workloads for break-fix thereby lowering the security profile of all microservice containers; it would be a container image level change in the security profile.

2. The operator needs a delineated container delivery pipeline that leads to the provisioning step in the developer bootstrap framework. Multiple teams may contribute image layers that are then flattened into golden images. The source code and server configuration combine into the container image configuration and are archived in a version-controlled repository. Without a defined security team contributing an image layer and performing security unit testing on candidate master images, the container certification and accreditation process falls to the engineering and operations team, creating a separation of duties control break in the container deployment pipeline. Operators and engineers have many tasks to manage, including patching, new attack vectors, and traditional governance, risk, and compliance management. Performing too many roles can affect the integrity of the overall platform supporting microservices.

Architect's Viewpoint:

1. Three different architecture roles are in play. Enterprise architects offer direction in the domain of business objectives to technology alignment but offer no input into the platform or solution. A solution architect works directly with operations, security, network, and application development leads to engineer a platform capable of supporting containers, microservices, and the chosen bootstrap framework. A software architect works closest to the code and frequently joins a specialist in information security who focuses on secure coding and scripting practices; this is inclusive of developer leads. The collective goal is to produce secure microservices. A particular challenge is that many institutions do not have this level of bench strength and rely on the DevSecOps team, or DevSecOps person to fulfill all architecture functions.
2. It is improbable to find multiple skills sets in a single person and that is a challenge unique to architect work in general – it has either become highly specialized, or work gravitates to common denominators. To mitigate such weaknesses, institutions need to employ third-party consulting or canned educational packages tailored to specific needs like secure coding, defensive software programming, and environment setup. Without this support, organizations are at risk of organically growing a microservice architecture of low fidelity and integrity that cannot scale as a result of its weak control plane infrastructure, not necessarily as a result of the microservice code itself.

4.3 Monolithic Application Decomposition Best Practices

Application decomposition should not be approached according to the Mark Zuckerberg Facebook mantra "Move fast and break things"; it is bad advice for infrastructure and code. "Move fast with stable infrastructure" is more fitting.

Deciding how to partition a system into a set of services is very much an art, but there are a number of strategies that can help. One approach is to partition services by verb or use case. Another partitioning approach is to partition the system by nouns or resources. This approach finds services responsible for all operations that operate on entities/resources of a given type. A third method is to follow the input and output paths and understand the locations where data creation, editing, updating, or deleting occurs. By working backwards from the data source, the services that consume the data reveal themselves. Ideally, in the design of microservices each service should have only a small set of responsibilities, using the Single Responsibility Principle (SRP). The SRP defines a responsibility of class as a reason to change, and that a class should have only one reason to change. In Application Decomposition, developers need to refactor existing applications into component microservices. The SRP is another way to restate the primary attribute of microservices, that all constructed services provide the execution of only one business rule, logic set, or process. A complete treatment of Web Scalability is explored in Abbott and Fischer's *The Art of Scalability: The Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise* 2nd Edition," Addison-Wesley Professional, 2015.

Developer's Viewpoint:

A microservice is an API. Effective and secure API design leads to repeatable and reliable client experiences whether that client is a person or another machine. APIs produce machine-readable content that can be used for any number of repeated tasks. Microservices are stateless; what used to be a stateful interface inside a monolith's application boundary will now become a stateless RESTful API. In 2014, James Lewis and Martin Fowler described the conceptual interpretation of microservices as "Service oriented, independently deployable, independently manageable, ephemeral, and elastic." Since that time, microservices have become inseparable from API communications and container technologies. Securing a microservice happens on three planes: The API, the code itself, and the container platform. All three layers act in concert to preserve the integrity of the microservice.

Developers should employ the following API design Best Practices:

1. Understand the underlying data model, and provide new capabilities for processing data without state information. REST APIs can offer only eventual consistency and this is a programming challenge when data crosses multiple network segments and security zones.
2. Build REST APIs from templates, scaffolding, and skeleton frameworks to increase productivity. Not only does doing so standardize API construction, it provides a "tree and leaf" approach to manually populating the API specification with the remainder of the software functionality without concern about what functionality goes into making a "tree."
3. Use code generators to create client stubs for the service under construction. Client

stubs are a form of service virtualization that allows the developer to unit test the API with synthetic data and the minimum services necessary for appropriate testing coverage. Client-stub services can be archived and captured into a library for use in future testing.

4. For all APIs constructed, an API Specification document using Swagger, YAML, or RAML should be provided. Swagger is the leading specification but others exist. Tooling such as this creates defined syntax, clarity and structure.
5. APIs are the interface to the business logic, and although the business capability accessed is singularly responsible for one and only one business capability, the API can retrieve multiple streams and types of data. Recognize that APIs gather, structure, format, deliver, and secure and protect data.
6. Use open standards and industry conventions to make APIs consumable by clients.
7. APIs should be built to be interoperable with other API's. Try to avoid wrapping methods and non-standard protocols for they create processing and translation overhead. Wrapping is sometimes unavoidable. If a service must be wrapped it will also require specific performance testing and memory tuning support when under load.
8. APIs should be built to be self-explanatory and predictable. API specifications and API definitions significantly help with delivering these non-functional requirements.
9. Consumers (other machines and other developers) of APIs should have the ability to explore API functionality and be discoverable. But, if the API has public exposure, apply "need to know" principles to protect the API from being provoked into unintentional information disclosure.
10. The API should deliver just enough error messaging to the right party to facilitate solving the problem.
11. APIs need to be reusable, versionable, and capable of backward compatibility to earlier versions.
12. All API requests should be authenticated through any of the following: credentials, an assertion (SAML or Oauth Token), an implicit trust relationship (web token or key), including combinations of these capabilities. APIs do not carry their own authentication mechanism; first, it is not an open standard, and second, not delegating authentication to a trusted component or intermediary (LDAP, PKI, HSM, Key ring, or Key store) is an API anti-pattern.
13. APIs delegate the authorization to a separate service or intermediary. Interface and data access authorization are better left to coarse-grained and fine-grained access entitlements governed by a separate platform.
14. Make implicit trust part of the overall architecture when it comes to infrastructure to avoid bloating the API with security controls that can be inherited or derived from the security zones and network segmentation plane. Trust must be transparent and known. If the developer does not know where the security control comes from, they should ask. If no one can answer, they should ask again until the source of the control becomes apparent.
15. Do not rely on LOCALHOST for forming trust relationships. It may work for testing, but not for production. Rely on HTTP URI namespace and DNS resolution as a means to both locate resources and offer building blocks for further validation.
16. Application exchange of security tokens should be non-proprietary open standards such as Oath, and the microservice architecture needs a trusted third party to distribute tokens and manage token life cycle.
17. In some situations, there may be a functional requirement to provide token authentication from the initial client request as an unauthenticated consumer (like a visitor to a web page who then accesses an API). Rather than using Oath implicit grant, the preferred method is to use Oauth2 and PKCE (Proof of Key for Code Exchange).

18. If SAML2 is the only federated identity provider available, never use base64 uuencode for the payload and always encrypt the contents with AES-256. Base64 is easily reversed into clear text. Regulated industries acknowledge that email is general personal identifiable information in need of additional confidentiality. A developer may not be able to control this feature, but if in this situation, they can demand AES-256 payload encryption from the federated identity platform.
19. Use of JSON-based tokens and standard protocols is the current state of practice. JSON-based tokens are short-lived and possess constrained capabilities; they are ideally suited for limiting the threat landscape in untrusted environments and cases where the client is not trustworthy.
20. Use API keys as specific case implementation. A determined hacker can locate that key (even if held in memory and not at rest) and recover the key to become a trusted client to the API. API keys are not authoritative and should be used in a layer IAM control setting to mitigate its known weaknesses. API keys identify applications, not users of the application.
21. If the microservice goal is to use JWT (JSON-web token) for authoritative claims, use JWT in conjunction with ABAC (attribute based access control) to shift the claim from an identity base claim (i.e. "I can access everything") to an attribute one where access claims are for only the attributes set (i.e. "I can access this resource").
22. Use TLS 1.3 to ensure that API communications remain confidential.
23. The most demanding portion of the TLS negotiation is the handshake during the HTTP request. As a general rule, offload all TLS terminations to an intermediary such as an API gateway or load balancing appliance. Doing so not only affords offloading the performance requirement, but also inherits the denial of service mitigation capability that this class of purpose-built appliances offer.
24. Use the strongest cryptographic method available without compromising performance and functionality.
25. Reinforce the integrity of each microservice by using authoritative sources for code signing and certificate management. The developer ensures trust throughout the microservice environment by verifying the source code and, in some cases, the container images that are utilized. The result will be verification of trusted sources for microservices.
26. Identifying reusable code and ensuring that common libraries are populated and available to the development community. Common libraries require processes for maintenance, ownership, periodic code refactoring, and abstraction as accompaniments to a solid testing framework.
27. Developers should shift-left, unit test code, and pursue test driven security (TSD) which recommends developers write tests that represent the desired behavior first, then write the code that implements the tests. TSD is expected to fail in the first instance but this approach identifies critical security gaps in code which can be addressed in the next run.
28. All developers need to participate in reviewing each-other's code, preferably in small group peer-review during feature demonstration.
29. Development teams should publish their testing methods and results so that other developers consuming their services can build trust and understanding of the testing methods and influence how their applications will consume the microservices.

Operator's Viewpoint:

In the modern organization context, Development (Dev), Security (Sec) and Operations (Ops) enjoins to be a hybrid of all called DevSecOps. DevSecOps can be a person or it can be a team of people. DevSecOps aspires to improve efficiency, much like the merging of storage, server, and networking teams when server virtualization changed infrastructure service management. The DevSecOps construct merged the three discrete domains to foster efficiencies with shared business goals. DevSecOps inherits underlying infrastructure as code, the overlay microservice code, and bootstrap frameworks absorbed into continuous integration and continuous delivery (CI/CD) tool chains, as well as QA and testing requirements plus traditional infrastructure tasks. Dev "Operators" continually embrace additional practices in support of the microservice and containerization evolution.

Common Best Practices for DevSecOps Engineers supporting Microservices:

1. The CI/CD services used by an organization should scale as the organization moves from traditional applications and services to microservices. This is especially important as microservices should have more unit tests to ensure compatibility with each other.
2. Secure communication channels for communication should be leveraged. What used to be sensitive data being transferred within a monolithic application is now traversing the network. The operator needs to provide secure network communications when the application itself cannot encrypt/decrypt traffic. This can be done, for example, by providing SSL termination at provider load balancers.
3. Network segmentation should be leveraged to create secure communication areas. To minimize exposure, microservice architectures should be implemented with a "need to know" model. Third parties should have access only to published services, not communication between microservices and their dependencies. The API gateway pattern could be leveraged to have clients send their requests to a single-entry point, hence, shield the underlying microservices by implementing security measures.
4. Microservices and containers can be short-lived. Operators and engineers need to consider how to ensure the application's state. The result will lead to a better platform with less points of failure. By providing reliable orchestration that can scale microservices and restart them in the event of failure, the operator allows developers to gain trust in the platform running the microservices they develop and consume.
5. In conjunction with the Security organization, the operator should establish trusted signing and certificate authorities. Providing trusted signing and certificate authorities will ensure trust within the microservice environment by securing communications and provide the basis for verifiable code and container images. Operators need to provide reliable signing authorities to ensure trust within the microservice-based environment; what once was trust within application packages and code is now trust between microservices and containers on a network.

Architect's Viewpoint:

1. The Architect should focus on balanced architectures. Microservice environments require understanding of the overall architecture to ensure cohesive development with reusable elements. Building a strong application architecture design based on capabilities decomposed into microservices elements allows for the development of loosely-coupled, independent components with security embedded by design. There is always a balance between ensuring that a microservice performs one function well and the desire to eliminate duplication of effort and functionality. It is worth offloading security services to a sidecar service or a reusable independent service as a security orchestrator.

2. The architect should closely follow regulatory requirements to consider privacy requirements during the design phase. Privacy by design is an essential requirement for today's applications, considering the emergence of new privacy regulations and the erosion of a traditional network perimeter.
3. Data cannot be accessed by other microservices directly. Each microservice should have a separate database ID so that separate access can be given to put up a barrier and prevent it from using other service tables.
4. A shared database per service is not ideal. For decomposing applications, this is a good place to break the application into smaller logical pieces; this tactic should not be applied to new microservice applications. One database can be aligned with more than one microservice, but it has to be restricted to 2-3 maximum as a transitional software architecture, otherwise scaling, autonomy, and independence will be challenging to execute. Granular access control should be considered to provide "need-to-know" access by each microservice. The end architectural state is that each microservice controls its own data.
5. When moving to per service database from shared database, it is essential to consider security and privacy of data while routing, messaging, transformation and queuing data. These activities may mix up data with different classifications, leading to data leakage. The architect should use an orchestration service to deal with different data in different classification.
6. The architect should develop a data model with security and privacy in mind and focus on the segregation of classified data from unclassified data. Where possible, the use of pseudonymization, anonymization and tokenization to reduce the use of sensitive data should be explored. Employ obfuscation or view segregation at the presentation layer where required.
7. The use and storage of sensitive data should be limited. The architect should conduct an evaluation to ensure that sensitive data elements are not being unnecessarily transported or stored. Where possible, tokenization should be used to reduce data exposure risks.

Appendix A: Acronyms

Selected acronyms and abbreviations used in this paper are defined below.

ACM	Application Container and Microservices
API	Application Interface
CI - CD	Continuous Integration - Continuous Delivery
CMP	Container Management Platform
CSP	Cloud Service Provider
ESB	Enterprise Service Bus
HSM	Hardware Secure Module
IAM/ IdAM	Identity and Access Management
IPC	Interprocess Communication
KMS	Key Management System
NFR	Non-functional Requirements
OS	Operating System
QSA	Quality Service Agreement
RBAC	Role-Based Access Control
SDLC	Software Development Lifecycle
SLA	Service-Level Agreement
SOA	Service Oriented Architecture
SOTA	State Of The Art
VM	Virtual Machines

Appendix B: Glossary

Application container [NIST SP800-180]	An Application Container is a construct designed to package and run an application or its components running on a shared Operating System. Application Containers are isolated from other Application Containers and share the resources of the underlying Operating System, allowing for efficient restart, scale-up or scale-out of applications across clouds. Application Containers typically contain Microservices.
Container management platform	A Container Management Platform is an application designed to manage containers and their various operations including, but not limited to, deployment, configuration, scheduling and destruction.
Container lifecycle events	The main events in the life cycle of a container are Create container, Run docker container, Pause container, Unpause container, Start container, Stop container, Restart container, Kill container, Destroy container.
Developer [NIST SP 800-64 Rev 2/ ISO/IEC 17789:2014]	The cloud service Developer is a sub-role of cloud service partner which is responsible for designing, developing, testing and maintaining the implementation of a cloud service. This can involve composing the service implementation from existing service implementations. The cloud service Developer's cloud computing activities include: • design, create and maintain service components (clause 8.4.2.1); • compose services (clause 8.4.2.2); • test services (clause 8.4.2.3). NOTE 1 – Cloud service integrator and cloud service component Developer describe sub-roles of cloud service Developer, where the cloud service integrator deals with the composition of a service from other services, and where cloud service component Developer deals with the design, creation, testing and maintenance of individual service components. NOTE 2 – This includes service implementations and service components that involve interactions with peer cloud service providers.
Host	OS supporting the container environment
Lateral movement [LIGHTCYBER 2016]	Lateral action from a compromised internal host to strengthen the attacker foothold inside the organizational network, to control additional machines, and eventually control strategic assets.

Microservices [NIST SP800-180]	A microservice is a basic element that results from the architectural decomposition of an application's components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology. Microservices are built around capabilities as opposed to services, builds on SOA and is implemented using Agile techniques. Microservices are typically deployed inside Application Containers.
Microservices Architecture	A microservices architecture usually refers to an application that has been structured to use basic elements called Microservices, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.
Microservices Systems Software Development	This is the process of breaking down an application into components (microservices) basic elements to create through code extraction or rewrite capability (greenfield) microservice architecture of self-contained services that achieve a business objective.
Enterprise Operator	The individual or organization responsible for the set of processes to deploy and manage IT services. They ensure the smooth functioning of the infrastructure and operational environments that support application deployment to internal and external customers, including the network infrastructure, server and device management, computer operations, IT infrastructure library (ITIL) management, and help desk services.[14]
Enterprise Architect	The individual or organization responsible for strategic design recommendations. They determine, by applying their knowledge of cloud, container and microservices components to the problems of the business, the best architecture to meet the strategic needs of the business. Additionally, they develop and maintain solution roadmaps and oversee their adoption working with developers and operators to ensure an efficient and effective solution implementation.
Container resources	Three resources required for containers to operate are CPU, Memory (+swap) and Disk (space + speed) and Network.

[14] https://en.wikipedia.org/wiki/Information_technology_operations

Container resource requests	The amount of CPU, memory (+swap) and Disk (space + speed) that the system will allocate to the container its share considering the Resource Limit.
Container resource limit	The maximum amount of resources (CPU, memory (+swap) and Disk (space + speed)) that the system will allow a container to use.
Service Registry	The registry contains the locations of available instances of services. Service instances are registered with the service registry on startup and deregistered on shutdown. Client of the service and/or routers query the service registry to find the available instances of a service.
Client-Side Discovery	The client requests the network locations of available services from the service registry.
Server-Side Discovery	The Server requests the load balancer for the network locations of available services from the service registry.

Appendix C: References

[NIST SP 800-180]	NIST Special Publication (SP) 800-180 (Draft), *NIST Definition of Microservices, Application Containers and System Virtual Machines*, National Institute of Standards and Technology, Gaithersburg, Maryland, February 2016, 12pp. http://csrc.nist.gov/publications/drafts/800-180/sp800-180_draft.pdf
[NIST SP 800-160]	NIST Special Publication (SP) 800-160, *Systems Security Engineering: Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems*, National Institute of Standards and Technology, Gaithersburg, Maryland, November 2016, 257pp. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-160.pdf
[NIST SP 800-123]	NIST Special Publication (SP) 800-123, *Guide to General Server Security*, National Institute of Standards and Technology, Gaithersburg, Maryland, July 2008, 53pp. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-123.pdf nistspecialpublication800-123.pdf
[NIST SP 800-64 Rev 2]	NIST Special Publication (SP) 800-64 Rev 2, *Security Considerations in the System Development Life Cycle*, National Institute of Standards and Technology, Gaithersburg, Maryland, October 2008, 67pp. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-64r2.pdf
[NIST SP 800-190]	NIST Special Publication (SP) 800-190, *Application Container Security Guide*, National Institute of Standards and Technology, Gaithersburg, Maryland, September 2017, 63pp. https://doi.org/10.6028/NIST.SP.800-190
[NIST SP 800-204]	NIST Special Publication (SP) 800-204 *Security Strategies for Microservices-based Application Systems*, National Institute of Standards and Technology, by Ramaswamy Chandramouli Computer Security Division Information Technology Laboratory, Gaithersburg, Maryland, August2019, 33pp. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204.pdf [accessed 8/26/2019]
[Apriorit, 2018]	Apriorit. (2018). Microservice and Container Security: 10 Best Practices, by Bryk, A. https://www.apriorit.com/dev-blog/558-microservice-container-security-best-practiceshttps://www.apriorit.com/dev-blog/558-microservice-container-security-best-practices [accessed 9/10/2019]

[Addison-Wesley Professional, 2015]	Abbott, M., Fisher, M. (2015). The Art of Scalability: The Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. 2nd Edition. Addison-Wesley Professional.
[API University Press, 2016]	Biehl, M. (2016). API-University Series volume 3: RESTful API Design First Edition. API University Press.
[Aquablog, 2017]	Aqua. (2017) Managing Secrets in Docker Containers. The Challenges of Docker Secrets Management.Jerbi, A. Aquablog https://blog.aquasec.com/managing-secrets-in-docker-containers [accessed 9/4/2019] Microservices Governance : by Kasun Indrasiri, Prabath Siriwardena https://doi.org/10.1007/978-1-4842-3858-5 https://dzone.com/articles/microservices-logging-best-practices
[Beydoun et al., (2013)]	Beydoun, G., Xu, D., Sugumaran, V. (2013). Service Oriented Architectures (SOA) Adoption Challenges Service Oriented Architecture (SOA) Adoption Challenges. International Journal of Intelligent Technologies. 55.
[BMC Software, 2017]	BMC Blogs - *Microservices vs SOA: How Are They Different?* by Watts, S., 2017. https://blogs.bmc.com/microservices-vs-soa-whats-difference/?print=pdf [accessed 9/5/2019]
[Cerny et al., (2017)]	Cerny T., Donahoo J. M., Pechanec J. (2017). Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems. RACS '17 Proceedings of the International Conference on Research in Adaptive and Convergent Systems Pages 228-235 .
[Erl (2005, p. 54, p.263)]	Erl, T. (2005). Service-Oriented Architecture: Concepts, Technology, and Design: Prentice Hall
[Haddad, CIOReview]	CIOReview - *Moving from SOA to Microservices*. Haddad C. https://service-oriented-architecture.cioreview.com/cxoinsight/moving-from-soa-to-microservices-nid-6865-cid-95.html [accessed 4/9/2019]
[HashiCorp Vault]	Hashicorp Vault. Learn about secrets management and data protection with HashiCorp Vault. Operations and Development Tracks. https://www.vaultproject.io/guides/secret-mgmt/index.html [accessed 9/13/2019]

[IETF, 2018]	Internet Engineering Task Force (IETF). (2018). OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens. https://tools.ietf.org/html/draft-ietf-oauth-mtls-08#section-3 [accessed 9/10/2019]
[LeanIX, 2017]	LeanIX. Authorization and Authentication with Microservices. https://blog.leanix.net/en/authorization-authentication-with-microservices [accessed 9/10/2019]
[LIGHTCYBER 2016]	LightCyber. Cyber Weapons Report. (2016). Ramat Gan, Israel. 14pp. http://lightcyber.com/cyber-weapons-report-network-traffic-analytics-reveals-attacker-tools/ [accessed 5/11/17].
[Lund University 2015]	Lund University. (2015). Knutsson, M., Glennow, T. Challenges of Service-Oriented Architecture (SOA)-From the public sector perspective. School of Economics and Management Department of Informatics. https://pdfs.semanticscholar.org/23ef/7b2abcfaed46f37ba7330c1f4409ca8aff6a.pdf [accessed 9/4/2019]
[LWN.net, 2018]	LWN.net. (2018). Easier Container Security with Entitlements, by Beaupré, A. https://lwn.net/Articles/755238/ [accessed 9/10/2019]
[Medium, 2018]	Medium Corporation. (2018). Tech Tajawal - *Microservices Authentication and Authorization Solutions*, by Ajoub, M. https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a [accessed 9/5/2019]
[Newman, 2015]	Newman S. (2015). Building Microservices. Designing Fine-Grained Systems. O'Reilly Media.
[Nordic APIS, 2017]	NORDIC APIS. (2017). API Security: The 4 Defenses of The API Stronghold. Sandoval, K. https://nordicapis.com/api-security-the-4-defenses-of-the-api-stronghold/ [last accessed 11/11/2019]
[O'Reilly Safari Publications, 2018]	McLarty, M., Wilson, R., and MorissIon, S. (2018). Securing Microservice API's: Sustainable and Scalable Access Control. First Edition. O'Reilly Safari Publications.
[Project Calico]	Project Calico. Policy-driven Network Security. Tigera Inc. https://www.projectcalico.org [accessed 9/10/2019]
[Red Hat OpenShift]	Red Hat OpenShift - OpenShift Container Platform - Authorization. https://docs.openshift.com/container-platform/3.6/architecture/additional_concepts/authorization.html#rules-def [accessed 9/10/2019]

[Rosen at al., (2012)]	Rosen, M., Lublinsky, B., Smith, K. T., & Balcer, M. J. (2012). Applied SOA: Service-oriented Architecture and Design Strategies: John Wiley & Sons
[Simplicable 2011]	Simplicable Technology Guide. (2011). SOA Security Challenges by Spacey, J. https://arch.simplicable.com/arch/new/9-soa-security-challenges [accessed 9/4/2019]
[SPIRE]	SPIRE. The SPIFFE Runtime Environment. https://spiffe.io/spire/ [accessed 9/10/2019]
[Stojanovic et al., (2004)]	Stojanovic, Z., Dahanayake, A., Sol, H.G. (2004) Modeling and Design of Service-oriented Architecture. Conference: Proceedings of the IEEE International Conference on Systems, Man & Cybernetics: The Hague, Netherlands.
Tanenbaum at al., (2007)]	Tanenbaum, A. and Van Steen,M. (2007). Distributed Systems: Principles and Paradigms. Pearson Prentice Hall.
[TechTarget, 2018]	TechTarget. (2018). How Proper Networking Supports Microservices Security by Nolle, T., CIMI Corporation. https://searchapparchitecture.techtarget.com/tip/How-proper-networking-supports-microservices-security [accessed 9/10/2018]
[The SOA Source Book]	Microservices Architecture – SOA and MSA. https://www.opengroup.org/soa/source-book/msawp/p3.htm [accessed 8/30/19] Governance Framework – SOA Governance Reference Model (SGRM) https://www.opengroup.org/soa/source-book/gov/p4.htm [accessed 9/4/2019]
[Unbound, 2018]	Unbound Tech. (2018). Introduction to Cloud-Native Secrets Management: Part III. https://www.unboundtech.com/importance-of-container-identity/ [accessed 9/10/2019]
[Weaveworks]	Weaveworks. Microservices Security: Built-In Best Practices. https://www.weave.works/use-cases/security/ [accessed 9/13/2019]
[Wei et al., (2018)]	Wei, F., Ramkumar, M., Mohanty, S.D. (2018). A Scalable, Trustworthy Infrastructure for Collaborative Container Repositories. *ArXiv, abs/1810.07315*.

[Zimmerman, 2016]	Zimmerman, O. (2016). Microservices tenets: an Agile Approach to Service Development and Deployment. Computer Science – Research and Development, pp. 1-10.
[Yarygina et al., 2018]	Yarygina, T. and Bagge, A.H. (2018). Overcoming Security Challenges in Microservice Architecture. Proceedings of 2018 IEEE Symposium on Service-Oriented System Engineering.