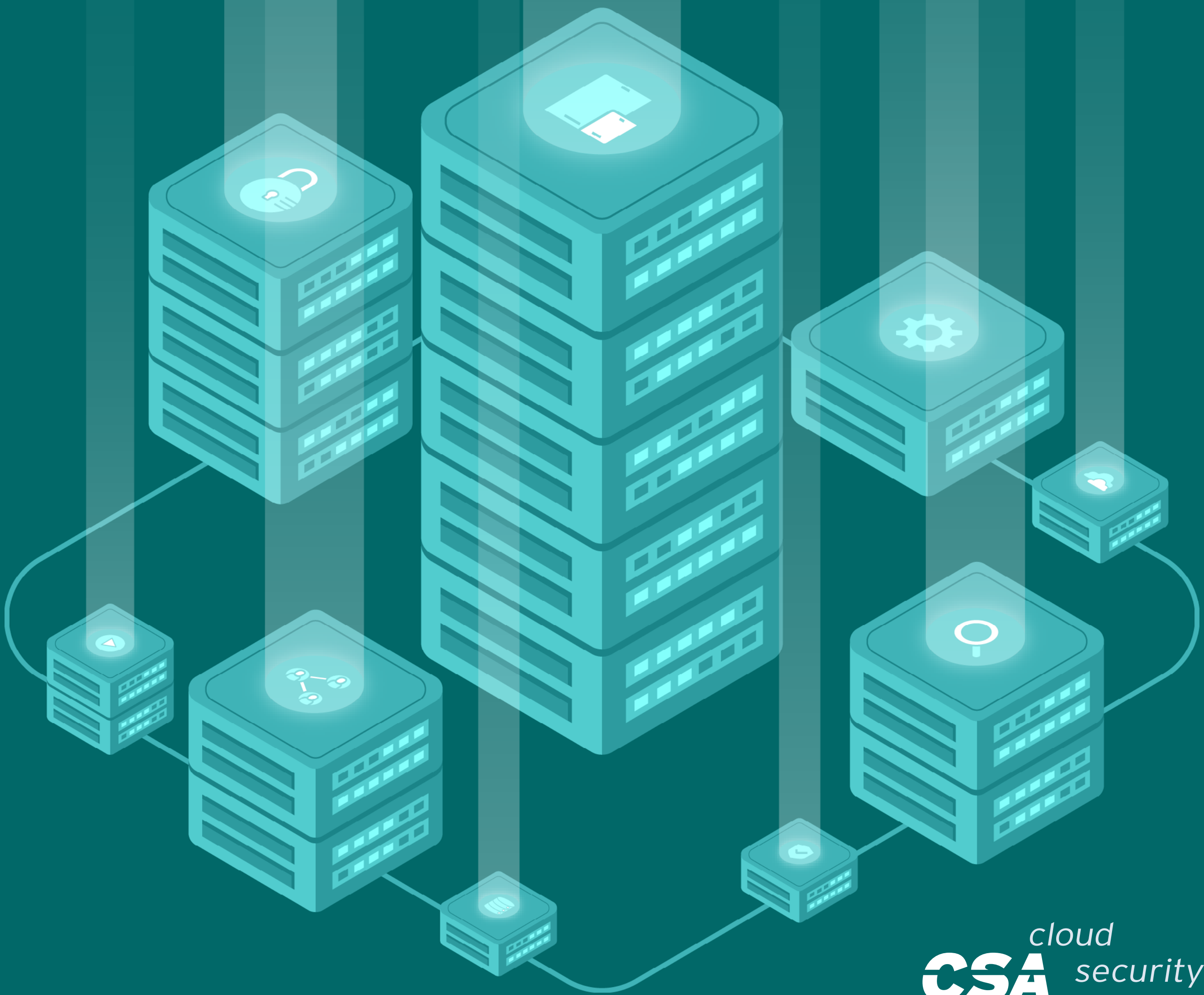
How to Design a Secure Serverless Architecture



cloud security alliance®

Acknowledgments

Team Leaders/ Authors:

Aradhna Chetal
Vishwas Manral
Marina Bregkou
Ricardo Ferreira
David Hadas
Vani Murthy
Elisabeth Vasquez
John Wrobel

Key Contributors:

Amit Bendor
Peter Campbell
Madhav Chablani
John Kinsella
Namrata Kulkarni
Akshay Mahajan
Eric Matlock
Shobhit Mehta
Vrettos Moulos
Raja Rajenderan
Abhishek Vyas
Brad Woodward

CSA Global Staff:

Marina Bregkou
Claire Lehnert (Design)
Stephen Smith (Design)
AnnMarie Ulskey (Cover, Design)

Reviewers:

Anil Karmel
Alex Rebo

Table of Contents

Executive Summary

Serverless platforms enable developers to develop and deploy faster, allowing an easy way to move to Cloud-native services without managing infrastructures like container clusters or virtual machines. As businesses work to bring technology value to market faster, serverless platforms are gaining adoption with developers.

Like any solution, Serverless brings with it a variety of cyber risks. This paper covers security for serverless applications, focusing on best practices and recommendations. It offers an extensive overview of the different threats focusing more on the Application Owner risks that Serverless platforms are exposed to and suggest the appropriate security controls.

From a deployment perspective, organizations adopting serverless architectures can focus on core product functionality without being bothered by managing and controlling the platform or the compute resources with their respective load balancing, monitoring, availability, redundancy, and security aspects. Serverless solutions are inherently scalable and offer an abundance of optimized compute resources for the “Pay as you go” paradigm.

Further, from a software development perspective, organizations adopting serverless architectures are offered deployment models under which the organization is no longer required to manage and control the underlying operating system, application server, or software runtime environment. As a result, such organizations can deploy services with less time to market and lower their overall operational costs.

This paper›s recommendations and best practices were developed through extensive collaboration among a diverse group with extensive knowledge and practical experience in information security, cloud operations, application containers, and microservices. The information is intended for a wide variety of audiences who may have some responsibility in Serverless environments.

1. Introduction

Purpose and Scope

The purpose of this document is to present best practices and recommendations for implementing a secure serverless solution.

Two players are involved in a Serverless service:

- The Service/Platform Provider - the provider of the serverless platform on which serverless applications are built
- The Application Owner - the user of the serverless solution whose applications run on the platform

Under a serverless solution, a service provider offers compute resources that are elastically auto-allocated to serve the needs of different executables without the service users controlling the resources required to serve each executable. The scope of this document is limited to users from one administration implementing their workloads on top of Serverless solutions offered by other administrations acting as platform providers. Implementations of Serverless include both the Container Image-based Serverless, also called Container-as-a-Service (Serverless Containers), and Function-based Serverless, also called Function-as-a-Service (FaaS).

In this document, we focus more on the Application Owner and consider the threats to the Application while using a Serverless service. We then make specific recommendations for the security best practices and list the recommended controls to be adopted by Application Owners.

As a lot of the details of securing different cloud services are covered in other documents, this document will focus only on the aspects that change due to moving to Serverless services and avoid detailing with more generic cloud-related security aspects.

The primary goals of this paper are to present and promote serverless as a secure cloud-computing execution model. The aim is also to help guide Application Owners looking to adopt the serverless architecture.

Audience

The intended audience of this document is application developers, application architects, Security Professionals, Chief Information Security Officers (CISOs), risk management professionals, system and security administrators, security program managers, information system security officers, and anyone else interested in the security of serverless computing.

The document assumes that the readers have some knowledge of coding practices, security and networking expertise, and application containers, microservices, functions, and agile application development. Owing to the constantly changing nature of technologies in the serverless space, readers are encouraged to take advantage of the other resources, including those listed in this document, for current and more detailed information.

The audience is encouraged to follow industry-standard practices related to secure software design, development,and deployment.

2. What is Serverless

a. Definition of Serverless

Serverless computing is a cloud-computing execution model in which the cloud provider is responsible for allocating compute and infrastructure resources needed to serve Application Owners› workloads. An Application Owner is no longer required to determine and control how many compute resources (and at what size) are allocated to serve their workload at any given time. It can rely on an abundance of compute resources that will be available to serve the workload on-demand. Therefore, serverless computing is offered under a **"Pay as you go"** paradigm where payment is generally made on actual physical resources like central processing unit (CPU) usage time.

Application Owners using Serverless provide the Service Provider with a **"Callable Unit"** that needs to be executed (called, triggered) and a set of **"Events"** under which the Callable Unit needs to be executed (called, triggered). Application Owners can also provide supporting code to run along with the callable unit. The supporting code can be provided as libraries called "layers" that can run in the context of the running function (as part of the same process), as well as "extensions" or supporting threads/ processes that can run in the same environment (different process from the callable unit). The "layers" and "extensions" share the same resources allocated to the function and run in the same context.

Note that the name "serverless" applies only to the behavior experienced by the Application Owner who is using the service. Under the hood, some «servers» still exist that execute the code but are abstracted away from the Application Owner.

b. The Callable Unit

Different Serverless solutions offer a range of options for providing the Callable Unit. One common service option enables Application Owners to provide function code under one of the runtimes supported by the Service Provider (JavaScript, Python, Java, etc.). Such services are known in the industry as "Function as a Service" or FaaS in short. Under FaaS, the function code provided by the Application Owner is typically embedded in a container image owned and provided by the Service Provider. An extension of FaaS may include extending the Service Provider's image by installing additional libraries and injecting data to the container before the image is spun on and the function is executed.

A second common option enables Application Owners to provide container images of their control to serve as the Callable Unit. This extension of **FaaS** is well distinguished from non-serverless services in which container images are provided by Application Owners to be executed on top of managed or unmanaged container services. The following table highlights this distinction. Under Image-based Serverless, the service provider is responsible for implementing the correct number of instances of the Callable Unit in response to Events at any given time.

	Serverless (discussed in this document)		**Microservices** Non-Serverless (not discussed here)	
Name	Function based Serverless	Container Image based Serverless	Managed Container Services	Kubernetes Services
Callable Unit delivered by Application Owner	A Function w/wo dependencies	A Container Image		
Dependency	Programming language-specific	Can run applications independent of the code's language, as all binary and dependencies are packaged.		
Control over scaling, load balancing, redundancy, instance monitoring of the executables	Service Provider		Application Owner	
Control over availability, redundancy of instances	Service provider has control over the availability and redundancy of instances		Application Owner	
Life cycle and scaling of the underlying servers	Service provider has control over the availability and redundancy of instances		Application Owner	
Execution Time	Typically short (seconds or less). Generally limited to a few minutes.		Typically long-lasting and unlimited.	
State	Stateless and ephemeral - all states primarily maintained outside of the Callable Unit		Mostly stateless by common practices for microservices but can maintain state in mounted volumes	
Scaling compute	Service Provider responsibility		Application Owner responsibility	
Payment model	Pay as you go		Pay for resources Allocated	
Runtime responsibility	Service Provider	Application Owner		
Examples	AWS Lambda Azure Function Google Cloud Functions IBM Cloud Functions	AWS Fargate Google Cloud Run IBM Code Engine	AWS ECS Red Hat Openshift (on AWS/Azure/..)	AWS EKS Google GKE Azure AKS IBM IKS

Under **Function-based Serverless (Also known as "Serverless Functions")**, the image is owned and controlled by the Service Provider, which leaves the responsibility for the security of the Image to the Service Provider. As a result, the Service Provider must have the proper controls to mitigate any risks to the Application Owner workload from the image (see "Service Provider conduct's Threats" in section 5.3).

Under **Container Image-based Serverless (Also known as "Serverless Containers")**, the image is owned and controlled by the Application Owner, which leaves the responsibility for the security of the image to the Application Owner. As a result, the Application Owner is the one that will be required to have the proper controls to mitigate the risks from the image to his workload (see "Application Owner Setup Phase Threats" in section 5.3).

c. The Events

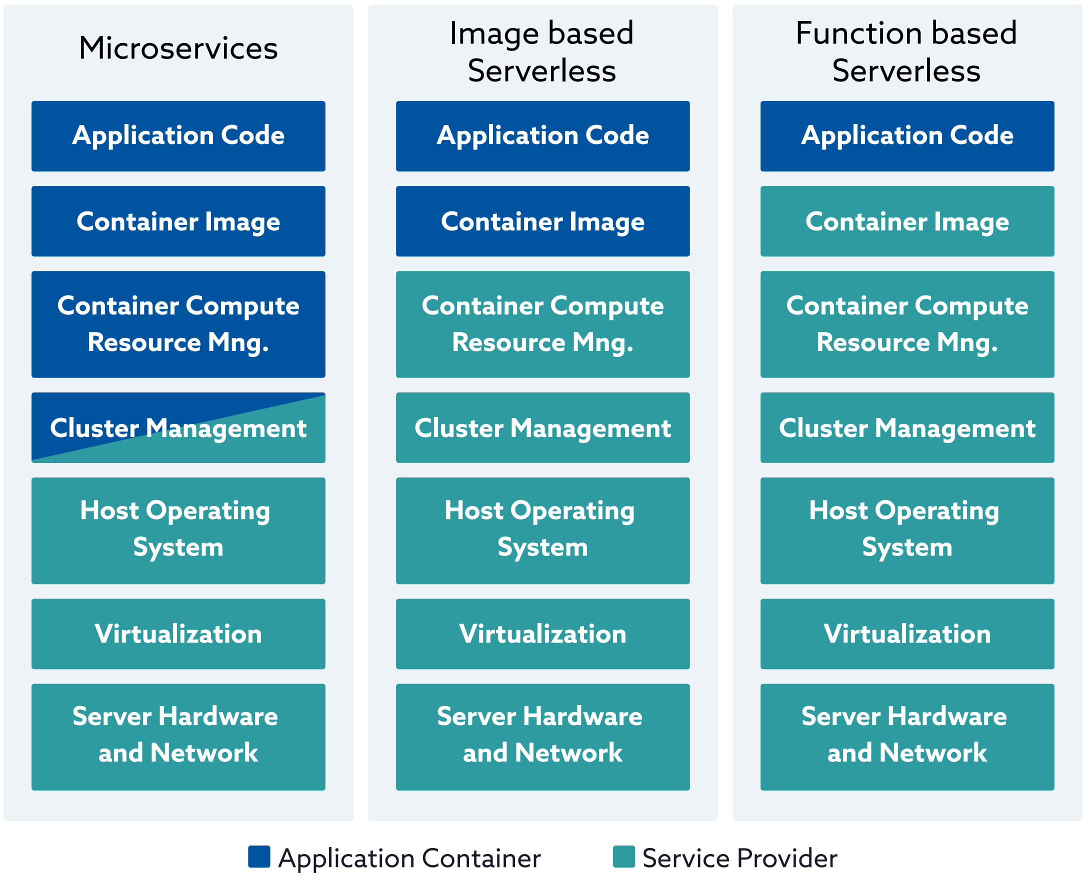
Events are conditions in the serverless environment that may cause a particular function to be triggered - it could include new additional data, a new packet received, or just the expiration of different periods. Apart from the Callable Unit, the Application Owner for event-driven applications can provide the Service Provider with a set of Events and thus defines when an instance of the Callable Unit needs to be executed to process the event. The Service Provider takes responsibility for queuing events, initiating sufficient models of the Callable Unit, and handling each event to be processed by a Callable Unit within some limited time based on the Service Level Agreement offered by the Service Provider. The actual processing time of the events is accumulated for billing purposes.

Events may vary based on the source of the Events and the type of each Event. Examples of events include a timer event, an event triggered by a web request, an event triggered by changes in the data stored on storage or in a database, an event triggered by some interaction between services occurring in the cloud account or between a user and service in the cloud account, event from an event service, a monitoring service or a logging service in the cloud account, etc.

d. Serverless Security Overview

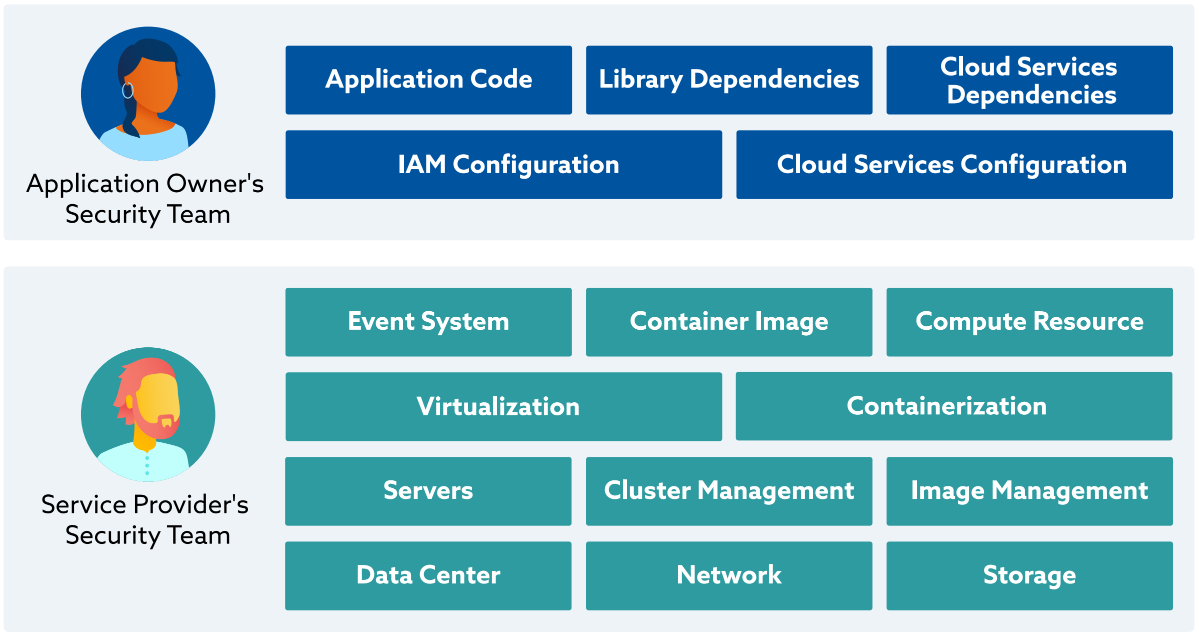
Serverless security brings in a new paradigm of security where the Application Owner is only responsible for the protection of the application in addition to securing the data. All aspects of managing the server or its security, including bringing up, patching the machine operating system (OS), updating, and bringing down, are managed by the Serverless platform provider, thus enabling the Application Owner to focus on the application itself instead of also focusing on the infrastructure. However, as will be detailed later in this document when discussing the threats Application Owners need to consider, Serverless introduces its own Security challenges.

To better explain the shared responsibility between the Platform Provider and Application Owner for the different models, we created the diagram below.



Comparative Shared responsibility model

Detailed diagram of Serverless responsibility model:



Function based Serverless (FaaS) shared responsibility model

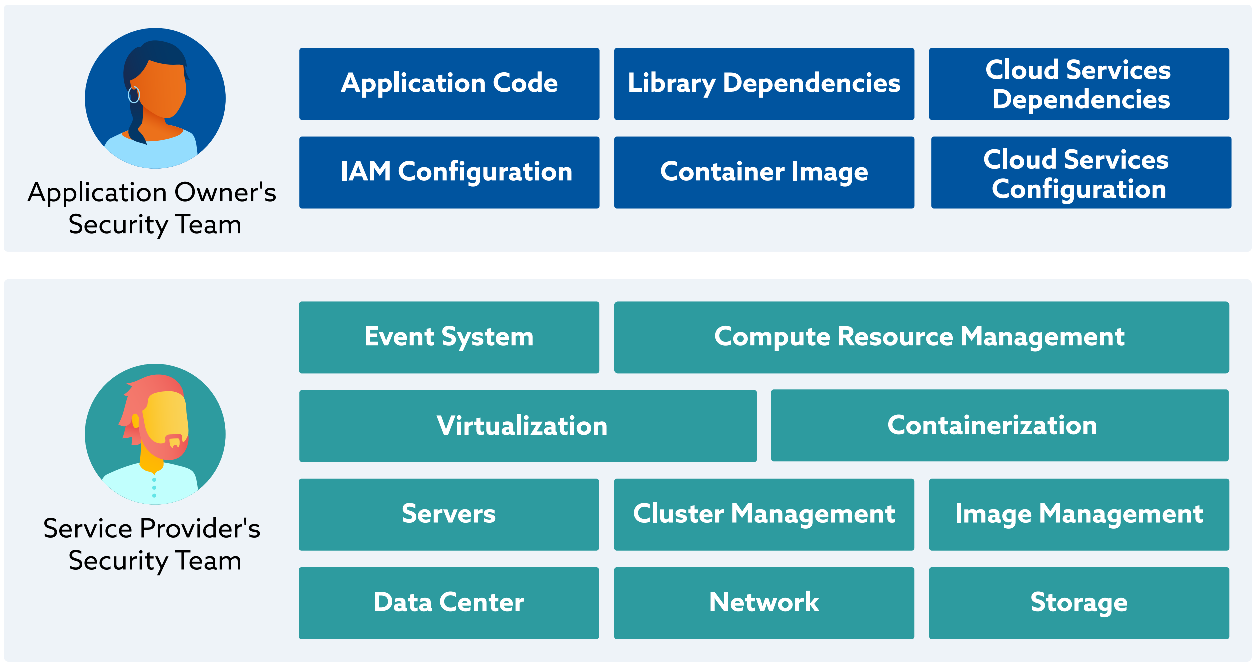


Image based Serverless shared responsibility model

As Function-based Serverless can be specific to an operating/ runtime environment, the platform provider takes up the responsibility of maintaining and updating different versions and programming languages.

e. Hybrid serverless architecture (private & public)

There are many serverless architectures. Some common infrastructures examples are (not comprehensive):

- Amazon: Lambda, Fargate, AWS Batch
- Google: Cloud Functions, Knative, Cloud Run
- Azure: Azure Functions, Azure Container Instances
- Nimbella: OpenWhisk
- IBM: Knative (Code Engine), OpenWhisk (Cloud Functions)
- RedHat: Knative (part of OpenShift)

3. Why Serverless

Serverless computing offers several advantages over traditional cloud-based or server-centric infrastructure.

In **Serverless Computing**, Application Owners generally do not need to be concerned with the infrastructure hosting their application, including the maintenance and patching of the operating system the application runs on or scaling out the infrastructure. This helps significantly reduce Operations overhead. [Manral, 2021]

Additionally, application code is hosted and run on a dynamic platform: A particular code may run on one of many physical machines. However, Application Owners have little to no visibility into where their code is physically resident.

Serverless platforms usually have dynamic scaling capabilities.

3.1 Advantages and Benefits of Serverless Architecture

The areas of benefit of serverless architecture are described in the table below for the readers' facilitation.

Serverless offers advangates for the:	**Serverless** (discussed in this document)
Speed of deployment	Serverless enables Application Owners to develop business applications without being concerned about the application's infrastructure, enabling the business to build and deploy applications at a fast pace. It is, therefore, a good tool for experimentation and bringing new value to the marketplace faster.
Cost	Infrastructure Cost: • Priced (usually) on a per-event basis, which means you do not need to pay when you're not using the infrastructure • Very cost-effective on burst workloads - you do not have to maintain servers when they are not required and can provide very fine granularity and control of the resources used. Operational Cost: • Not having an infrastructure to manage can cut operating costs and time spent on maintaining it. [Manral, 2021]

Application Owner experience	Easy to deploy: • Serverless services can be easily deployed with minimal configuration with CLI tools, from source control or a simple Application Programming Interface (API). Easy to monitor: • Most cloud providers offer out-of-the-box logging and monitoring solutions bundled with their serverless offerings. The platform is API-driven, which is critical for the app owner's productivity. No server management overhead: • Serverless services abstract all server management tasks such as patching, provisioning, capacity management, operation system maintenance.
Scale	Scalable by nature: • Serverless auto-scales on fine granularity based on usage by just configuring infrastructure without having to set up the infrastructure • It is not necessary to configure policies for scaling up or down the workload. • When working on-premise, scaling is limited to the available infrastructure.

3.2 Shared Responsibility Model for Serverless

In a **shared responsibility model**, the developer is responsible for securing their code and the tools used to deliver applications to the cloud. In this shared responsibility model, each party maintains complete control over those assets, processes, and functions they own.

Service	Application Owner	Serverless Platform Provider
Platform patching		Image and Function-based Serverless.
Platform configuration	Application and platform configuration related to the application.	Exposing minimal configuration to the Application Owner.
Image patching	Container Image-based Serverless.	Function-based Serverless.
Secure coding practices	Image and Function-based Serverless.	

Supply Chain security	Application and components-based supply chain.	Platform supply chain.
Network security monitoring		Image and Function-based Serverless.
Application security monitoring	Image and Function-based Serverless.	
CI/CD Pipeline Configuration	Image and Function-based Serverless.	

3.3 When is Serverless Appropriate

The serverless model is most appropriate when there is a relatively large application or set of applications and a mature software development and operations (DevOps) team, process, and products available to support them.

In such a case, the application(s) can be broken down into smaller components called Microservices (see Best Practices in Implementing a Secure Microservices Architecture). Each is supported by one or more teams and runs in a serverless environment. This allows for more effective use of development resources by focusing on a specific piece of functionality. This model also allows for more agile development of each microservice when compared to a monolithic application because functionality for each part of the application can be moved into production without as much concern for full integration and regression testing with the other application features.

With relatively small applications or teams, a serverless model can sometimes be less cost-effective than having a traditional infrastructure to support the application (such as Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) services). There is typically less complexity with a smaller application, and the benefits of breaking the application down into microservices are lost. In such a case, microservices can be so tightly coupled with other services that some benefits of microservices, such as reusability, are lost. Insufficient resources to support many microservices may cause teams to stop working on one microservice in order to support another.

It is also important to note that serverless architectures will simplify the deployment process in almost all cases. deployment consists of simply uploading a container image or set of code without as much concern for resource provisioning and network architecture as with traditional application deployment. Organizations need to perform a business impact analysis and cost/benefit analysis when deciding on using serverless architectures to choose the most technically efficient, cost-effective, and appropriate solution for their business needs.

4. Use Cases and Examples

The key role of serverless computing is to give cloud programmers the necessary tools that can diminish the complexity of cloud-oriented architectures by removing the need to consider infrastructures and enabling direct use of programming languages. However, many concerns need to be addressed in advance, including load balancing, request routing to efficiently utilize resources, system upgrades such as security patching, migration to newly available instances, and geographic distribution of redundant copies to preserve the service in case of disaster.

One use-case for the embrace of serverless architectures is bringing immense scale without broad technical expertise in operating and scaling the infrastructure. Building, deploying, managing, and scaling a serverless application is possible without maintaining hardware, operating systems, or supporting software. This enables the Application Owners to focus on business logic instead of non-essential capabilities. The Application Owner's cost is highly granular, allowing prices to scale linearly with use, resulting in consistent economic viability. The total cost may, however, depend on the scale of an operation. A customer/user will have to decide the platform based on their needs.

Some business use cases for serverless are: (Note: this is not intended to be a comprehensive repository of use cases but just providing some examples for providing context for security professionals)

1. Web Applications: where a user needs to access an existing service and view or make minor updates. After that activity is complete, the function may be deleted—basically, traditional request and response workloads. Serverless functions can be used, but security threats such as improper authentication and non-repudiation will still need to be addressed.
2. Data processing activities are event-driven, and after the data processing request is complete the service may be deleted. For example, a trigger is initiated to pull a report of all account debits or stocks purchased in a day for a customer through a brokerage account.
3. Data integrity, confidentiality, and security issues still need to be addressed as part of the serverless functions in addition to authentication and authorization.
4. Batch processing use-cases where a series of triggers can be set up and a workflow built to extract, manipulate and process data. For example, pull a list of automobiles who failed the carbon emission test and send the owners an email with state requirements and standards and a deadline to meet those. Security concerns will be low privilege access, the confidentiality of data, and the users' privacy.
5. Event ingestion and integration: gathering all events from an analytics application and feeding them to Database for indexing and filing the events and with specific triggers initiate a reporting function or publishing to a dashboard/web interface. Access management and non-repudiation concerns have to be addressed in such cases from a security perspective and ensuring there are logs generated with the metadata needed for detection.
6. Serverless is already used for security detection and auto-response in the industry, heavily for alerting misconfigurations and taking subsequent actions.
7. Serverless can be used for image recognition and processing. Examples include Google Vision and Amazon Rekognition services, then indexing those pictures based on the identification.
8. Alternatively, serverless can be used with apps that allow customers to upload their credit card information and then extract attributes to process the transaction.

9. Data security, privacy,identity, and access management concerns must be addressed for such use cases.
10. There are use cases where triggers can be generated to pipe events to SaaS providers to process.
11. CI-CD pipelines need the capability to iterate through software development quickly. Serverless can automate many of these processes. Code checkings can trigger builds and automatic redeploys, or change requests can trigger running automated tests to ensure code is well-tested before human review. Anywhere where automation is needed, there is potential to simplify or accelerate it with Serverless Applications and make it easy to eliminate manual tasks from the workflow.
12. In an industrial environment where IoT sensors input messages, there is a need to process those messages based on the triggers that can be set then use serverless functions to respond to messages and scale in response. The impact of insecure functions can be severe in some cases. Hence authentication, data protection, and detection are vital controls which need to be addressed in these scenarios in addition to failures.
13. If there is an application that requires specific steps to be taken based on business logic: the orchestration of microservice workloads that execute a series of steps can be implemented using serverless functions. There are several security concerns in using serverless from an orchestration perspective, data/metadata passed by the triggering event from an auditability perspective, failures/availability, and non-repudiation issues in addition to authentication and authorization.
14. Customer service use-cases during holiday season respond to customer queries viz. Chat-bots, serverless, provide capabilities to automatically scale for peak demands and remove functions after the chat is over.
15. User authentication and data protection/privacy are still concerns from a security perspective that need to be addressed.
16. Other prevalent use cases for serverless in the industry are infrastructure automation tasks such as backups at scheduled times etc.

A significant issue in enterprises from a security team's perspective is that they may not know where all Serverless functions are used in their platforms --Considering some of these functions can be short-lived, it makes it difficult. The following chapters provide detailed recommendations for building controls that will help with visibility and asset tracking for serverless in an Enterprise.

As discussed above, security teams can use serverless functions to trigger and use serverless functions in automating security tasks. Detection and response use cases are already prevalent in the industry, but other tasks where Security teams can use serverless are automated scans and enforcing breaking builds if they have vulnerabilities in CI-CD pipelines.

Serverless functions can trigger on-demand scans of the infrastructure and report the results or automated enforcement thereof.

Serverless functions can also be used for enforcing other security controls. For example, any time a user uploads a data element, a function will check if the data has been tagged or labeled. If not, it will generate an alert or email to the user that the uploaded file is not tagged. It will then be isolated until labeled to ensure appropriate DLP or other compliance policies can be enforced.

Serverless functions can be used for enforcing security policies viz. E.g., while validating the access to a service, a function checks that the assurance of authentication can be implemented. A step up authentication may be required, so a failed level of assurance of authentication can be a trigger for another function to request a step up authentication. And similarly, there could be other functions used in enforcing security policies.

Similarly, use of serverless in security is being explored, and there is much potential to simplify and automate security tasks using serverless. At the same time, it is pertinent that security teams and developers understand the threat landscape and security controls critical for serverless technologies, which are discussed at length in the following chapters.

5. Security Threat Model of Serverless

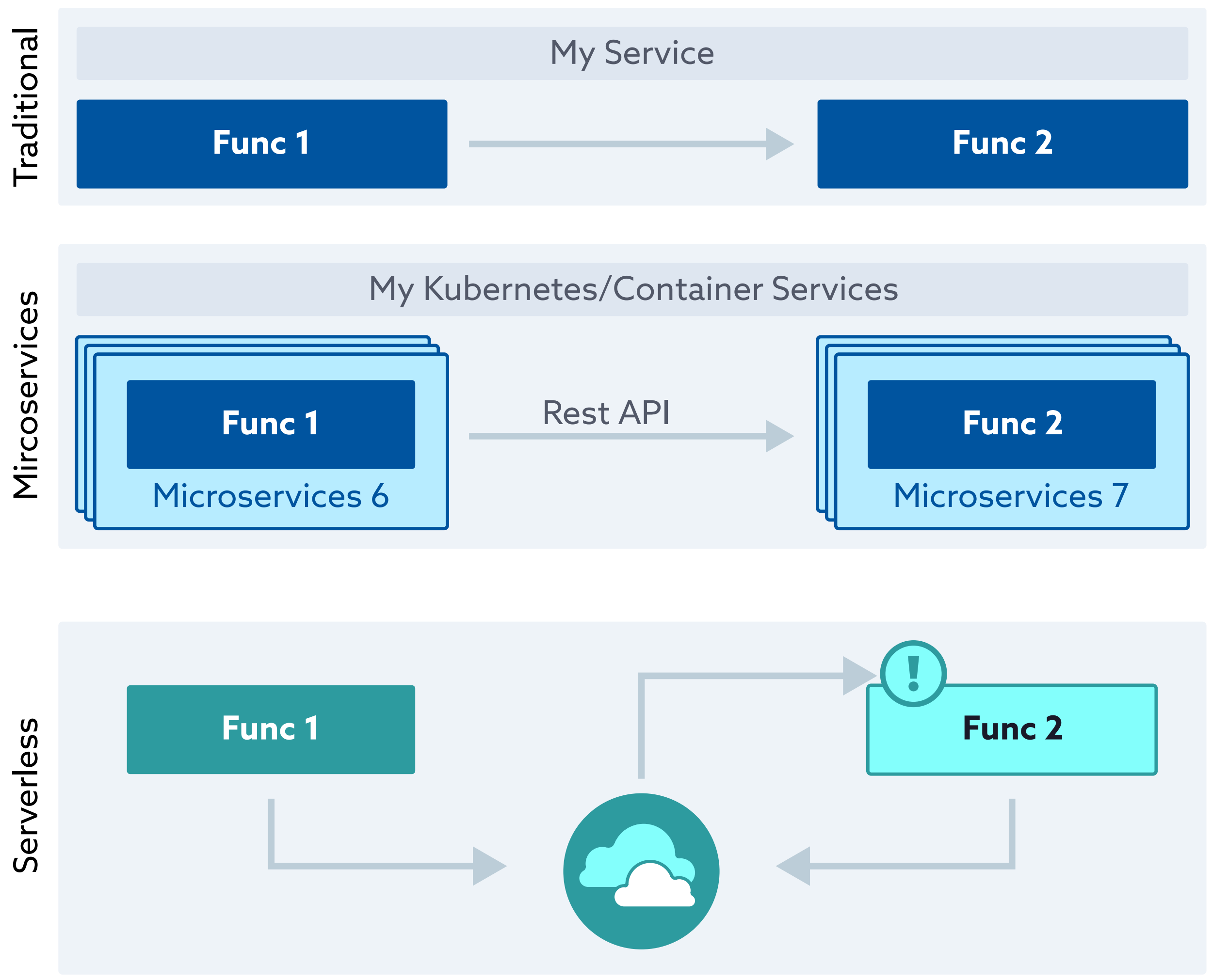
While using serverless technologies, there are many threats application developers have to consider. This section discusses how and why the use of serverless services changes the threat landscape for the application services. A detailed threat model with unique threats and how they manifest in Serverless technologies and discuss special security attention and tools for Serverless. Serverless mitigations, architecture design, and mitigating security controls will follow in Chapter 6.

5.1 Serverless - A whole new Security ballgame?

Cloud service providers' introduction of serverless services brings new security challenges to application/service developers and owners. Serverless, as an event-driven architecture, breaks workloads into a multiplicity of seamlessly isolated execution environments. Each carries a specific task and handles an individual event, running separately in time and space, with its own dependencies, code, image, privilege requirements, configuration, and lifespan. This is a significant break from the traditional security threat model and the cloud threat model in the non-serverless microservices environments. Application Owners utilizing Serverless are required to re-evaluate the evolved threat landscape and reconsider the appropriate security controls needed.

Serverless may include flows crossing trust boundaries, such as getting data from a public location, inputting data into customer premises, calling functions in other locations, etc. The existing network boundaries fade away. This fragmentation may lead to a need to drastically increase the volume of the security raw data to be collected, processed, and analyzed to detect attacks.

Serverless is yet another step of immersing the workload deep into the cloud while moving even more functionality that used to be owned and controlled by the workload developers to the Cloud Service Provider (CSP). For example, what was previously implemented as a function-call in traditional code and potentially became a Rest API in microservices, now moves to be an event submitted, queued, and handled by the CSP. The CSP will see this through until the actual data handling is complete with a function -- sometime after the requester asked to have this data handed. Much control is released by the Application Owner and handed to the Service Provider -- a process that again reframes what can and should be done by service consumers to ensure the security of the serverless microservices/applications and by whom.



Serverless - A whole new Security ballgame

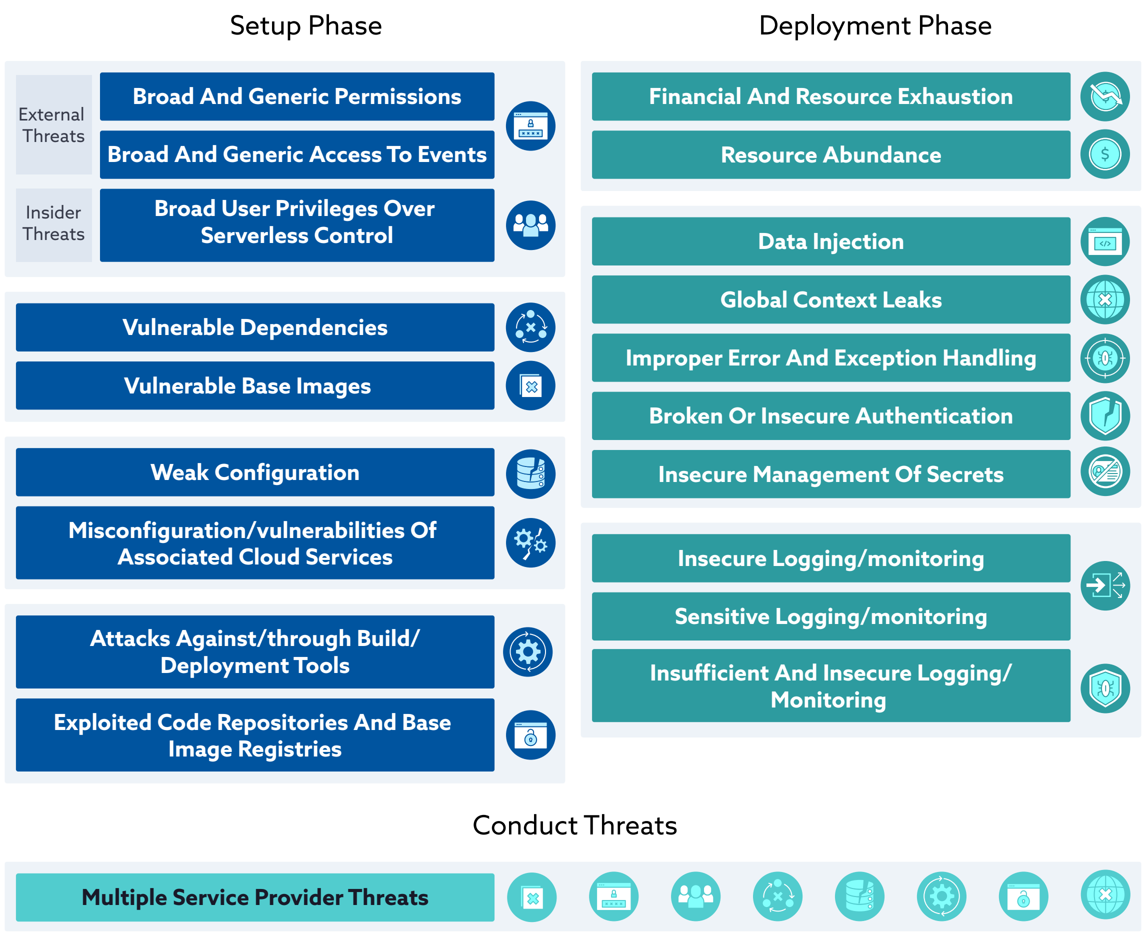
5.2 Serverless Threat Landscape

5.2.1 Key threat areas

Threats to the Application Owner workload when using Serverless can be divided between

> 1. Application Owner Setup Phase Threats
> 2. Application Owner Deployment Phase Threats
> 3. Service Provider's conduct Threats

In the below diagram, we summarize key threat areas to Application Owners under Serverless.


Setup Phase
Deployment Phase
External Threats
Broad And Generic Permissions
Broad And Generic Access To Events
Insider Threats
Broad User Privileges Over Serverless Control
Vulnerable Dependencies
Vulnerable Base Images
Weak Configuration
Misconfiguration/vulnerabilities Of Associated Cloud Services
Attacks Against/through Build/ Deployment Tools
Exploited Code Repositories And Base Image Registries
Financial And Resource Exhaustion
Resource Abundance
Data Injection
Global Context Leaks
Improper Error And Exception Handling
Broken Or Insecure Authentication
Insecure Management Of Secrets
Insecure Logging/monitoring
Sensitive Logging/monitoring
Insufficient And Insecure Logging/ Monitoring
Conduct Threats
Multiple Service Provider Threats

The Threat Landscape

5.2.2 Application Owner Setup Phase Threats

We name the bundle of threats related to the Application Owner preparation of workload assets, for deployment including all necessary code, images, CI/CD work, provisioning and configuration of cloud resources, etc. - the **Application Owner Setup Phase Threats**. These include both a set of threats that are unique to Serverless and a set of threats that exist in systems such as Microservices but aggravate with the use of Serverless to the expected number of different code fragments running or starting up in the system over the course of time:

Application Owner Setup Phase Threats that are unique to serverless include threats resulting from access permission or misconfiguration:

- Broad and generic permissions
- Broad and generic access to Events
- Broad user privileges over serverless control
- Weak Configuration

Application Owner Setup Phase Threats that *aggravate with the move to serverless* include:

Delivery Pipeline related threats either in the CI/CD pipeline or the dependencies:

> - Exploitable repositories and base image registries
> - Attacks against/through build/deployment tools
> - Vulnerable dependencies
> - Vulnerable base images

Service setup related threats resulting from misconfiguration:

> - Misconfiguration or vulnerabilities of associated cloud services

5.2.3 Application Owner Deployment Phase Threats

We name the bundle of threats related to the Application Owner deployment of workloads assets, including all serverless associated cloud and off-cloud assets, the **Application Owner Deployment Phase Threats**. Again we divide between threats that are unique to Serverless and those which aggravate with the use of Serverless:

Application Owner Deployment Phase Threats that are unique to serverless include:

Runtime related threats contributed by the design and implementation of the Callable Units:

> - Data Injection
> - Global Context Leaks
> - Improper Error & Exception Handling
> - Broken or insecure Authentication

Threats related to the Pay as go you nature of Serverless:

> - Financial and resource exhaustion (if limits on resources have been set)
> - Resource Abundance and unintended expenses.

Application Owner Deployment Phase Threats that *aggravate with the use of Serverless* include:

> - Insecure management of secrets
> - Insecure logging/monitoring
> - Sensitive data in logs and metadata
> - Insufficient and insecure logging/monitoring

5.2.4 Service Provider's conduct Threats

We name the bundle of threats related to the Service Provider services that the Application Owner consumes, including the entire stack used by the Service Provider and any dependencies, personal and other assets used to form the Serverless or associated services. The **Service Provider conduct's Threats**. These threats are unique to Serverless but have parallels in other services:

- Vulnerable/malicious service base image
- Vulnerable service runtime
- Leak between Callable Unit invocations
- Leak between different Callable Units
- Serverless service correctness
- API/Portal/console vulnerabilities

5.3 Threat Model - 25 Serverless Threats for Application Owners

The various threats for Serverless are provided in the table below.

Sr. No	Threat Summary	Threat Description	Mitigations (Security Controls)
		Application Owner Setup Phase Threats (A)	
		Unique to Serverless	
1	**Broad and generic permissions** Not maintaining the least privilege principle for Callable Units.	Application Owners may define the set of privileges that each Serverless Callable Unit will have while running. Excessive permissions can be taken advantage of as part of an attack.	See section 6, 6.3.5, 6.3.7.1, 6.3.7.2. etc.
2	**Broad and generic access to Events** Not maintaining the least privilege principle for the initiation of events triggering the Callable Unit.	Application Owners may define who may initiate Events that will trigger the Callable Units. Broader access greatly simplifies the execution of attacks. Especially in event-driven serverless architectures, this has an impact on the attack surface.	See section 6, 6.3.5, 6.3.7.1, 6.3.7.2. etc.
3	**Broad user privileges over serverless control** Not maintaining the least privilege principle for the DevOpsteam.	Application Owners may define who may have access to set up the Serverless service, image store, etc. Broader access adds additional potential paths for attackers and increases risks from insiders.	See section 6, 6.3.5, 6.3.7.1, 6.3.7.2. etc.

4	**Weak Configuration** Mismanagement of Serverless configurations and configuration drift can leave the platform and resident applications vulnerable.	Many services for hosting serverless applications are configured insecurely. Specific configuration parameters have critical implications for the overall security posture of applications and should be given attention -- for example, who can assume a Role to execute a function, and what can you achieve based on that assumed role?	See section 6, 6.3.7.2. 6.3.2 6.3.3 6.3.4, etc.
Aggravate with Serverless			
5	**Misconfiguration or vulnerabilities of associated cloud services** Additional cloud services that work in concert with the Serverless service to construct the workload may be misconfigured or vulnerable.	Often the security of the Callable Units depends on the security of the associated cloud services being used. For example, a Callable Unit may depend on the security of a secret service or the security of an Identity and Access Management system, etc. A Callable Unit may also rely on services owned by third parties as part of a supply chain. Hence the dependencies on other services and misconfigurations in those services being used as resources as part of the serverless application can impact the integrity of serverless functions.	See section 6, 6.3.7.1, 6.3.7.2, 6.3.3, etc.
6	**Exploitable repositories and base image registries** Vulnerability in the repositories and registries used to store the library dependencies and base images.	An attacker may attempt to incorporate malicious code by identifying vulnerabilities in shared (public or private) code repositories and image registries. This threat increases in Serverless due to the potential drastic increase in the number of independent Callable Units.	See section 6, 6.3.3, etc.
7	**Attacks against/through build/deployment tools** Vulnerabilities and misconfiguration of CI/CD automation tools used to build and deploy Callable Units and Events.	As part of CI/CD practices, automated tools are often used to construct the Callable Unit and deploy it (including the Events that will trigger it). Such automation requires providing the tools with elevated permissions to store Callable Units and set up Serverless cloud services. Attackers may use these elevated permissions to incorporate malicious code into a target application or as a way to cause a denial of service concerning serverless application updates.	See section 6, 6.3.7.1, etc.

8	**Vulnerable dependencies** Vulnerabilities or malicious code in any 3rd party library the Callable Unit may result in a supply chain attack.	Application Owners Callable Units often use multiple 3rd Party Library dependencies. Such libraries may include existing or newly discovered vulnerabilities. Further, a malicious contributor may embed malware in such libraries --This applies to all applications and services, including Serverless microservices.	See section 6, 6.1, 6.2, 6.3.7.1, 6.3.7.2, etc.
9	**Vulnerable base images** Vulnerabilities in base images used to form Images for an image-based Serverless service.	Base images used by Application Owners to construct images under Image-based Serverless are susceptible to many types of existing or newly discovered vulnerabilities in pre-installed dependencies and may also include pre-installed malware.	See section 6, 6.3.3, etc.

Application Owner Deployment Phase Threats (B)			
Unique to Serverless			
1	**Data Injection** Serverless Callable Units receive inputs during activation from various events - each such event represents a potential threat for data injection.	Injection flaws occur when untrusted input is passed directly to an interpreter or is executed before being properly vetted and validated. Such flaws are often part of an attack. AnyMost serverless architectures provide a multitude of event sources as a potential vector for a data injection attack. Event data injections can also break the orchestration of functions to execute a business function and cause a Denial of service.	See section 6, 6.3.7.2 , etc.
2	**Global Context Leaks** Serverless global context may allow, for example, to maintain tokens across invocations of the Callable Unit (e.g., saving the need to re-authenticate against Identity Management per invocation). The global context may leak sensitive data between requests.	Since different invocations of the Callable Unit are often used to serve data owned by other users of the workload, data leak between additional requests of the Callable Unit is a threat. Sensitive data may be left behind in the container and might be exposed during subsequent invocations of the function. Malicious data may be left purposely behind to attack future invocations of the function.	See section 6, 6.3.7.2, etc.

3	**Improper Error & Exception Handling** Cloud-native debugging options for serverless-based applications are limited (and more complex) when compared to debugging capabilities for standard applications.	Improper error handling can create vulnerabilities and allow malicious actions such as buffer overflow attacks and denial of service attacks. Verbose error messages could result in unintended disclosure of information to attackers --This is true for all applications and serverless as well in terms of exposure of metadata and resources.	See section 6, 6.3.7.2, etc.
4	**Broken or insecure Authentication** Improper authentication of the identity of the source of the event and/or the identity of the user/process initiating such event.	Often, Callable Units are required to affirm the identity of the entity behind the event being sent. An attacker will seek to exploit any vulnerability in the authentication mechanism used.	See section 6, 6.3.7.2, 6.3.5, etc.
5	**Financial and resource exhaustion** Serverless as a mechanism for an offender to cause significant unplanned expenses	An attacker may take advantage of the fact that Serverless is a "Pay as you Go" service and may force the Application Owner to pay significant unplanned expenses by creating many fake Events that invoke the Application Owner Callable Units and/or by initiating Events that result in long processing times (e.g., by exposing some other weakness in the code or a dependency).	See section 6, 6.3.6, etc.
6	**Resource Abundance** Serverless as a mechanism for an offender to tap into an endless pool of compute resources	An attacker may take advantage of the fact that Serverless is offered as an unlimited pool of resources.Given the vulnerabilities, the attacker may also be incentivized to exploit the abundance of computers available for the Callable Unit and utilize it for his gain, e.g., through crypto-mining or to initiate an attack on some third party.	See section 6, 6.3.7.1, 6.3.7.2, 6.3.3, etc.
Aggravate with Serverless			
7	**Insufficient and insecure logging/monitoring** Insufficient situational awareness for security incidents and the inability to investigate security breaches.	Insufficient logging will hamper an organization's ability to respond promptly to attacks/breaches and make it difficult or impossible to perform forensic analysis.	See section 6, 6.3.7.1, 6.3.7.2, 6.3.3, etc.

8	**Insecure management of secrets** A leak of secrets used by the Callable Unit that could lead to unintentional access to portions of the system or enable privilege escalation.	Often, Callable Units are required to access specific cloud or external resources when triggered. To be able to do so, Callable Units may need to obtain secrets. An attacker can take advantage of situations such as when secrets are insecurely stored or when a standard set of credentials are used. Like any other application in serverless functions, credentials and secret leaks can lead to impersonated identity and data leaks.	See section 6, 6.3.7.1, 6.3.7.2, etc. (see #3. Excessive Data Exposure)
9	**Insecure logging/ monitoring** Exposing logging data to an attacker or allowing an attacker to remove logs.	Insecure logging could enable an attacker to troubleshoot their attacks or delete traces of their actions preventing discovery and forensics. At the same time, the function owner may not detect any security issues and may not respond to them, thus impacting the overall integrity of the serverless applications.	See section 6, 6.3.7.1, 6.3.7.2, 6.3.3, etc.
10	**Sensitive logging/ monitoring** Logging sensitive information with security and privacy implications	Callable Units and Events may expose sensitive data via logging and monitoring systems, including secrets, PIIs, user data, etc.	See section 6, 6.3.7.1, 6.3.7.2, 6.3.3, etc.

Service Provider Conduct's Threats (C)			
Unique to Serverless			
1	**Vulnerable/malicious service base image** Under Function-based Serverless, the base image chosen by the Service Provider may include vulnerabilities/malware.	Images used by Service Providers include multiple 3rd Party dependencies. Such images are susceptible to existing or newly discovered vulnerabilities and may include pre-installed malware. Considering that service providers manage the underlying stack of the platform, this can impact the security posture of serverless applications.	See section 6, 6.3.3

2	**Vulnerable service runtime** Under Function-based Serverless, the runtime is configured and often augmented by Service Provider code, which may result in a vulnerable runtime.	Under Function-based Serverless, the base image is often augmented with additional code to form services and monitor the Service Provider. When deployed, the configuration of the executing containers is set by the Service provider--This may result in potential vulnerabilities such as open ports or other management facilities integrated into the runtime environment. Hence it is pertinent that serverless application security controls are evaluated in full context.	See section 6, 6.3.6 (Kubernetes risks & controls : deep dive) 6.3.7.1, etc
3	**Leak between Callable Unit invocations** Vulnerabilities in the Serverless service via isolation between invocations of a given Callable Unit are broken outside of the defined Serverless service contract.	Since different invocations of the Callable Unit often serve data owned by multiple workload users, data leak between invocations is a significant threat. For example, a Callable Unit reused to serve a different end-user or session context may leak a prior user-sensitive data left behind. Alternatively, malicious data/state left purposely behind may harm subsequent users.	See section 6, 6.3.7.2, etc.
4	**Leak between Different Callable Units** Vulnerabilities in the Serverless service via isolation between Callable Units executed on top of the same runtime environment instance in-sequence are broken. E.g., Function 1 ends, and Function 2 is loaded under FaaS on top of the same runtime environment without proper cleanups.	Since other Callable Units have different cloud and data access privileges, maintain different identities and secrets, have various exploitable vulnerabilities, and so forth, leak between different Callable Units is a significant threat. For example, if a serverless execution environment is reused to serve one Callable Unit and then another (of the same/different Application Owner), sensitive data may be left behind, or malicious data/state may intentionally be left behind to harm subsequent users, utilize subsequent privileges, identities and secrets or exploit the vulnerabilities of subsequent Callabler Units.	See section 6, 6.3.7.1, etc

5	**Serverless service correctness** Under Serverless, the workload is combined from fragments of Application Owner code glued by the Service Provider. The correctness of the Serverless core service, therefore, directly affects that of the workload.	Unlike with non-serverless, microservice cloud services, the security of the Workload is dependent on the correctness of the Service Provider events system, image management, and access control to the running instances of the Callable Units. The threat is for these systems to fail under certain conditions opening a door for attackers. E.g., send the wrong event, initiate the improper function or component, provide the wrong privileges, etc. Hence in addition to authentication and Authz, orchestration of functions and validation of key events before execution is essential.	See section 6, 6.3.2, 6.3.4, 6.3.7.1, etc
6	**API/Portal/console vulnerabilities** Vulnerable APIs allowing users to remotely configure and manage the Serverless platform, either using web APIs, throw CLI or throw a web interface.	In the case of a management portal or console could result in unauthorized access to the platform on multiple levels, resulting in the ability to modify (and weaken) configurations and perform reconnaissance activities on the environment. Other applications thru APIs may call serverless applications. As part of the serverless functions, a call may be made to additional resources and services; thus, secure APIs, their input outputs, and context are critical to the overall security of Serverless functions.	See section 6, 6.3.6, 6.3.1, 6.3.7.1, etc.

5.4 The Uniqueness of Serverless Threats

(This section further explains the above threats in the context of serverless applications)

In this section, we discuss aspects of Serverless compared to other IT environments and how Serverless affects threats identified in the previous sections. We do so to elaborate and highlight the significance of specific threats as relevant to the serverless environment. As we contrast the manifestations of these threats in Serverless compared to other IT environments, clarity about the uniqueness of Serverless Security begins to emerge.

All the threats mentioned in this section are labeled in bold and described in the previous sections.

5.4.1 Application Owner Setup Phase Threat (Ref: 5.3 (A))

In this subsection, we detail the differences between Serverless and other environments related to Setup Phase Threats and identify the aspects where Security officers and practitioners need special attention.

Aspects of fragmentation

Maintaining the Least Privilege principle is especially hard in an environment where the sheer amount of configurable entities and parameters is vast. When using Serverless services, the workload is fragmented into small Callable Units, each with a set of parameters that govern its security. These include users' credentials and access rights, systems allowed to modify and set up the Callable Unit, credentials and access rights used by the Callable Unit when executed, control over logging, monitoring, usage, and other notifications related to the Callable Unit. This introduces new challenges to Application Owners seeking to protect against threats due to **"Broad and generic permissions," "Broad and generic access to Events"** and **"Broad user privileges over serverless control."**

Further, creating and later maintaining a secure configuration of multiple fragments is also a challenge. The number of fragments complicates handling the identified threats above, such as **"Weak Configuration"** and **"Misconfiguration or vulnerabilities of associated cloud services."**

Even when the Application Owner ensures secure configuration for all components, and uses a well-restricted set of privileges during initial deployment to production, the configuration and privilege-sets may drift as the application matures due to the new components/services, dependencies, and requirements for broader access, etc.

At the same time, Serverless offers a more opinionated, well-structured environment in which to deploy. The service provider first processes inbound traffic before being propagated to the Callable Units. In general, Serverless is a less flexible environment than microservices or VMs and reduces threats due to **"Weak Configuration."** Examples of the reduced threats include CSP ensuring that no inbound traffic will reach Callable Units outside of the Events declared per Callable Unit or that the latest recommended version of TLS is always used. The CSP opinionated environment, therefore, may reduce the attack surface and the opportunities for misconfiguration.

As the number of fragments grows, so are the challenges from supply chain vulnerabilities. Without proper handling, each fragment may potentially use a different set of dependencies. Each may use other repositories and build/deployment tools. When using Image-based Serverless, each may potentially depend on a different base image. As a result, the attack surface of the workload increases, with the increased number of various dependencies comes an increase in the number of potential vulnerabilities. These threats were identified above as: **"Vulnerable dependencies," "Vulnerable base images," "Exploitable repositories and base image registries," "Attacks against/ through build/deployment tools."**

5.4.2 Application Owner Deployment Phase Threats (Ref 5.3 (B))

In this subsection, we highlight the novelty of Serverless as it relates to Deployment Phase Threats.

Aspects of Data Injection

Serverless takes advantage of a vast scope of events from various sources that Callable Units can process. Each type of event used, regardless of its source, is a potential **Data Injection** threat. Though the Service Provider may provide specific protection by controlling that the right type of events come from the correct source (and even given the proper basic event format), such protection does not prevent an attacker from exploiting injection flaws. Any information that an offender may control (e.g., using another vulnerability or misconfiguration or after obtaining false credentials) and is passed to the Callable Unit as part of the event should be considered a threat.

For example, an event to a Callable Unit may be sourced from a notification about an object loaded to object-store. This event may include information about the loaded object - information that may be under an offender's control. Similarly, an event to a Callable Unit may also be sourced from a web request that may be under an offender's control. Suppose Callable Unit uses the information carried by the Event directly (without proper screening) as part of a database access request. In that case, an offender may use well-crafted information to modify the database or read confidential information.

Aspects of Global Context

Multi-tenancy is supposedly handled "by design" in Serverless. Each invocation of a Callable Unit handles a specific event and hence can be attributed to a specific tenant (ignoring for a minute cases where the event is service-wide and not associated with a specific tenant). This "by design" multi-tenancy support holds as long as the information does not leak between Callable Unit invocations. Since CSPs may use the same Serverless instance to process multiple events in sequence, Application Owner coders need therefore ensure that information from one invocation due to one event does not leak to subsequent invocations due to subsequent events.

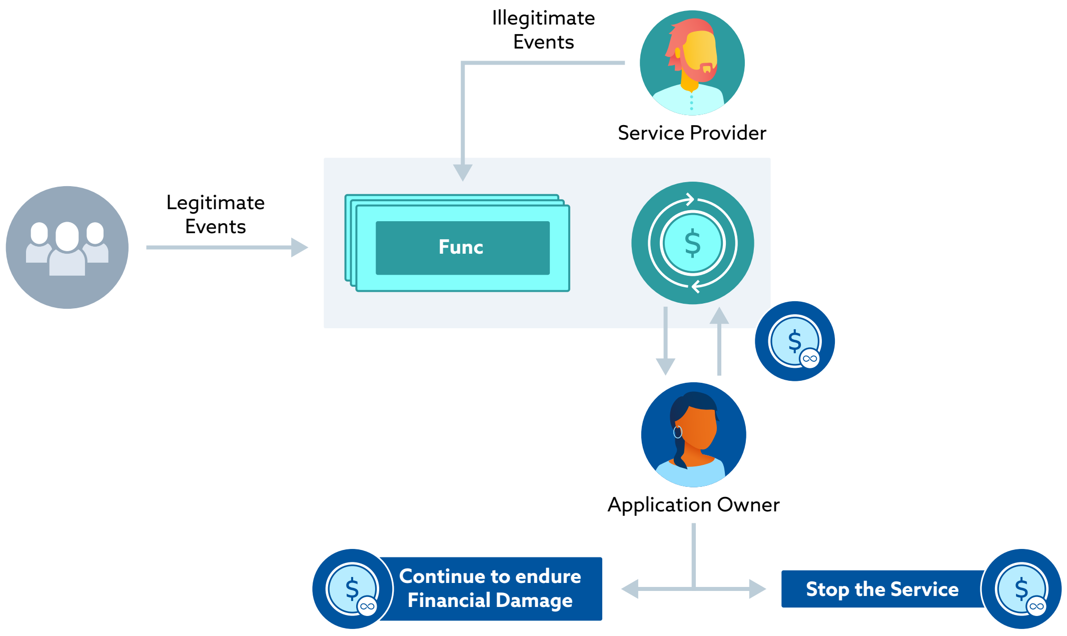
When a Callable Unit/function instantiates, it may need to initialize to authenticate, obtain secrets/ tokens and establish connections and context to supporting services. Such initialization only needs to be performed once for the sequence of invocations handling a series of events. TheCallable Unit requires a global context to maintain (cache) the tokens, secrets, open connections, etc. Such global context is required for a Callable Unit to be efficient; otherwise, it will need to authenticate and connect with other systems with every event, even when a high rate of events arrive in sequence. This will prove to be a tremendous overhead for the Callable Unit's actual function, which is typically somewhat limited and short in time.

Introducing a global context brings about the "Global Context Leaks" threat as identified above, where information from one invocation may leak to subsequent invocations.

Aspects of losing control over the assignment of compute resources (The "Pay as you go" hazard)

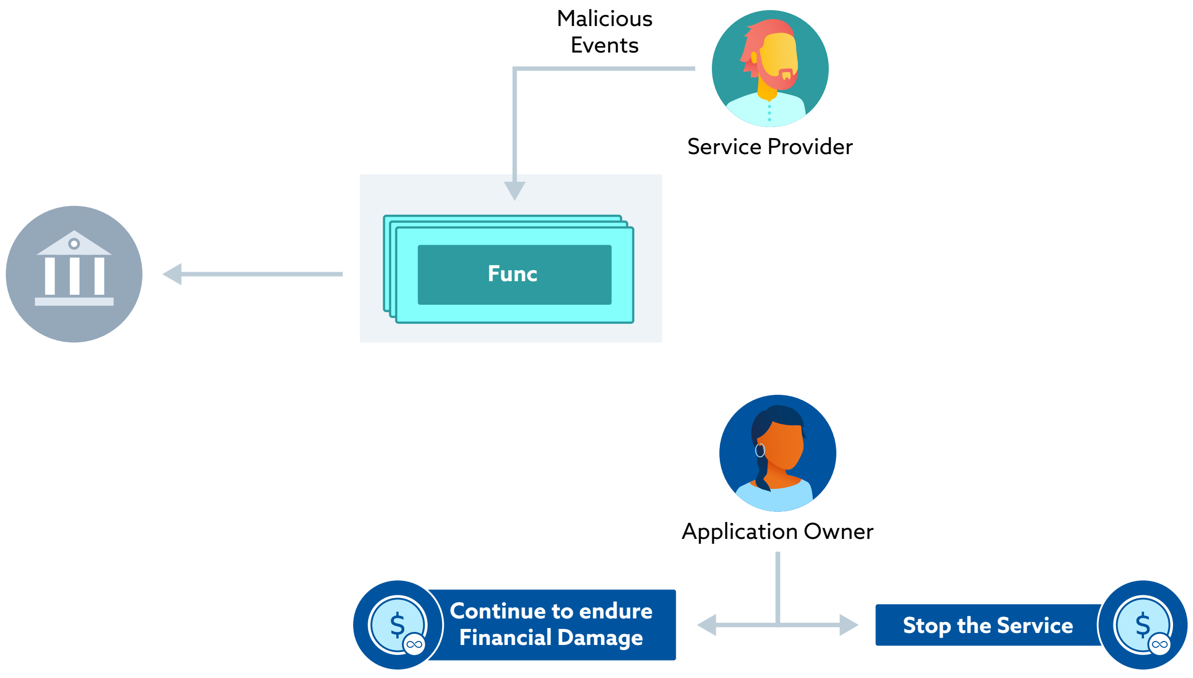
Unlike other options such as using VMs and Microservices, Serverless does not curb the amount of compute resources used by an Application Owner to some limit defined during setup. The Application Owner loses control over the number of resources used at any given time and is required to retake such control by setting up budgets and forecasts and processing billing notifications from the Service Provider. One security aspect of this novelty is that an Application Owner faces the **"Financial and resource exhaustion"** threat described above without proper monitoring and processing.

One example of this threat is that an attacker can cause an *Induced Denial of Service* to an Application Owner by continuously imposing a load of events on the Application Owner's workload. Even assuming sufficient resources are made available by the Service Provider -- in order to process both legitimate and non-legitimate Events, a continuous high load of non-legitimate Events will introduce unexpected expenditure on the Application Owner -- potentially to a level that forces the Application Owner to turn the service off or to limit it which could be out of line with their business intentions and needs. For example, *Induced Denial of Service* may materialize as part of a distributed attack by an offender seeking to force the Application Owner to take financial damage, either directly by increasing his cloud costs or indirectly by turning off his workload.



Induced Denial of Service through Direct Financial Damage

An *Induced Denial of Service* can also result as an outcome to the "**Resource Abundance"** threat materializing via which an offender identifies a way to take advantage of the Callable Unit to further his cause. For example, consider an offender able to get the Callable Unit to send traffic to a destination under his control.he unlimited resources can then be used as part of a Denial of Service attack on a third party. Note that Serverless does not limit Callable Units from initiating outbound traffic. An attack like this cannot be prevented by processing periodical billing notifications from the service provider as it requires an immediate response by the Application Owner. Once such an offense has started, the Application Owner is immediately required to stop it, hopefully before the entity is attacked, pinpointing the attack source to the Application Owner's Callable Units. Stopping the now offensive behavior of the Callable Units may require shutting the service down, thus forcing an Induced Denial of Service.



Induced Denial of Service through Damage to Reputation

In a third example, if the offender can get the Callable Unit to execute their code (e.g., through a supply chain attack or by exploiting code or dependency vulnerability), the Callable Units may offer a conveniently abundant crypto-mining resource. This threat is challenging to detect if the offender keeps the usage level within the boundaries of the legitimate workload. Executing their code will also allow offenders to utilize the Callable Unit as a general-purpose botnet where activity is triggered by an event that lasts for the Callable Unit time limit. The abundance of resources allows keeping the botnet away from being detected. A Callable Unit may further persist the botnet before being terminated by triggering one or more other instances of the Callable Unit. Such a botnet may use a client-server command and control architecture as outbound traffic is available under serverless. Such a botnet may also use a modified peer-to-peer command and control by repurposing the Events associated with the Callable Unit. Note that once the Callable Unit code is under the offender's control, existing Events may offer the necessary communication channel between bots.

Therefore, the Resource Abundance is a significant threat to Application Owners of Serverless and needs to be appropriately dealt with to avoid being exploited by offenders.

Aspects of complexity resulted due to fragmentation

Serverless architectures promote a fragmented system design. Applications built for such architectures may contain dozens (or even hundreds) of distinct serverless functions, each with a specific purpose. These functions are weaved together and orchestrated to form the overall system logic. Some serverless functions may expose functionality via public web APIs, while others may serve as an "internal glue" between processes or other functions. Some functions may consume events of different source types, such as cloud storage events, NoSQL database events, IoT device telemetry signals, or even SMS notifications. These create significant complexities and, therefore, can be potential "threat" opportunities for offenders. These are all vectors for potential authentication-related vulnerabilities identified above as "Broken or insecure Authentication" and "Insecure management of secrets."

Another example related to the resulting complexity of the system is the threat that the Application Owner faces of losing any ability for situational awareness. A complex and fragmented workload where parts of the control and data path are under a Service Provider's control (i.e.,Event System) and others are under the control of the Application Owner (Event Processing) may lead to a complete loss of situational awareness indicated above as **"Insufficient and insecure logging/monitoring."**

Aspects of code robustness and correctness

Serverless is unlike Microservices and VMs, where developers can completely replicate the execution environment locally. Since a serverless-based workload offloads portions of the data and control path to the Service Provider, more development and testing need to be done on-cloud than alternative compute options. Cloud-native debugging options for serverless-based applications are limited (and more complex) when compared to debugging capabilities for standard applications. This reality is especially true when a serverless function utilizes cloud-based services that are not available when debugging the code locally. This threat is identified above as **"Improper Error & Exception Handling,"** which may manifest as information disclosure through unnecessarily verbose error messages or hidden vulnerabilities. There are best practices and standard error message templates; it is pertinent for serverless deployments to be thorough from a security perspective testing of the serverless applications is done and error messages validated to be generic and that they don't leak any data or metadata.

5.4.3 Service Provider's Deployment Threats (Ref.: 5.3 (C))

In this last subsection, we consider threats originating from the Service Provider and explain those aspects concerning Serverless.

Aspects of isolation

As with any service, a prime security concern for Service Providers is establishing unbreakable isolation between cloud tenants. Serverless introduces many new challenges when it comes to the isolation provided to cloud users. Both the Event system and the Compute system are attack surfaces for a malicious tenant. Potential vulnerabilities in these systems, together with many other potential hazards related to the core serverless services, were identified above as '**Cloud service vulnerabilities".**

With Serverless, unlike VMs or Microservices, new isolations need to be established by the Service Provider. As Serverless Callable Units get started and terminated frequently (and since cloud users pay only on actual compute time consumed), it is in the interest of the Service Provider to aggressively reduce the overhead due to spinning up and down of instances. Service Providers are therefore reusing Callable Units which are already spun to serve multiple Events. Since different Events can be associated with varying tenants of the Application Owner workload, isolation between invocations of the Callable Unit is required and expected by the Application Owner. This introduces a unique threat to Serverless identified above as **"Leak between Callable Unit invocations."**

A second threat identified as **"Leak between Different Callable Units"** considers isolation issues between Callable Units that may be executed in the same runtime environment. Depending on the Service Provider, when a function under FaaS completes execution, a subsequent function may also be implemented in the same runtime environment --This introduces a sizeable potential attack-surface for offenders to collect the residual data left behind from previous Callable Units and modify the runtime environment to potentially exploit a vulnerability in a subsequent Callable Unit.

Aspects of shared responsibility

Any use of services from a Service Provider results in a shared responsibility for the workload security. However, Serverless takes that shared responsibility to the extreme. Under VMs and Microservices, the responsibility-divide between the Application Owner and the Service Provider is well defined, at the boundary of the container or VM, and the execution of the workload is entirely under the control of the Application Owner (unlike data store, network, etc. where the responsibility lay at the Service Provider side). When it comes to Serverless, the boundary is vague at best.

Under Function-based Serverless, the Application Owner code runs within a runtime controlled by the Service Provider. Security can be determined by the design of the code, the security of the runtime, and the interaction between the two. The Application Owner's code may rely and be tested on a particular image with a pre-installed set of libraries. Once that image gets updated by the CSP, the interaction between the code and the new image may introduce Callable Unit vulnerabilities. Further, under Serverless workload, functionality is formed by Application Owner code fragments joined by Events handled by the Service Provider. Again, security can be determined not only by the design of the code or the security of the Event System but also by the interaction between the two. For example, when the Application Owner code uses the Event System in a certain way.

These examples aim to clarify that aspects of shared responsibility in Serverless are more complex when compared to VMs and Microservices. This threat is identified under **"Serverless service correctness"** discussed in the previous sections.

6. Security Design, Controls, and Best Practices

Microservices are independently released and deployable services that support a business domain. A service may encapsulate functionality and make it accessible to other services via APIs. For example, one service might conduct accounting, another account management transactions, and yet another reporting, but together they might constitute an entire account management banking system.

The service-oriented architecture is flexible and does not specify how service boundaries should be drawn as long as they are independently deployable. Being technology agnostic is one of the key advantages of microservices. Some key features of Microservices are:

1. Independent deployability
2. Modeled Around a business domain
3. Owning their state
4. Size and flexibility

Based on domain driven architectures (Martin Fowler), there has been much debate on Event driven microservices - Synchronous and Asynchronous event driven microservices. Synchronous services have drawbacks like dependent scaling, point-to-point coupling, API dependencies, failure management, data access dependencies, and management challenges. Event-driven asynchronous Microservices have gained popularity due to flexibility, technology independence, business requirements flexibility, Loose coupling, continuous delivery models, etc.

While the industry is still evolving, there will also be hybrid architectures where some services are monolithic, and some are event-driven functions-based microservices.

Some Architectural best practices for Event-Driven Microservices, which essentially use FaaS services, are described below:

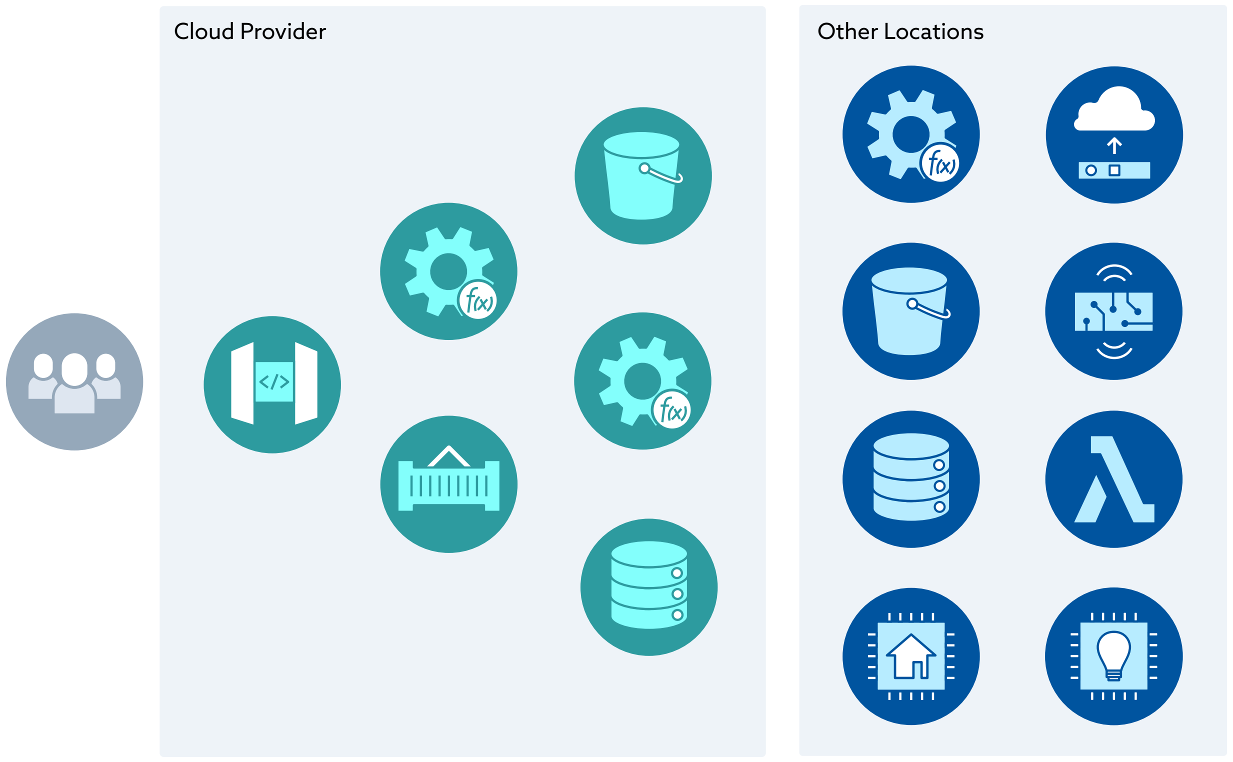
Interfaces - API & Contracts

- Automatic retries
- Concurrency
- Scale requirements
- Message & payloads
- User and Service/session Identity
- How to handle failures, Timeouts, retries etc., Rate limits

A serverless environment brings many architectural changes that force the evolution of security controls. Here are some examples:

- **Events**
    - Under Serverless, Callable Units communicate via Events. These events travel between the administrative domains. A Callable Unit, therefore, may send data to other Callable Units while potentially crossing untrusted boundaries.
    - The number of Callable Units being triggered increases due to the workload fragmentation, Callable Units' short lifespan, and the nature of event-driven architectures, e.g., every time a file changes, several functions might be used.
    - Chained functions are being used, some of which pull data from several trusted or untrusted sources.
- **The Network**
    - The network construct is under the control of the Service Provider, which is now the only entity with access to network logs.
    - Networking chokepoints (e.g., serverless platform performance issues) are under the control of the Service Provider.
    - The new Perimeter, any network security controls need to be reinvented or replaced with a Zero Trust model and identity control.
    - In transit network security is primarily the responsibility of the Service Provider. Consumers must configure application transit security.
- **Lifecycle**
    - Lifecycle is quite different; functions can have a duration of milliseconds before they get destroyed (limited lifecycle), which may affect security monitoring systems.
    - The short lifespan may affect IAM and Secret Management systems as Callable Units are spun on and off.
    - The sheer number of functions/images to control and handle affects the controls needed to ensure integrity.
- **Attack Surface (as Discussed in detail in Chapter 5)**
    - The traditional attack surface is decreasing due to the infrastructure abstraction offered by Serverless.
    - The number of cloud service instances is significantly increased, and with it the opportunity for misconfiguration, the complexity of setting proper IAM roles, and setting up proper security processes for all workloads in the application development lifecycle.
- **Monitoring**
    - Aspects like logging and monitoring that have been traditionally coupled inside the application (e.g., Nginx/Apache) are now abstracted from the user. This requires redesign and/or new requirements on developers to add proper logging and monitoring events along with the code.
- **Security Controls**
    - Traditional security controls like firewalls, IDS/IPS, and SIEM's do not efficiently deal with this new paradigm of increased connectivity and integration. As such, it requires a more distributed security architecture to secure the serverless components.
    - New approaches to serverless security are emerging and need to be considered by Application Owners. We expand more on this topic in the *"Futuristic Vision for serverless security"* chapter of this paper.

Below we have a diagram on the serverless architecture where the de-perimeterization is quite evident:



Serverless Security view-point

The image above depicts a collection of services (loosely coupled) that might compose an application without necessarily having the traditional security trust boundaries, such as a static firewall, etc.

The serverless paradigm uses functions that, when correctly coupled with third-party services (e.g., hosting), allow an organization to run end-to-end applications without needing to care about traditional infrastructure management and other aspects of traditional security.

Appropriate security patterns are still emerging for Serverless, requiring a more thoughtful security approach for the Application Owner. Those patterns will take time to mature, with this and other papers in the area.

6.1 Design Considerations for Serverless

Application Architects have to be cognizant of both inherent weaknesses present in Serverless technology vs. weaknesses they can introduce in their design. Understanding these weaknesses will give them a better grasp of the security controls required to be implemented (these are explained in the next chapter).

Serverless architecture has many benefits, even from a security perspective, that are weighed as part of Secure microservices design considerations. Some of those benefits include:

1. Stateless and Ephemeral: Short-lived serverless functions are processing unencrypted data in memory for a short period of time. Serverless functions do not write to a local disk. Hence, functions that need to persist state rely on external data stores, thus *reducing the likelihood of exploits by attacks designed for long-lived targets.*
2. Each serverless function requires only a subset of data to perform its micro-focused service. *So as long as this function has the correct permission to access only the data it requires, then a successful exploit of a function should be more focused on what data it can potentially exfiltrate.*
3. Serverless applications run within containers managed by a CSP or within self-managed containers. Hence, they have some inherent security benefits of containers that run on immutable container images. Containers that do not require long-lived servers can easily be assigned continuously to patch container images and compute instances. *Lessening concerns of running on Vulnerable or Unpatched underlying infrastructure*

Some other considerations the architects and developers may want to consider are:

Vendor Lock-in: Functions may be using other Cloud services that may not be compatible with other service provider environments as integrations might be across various services and dependencies in a CSP environment. E.g., backend databases may be in RDS or MS SQL Azure and so on.

Deployment Tools Limitations and Execution Environment Limitations: Based on the application's need, it is pertinent to understand the tools and execution environment necessary and select the provider based on those needs. It is also important to explore potential limitations that may impact the overall functionality of the services.

6.1.1 Serverless Platform Design Impacts on the Serverless Microservices Security.

As introduced earlier, function/ Application Owners have a greater responsibility to manage and control applications. They are less concerned with the platform's infrastructure and security, which falls under the service provider's responsibility. With this shift of responsibility, the function/ Application Owner inherently loses some visibility and responsibility for the infrastructure. Now, from the function owner side, questions emerge like:

- Where are the functions running?
- What is the actual network exposure of the functions?
- Are the functions executed in an idle container already initialized by a previous execution as a "warm start" or a newly instantiated container as a "cold start"?
- Is any of the previous data still available in cached memory?

Let's start by understanding some of the inherent design characteristics needed during the design process for Serverless microservices.

Functions inherently have public-facing egress access.

In a serverless world, CSPs has more responsibility for security when compared to cloud customers. Cloud customers (CSC) have limited responsibility. For example, CSC is responsible for security within their application, for the secure configuration of Serverless components and services. The security provided by default by the CSP may not meet or address all the security requirements as required by all the customers (for example, due to compliance). API Gateways provided by the CSP are accessible anywhere on the public internet with access controlled via API keys. Customers would be responsible for setting up any supporting controls (transport-based access controls) based on their business needs.

Design for enhanced security using the following controls to limit exposure of data to the internet.

> i. Apply network policies to limit the endpoints that are reachable from that FaaS by using the Virtual Private Cloud (VPC) network policy configurations that are made available by the various CSPs.
> ii. Apply service or resource policies that allow you to limit the endpoints that can access your data stores/services Therefore, you can reduce the exfiltration paths for your data.

Examples include AWS VPC endpoints, Azure VNet Service endpoints, and Google VPC Service Controls.

By design, large applications when broken into microservices, make it difficult to have complete visibility into the serverless microservice deployed architecture.

Serverless Applications are built upon microservices architecture, allowing you to build many logically focused functions to run concurrently and scale out your processing.

However, as you continue to grow the number of available functions, whether for a single application or as distributed APIs for larger functional microservices, challenges arise amid creating full visibility of how that function is executed. For example, some questions that would be helpful to consider are:

> 1. Are all those functions executed within a VPC, especially if the criticality of the data requires that you minimize any public exposure?
> 2. Are all the roles you created for each function tuned to allow the minimum required privilege?
> 3. Did developers reuse pre-existing roles, so the functions that need to read data only use the same role as those that need to process and update your data values?
> 4. Suppose an application redesign now specifies the function will be launched by an event queue instead of an HTTP event. Has the HTTP event trigger been removed as part of the deployment options?
> 5. Are all the functions that can be triggered by an HTTP event behind an API gateway?

Some of the specific concerns regarding Serverless environments are -

> A. Inadequate Function logging & monitoring
> B. Insecure serverless deployment configuration
> C. Insecurely storing Application Secrets
> D. Insecure management of third party dependencies

A. Inadequate function logging and monitoring: *Controls to get detailed visibility into the execution environment of event-driven microservices/Functions are:*

> i. Inadequate function Logging: Logging provides a record of activities carried out by entities in a given period of time. It provides insight into the system/application and provides the ability to debug, assess and replay in case of an incident. A best practice for logging in Serverless is to structure the logged values. Structured logs are easier to parse. Another best practice is determining the desired log level or verbosity so the logs will provide value when needed. Logs should be leveraged to monitor anomalous patterns or conditions to alert and notify operations to take corrective actions. Ensure that you are using integrated logging that you can centralize to facilitate your overall application performance and security monitoring. Platform Provider logging will help collect statistics on the quantity, duration, and memory usage of your function's executions. Visibility of application errors is within your control by adding logging statements as needed within your serverless functions. For instance, does your error-logging enable you to identify if the failure occurred within a process you defined or from unexpected data input, or a result of a third-party functional process?
>
> ii. Inadequate function Monitoring: Use application and security monitoring tools to help surface visibility for how often a function is executed, and the logical execution path. One should monitor and discover all the APIs or endpoints that you expose and all your downstream or dependent APIs. You should monitor and discover which events and roles are executing your functions and any unexpected execution paths or methods.

B. Insecure serverless deployment configuration: Default Platform Provider configurations can impact the level of security needed for a particular microservice use case:

Serverless provides customization and configuration settings for any specific need or task (network policies, command-line interface).

Serverless security depends on the configurations of the functions and several of the upstream and downstream services that integrate with your Serverless microservice. You must understand the configurations of those PaaS services and all of your security hardening options. The probability of misconfiguring critical configuration settings can have a huge impact.

For instance, if using PaaS data services (examples: Amazon S3, EMR, or RedShift), one is still using any default configurations, and that configuration is still potentially exposing the data publicly. Are you using default Platform Provider IAM roles that allow your function to read and write data when your function only needs to read and process that data.

Suggested best practices:

> i. Security settings and policies must be defined based on the Microservice use case, and reliance on minimized default platform provider configurations. Determine how to securely configure each service based on the microservice and data security requirements and risk posture.
>
> ii. Serverless provides customization and configuration settings for any specific need, task (network policies, command-line interface). The probability of misconfiguring critical configuration settings can have a significant impact on serverless. Therefore, security testing should be performed for the microservices to ensure that your configurations meet

all of your security requirements. The security testing should help you validate the methods and roles that can access your functions or data. More importantly, application security testing will help reveal that your application logic or input validation is still exploitable by injection attacks and any other configuration weaknesses that still exist.

C. Insecurely storing Application Secrets

As applications are growing in scale and complexity, the need for storing and maintaining Application Secrets (API keys, encryption keys, etc.) becomes critical.

Out-of-the-box CSP offerings do not address the security needs of all tenants. Tenants should determine their level of security (PII data, sensitive data, regulatory compliance requirements, etc.) and upgrade security on top of the default CSP offerings. Some questions that a tenant could consider are - Are you designing a Serverless Application that will access and process confidential data or highly regulated data? If so, then the security requirements for how that data is protected increase, and you have to consider not only data exfiltration by external attackers but also malicious insiders.

Platform Providers are executing Serverless applications on compute instances not encrypted by default. Furthermore, Serverless applications most likely rely on PaaS data services offered by Platform Providers that are not encrypted by default. Out-of-the-box CSP offerings do not address the security needs of all tenants. Tenants should determine their level of security (PII data, sensitive data, regulatory compliance requirements, etc.) and upgrade security on top of the default CSP offering.

Suggested best practices:

i. Ascertain if your Platform Provider offers the option of deploying your Serverless code to be executed on a confidential compute instance [IEEE Spectrum, 2020]. This may help you decide between the use of Functions or other compute options to implement compensatory security controls to adequately meet application and data security requirements and risk posture.
ii. Ascertain the default or managed encryption options for each PaaS Data Service incorporated in your Serverless application andhold that confidential or highly regulated data. Ensure that your use of and configuration of the PaaS Data Service will meet your application and data security requirements. If not, then implement Application layer encryption as compensatory control.
Again Confidential compute is expensive and still evolving design decisions must weigh against the feasibility of the solution and the performance impacts that confidential compute will add to the execution of the services.

D. Insecure management of third-party dependencies: Serverless application reliance on third-party libraries

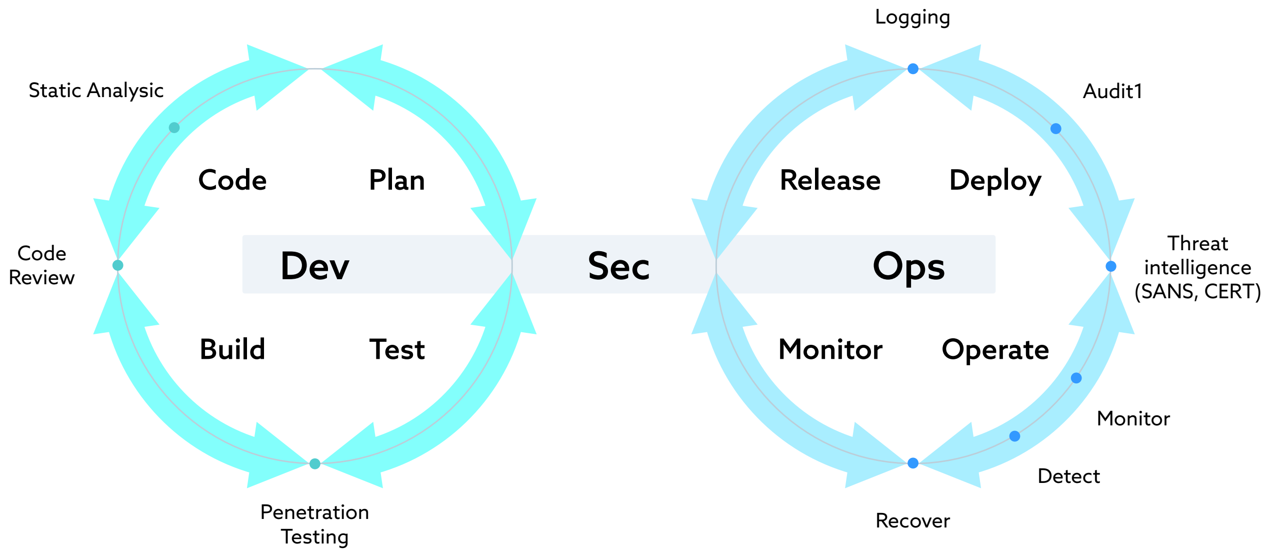
At times, the serverless function will depend on third-party software packages, open-source libraries and even consume third-party remote web services through API calls. Serverless application code and deployment vulnerability scanning have limited visibility of dependent third-party libraries. Especially concerning are any vulnerabilities introduced into those third-party libraries managed outside of your source code repository. It is wise to look at third-party dependencies as they could have vulnerabilities that could, in turn, expose serverless applications to attacks.

Suggested best practices:

i. Incorporate Source Composition Analysis of any third-party libraries into your deployments. This allows you to discover not only how extensive your dependencies are but if you are introducing risk with already known vulnerable components.
ii. Use Security monitoring solutions to identify vulnerable third-party libraries at run time and identify included libraries not being used. Remove redundant libraries to reduce the risk of unnecessarily including vulnerabilities.

In this section, we will briefly talk about security controls and best practices for securing Serverless.

Cloud services follow a shared responsibility model between the CSP and the CSC -- this is no different for Serverless. As serverless becomes adopted by various organizations, it is essential to understand platform service provider responsibility vs. customer responsibility in the Function and Image-based models. In the serverless architecture, CSP takes on the security responsibility of the cloud (securing the servers & operating system) leaving the tenants the security responsibility in the cloud with security controls such as IAAA (AuthN, AuthZ, audit logs), SDLC (code review, static analysis, build, test, release, deploy), data protection (at rest, in motion), policy enforcement (example: code review must be conducted by two peers, code should be release only after a clean vulnerability scan), etc.



6.2 Controls for FaaS

Note: All Controls which apply to FaaS are also applicable to Container Image-based Serverless as FaaS is a higher layer and can be built on top of Container Image-based Serverless. (This section mainly describes controls to be implemented if an Enterprise choses to build their own Private FaaS).

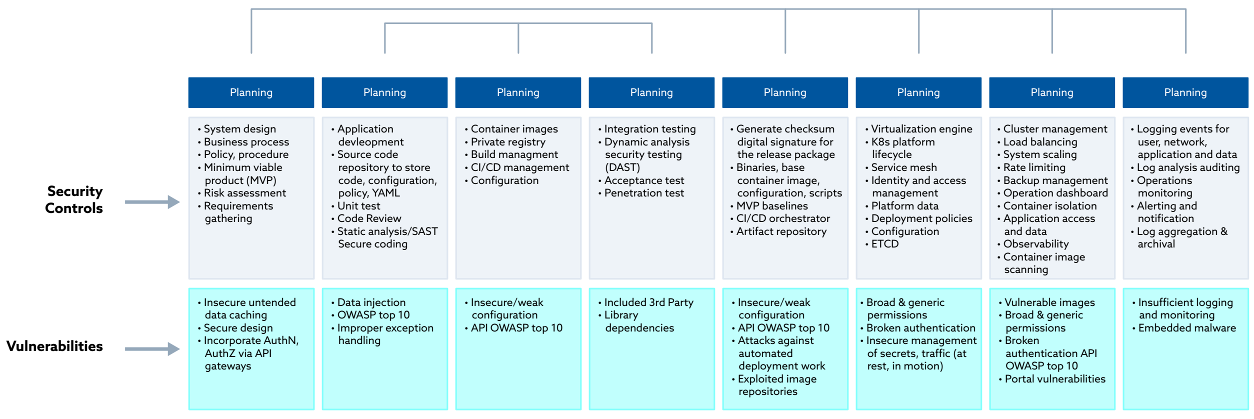
We are now moving to a FaaS model, where we have several open versions of FaaS. Security owners of Application teams can now add extensions (Lambda Extensions, Knative etc) and agents into containers running the functions - the best place to catch business logic level issues is in the Functions themselves.
The controls can be looked at in various ways:

> a. Platform configuration audit (code review, SAST, DAST, policy enforcement carried out as part of the CI-CD pipelines.
> b. Platform components and vulnerabilities (Cloud providers assume the responsibility for most of the security in serverless platforms. Viz. host operating system, Containers, orchestration service, service mesh, etc.) along with network policy violation detection, platform layer threat detection and management. This means, if an Enterprise decides to implement a Private FaaS, they will have to build all the layers of security controls.
> c. Function configuration audit (code review, SAST, DAST, policy enforcement can be bucketed again under CI-CD Controls and Runtime controls.
> d. Function components and Vulnerabilities (most serverless vulnerabilities) are related to programming; injection, broken authentication, misuse of access roles, deployment with known vulnerabilities or insecure storage of secrets, insecure deployment settings, improper exception handling, and insufficient logging and monitoring that can be bucketed under OWASP secure coding best practices.
> e. Identity and access management (user and application identity management, federation, SAML 2.0, access control, multi-factor authentication)
> f. Function workload security - system integrity monitoring, app allow listing, app hardening, anti-malware, exploit prevention, detection, and response.

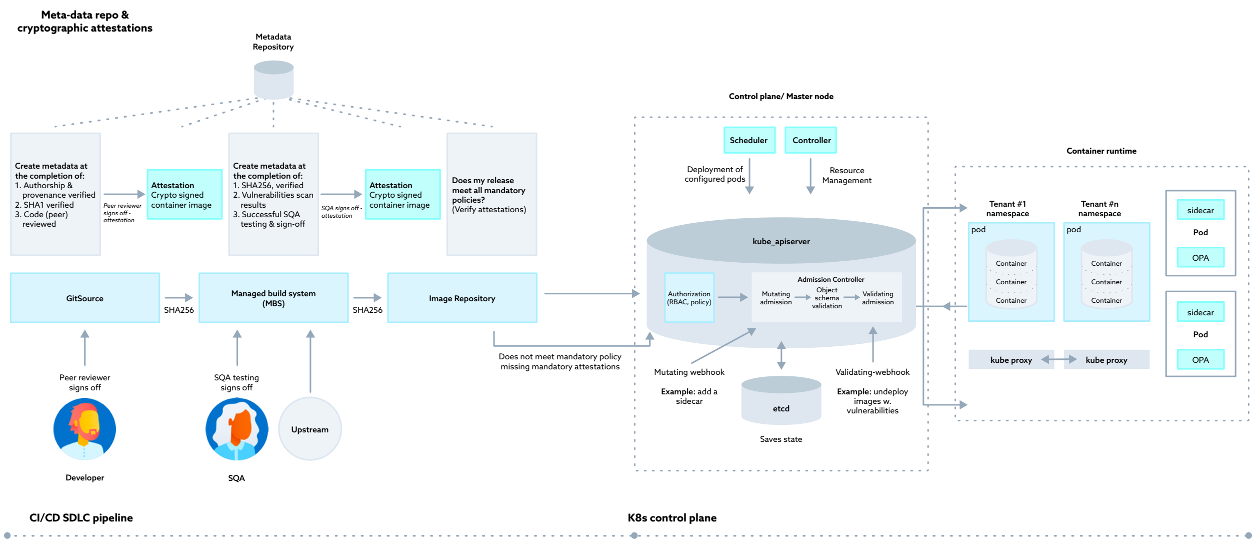
Again, if we need to define FaaS control categories, we can bucket them into high-level categories as follows:

> a. Platform Service Provider API and management controls & integrations
> b. CI-CD Pipeline security controls include components, config, vulnerabilities scans, etc.
> c. Identity and access mgmt
> d. Detection at Platform layer and Runtime detection controls for policy violations, failed functions/triggers, etc.

6.3 CI-CD Pipelines, Function Code, Code Scans and Policy Enforcement for Functions and Containers



	Planning	Planning	Planning	Planning	Planning	Planning	Planning	Planning
Security Controls	• System design • Business process • Policy, procedure • Minimum viable product (MVP) • Risk assessment • Requirements gathering	• Application devleopment • Source code repository to store code, configuration, policy, YAML • Unit test • Code Review • Static analysis/SAST Secure coding	• Container images • Private registry • Build managment • CI/CD management • Configuration	• Integration testing • Dynamic analysis security testing (DAST) • Acceptance test • Penetration test	• Generate checksum digital signature for the release package • Binaries, base container image, configuration, scripts • MVP baselines • CI/CD orchestrator • Artifact repository	• Virtualization engine • K8s platform lifecycle • Service mesh • Identity and access management • Platform data • Deployment policies • Configuration • ETCD	• Cluster management • Load balancing • System scaling • Rate limiting • Backup management • Operation dashboard • Container isolation • Application access and data • Observability • Container image scanning	• Logging events for user, network, application and data • Log analysis auditing • Operations monitoring • Alerting and notification • Log aggregation & archival
Vulnerabilities	• Insecure untended data caching • Secure design • Incorporate AuthN, AuthZ via API gateways	• Data injection • OWASP top 10 • Improper exception handling	• Insecure/weak configuration • API OWASP top 10	• Included 3rd Party • Library dependencies	• Insecure/weak configuration • API OWASP top 10 • Attacks against automated deployment work • Exploited image repositories	• Broad & generic permissions • Broken authentication • Insecure management of secrets, traffic (at rest, in motion)	• Vulnerable images • Broad & generic permissions • Broken authentication API OWASP top 10 • Portal vulnerabilities	• Insufficient logging and monitoring • Embedded malware



Meta-data repo & cryptographic attestations
Metadata Repository
Create metadata at the completion of: 1. Authorship & provenance verified 2. SHA1 verified 3. Code (peer) reviewed
Peer reviewer signs off - attestation
Attestation Crypto signed container image
Create metadata at the completion of: 1. SHA256, verified 2. Vulnerabilities scan results 3. Successful SQA testing & sign-off
SQA signs off - attestation
Attestation Crypto signed container image
Does my release meet all mandatory policies? (Verify attestations)
GitSource
SHA256
Managed build system (MBS)
SHA256
Image Repository
Does not meet mandatory policy missing mandatory attestations
Peer reviewer signs off
Developer
SQA testing signs off
SQA
Upstream
CI/CD SDLC pipeline
Control plane/ Master node
Scheduler
Controller
Deployment of configured pods
Resource Management
kube_apiserver
Authorization (RBAC, policy)
Admission Controller
Mutating admission
Object schema validation
Validating admission
Mutating webhook
Example: add a sidecar
etcd
Saves state
Validating-webhook
Example: undeploy images w. vulnerabilities
K8s control plane
Container runtime
Tenant #1 namespace
Tenant #n namespace
pod
Container
Container
Container
pod
Container
Container
Container
kube proxy
kube proxy
sidecar
Pod
OPA
sidecar
Pod
OPA

All code changes go through a peer review or manual code review or via automation (Static Analysis) before being included in a release. All changes/updates are submitted by a pull request and must be code reviewed by someone other than the person making the change. Once a change has been reviewed, it is merged into the master branch, tested, and subsequently deployed to production.

Static scanning

As applications are developed and deployed faster, an automated method of scanning "Function or Container code" before they deploy is critical. Static analysis tools analyze source code for issues integrated into the CI/CD pipeline. Having gates which prevent particular kinds of issues from being propagated becomes essential.

For scanning to be successful, the following are needed:

- Deep and clear visibility across environments - simple, easy and intuitive discovery and controls of assets to be scanned.
  - This would mean the ability to do code scannings, not for the complete code base but just the affected and changed code as soon as code is checked in.
  - Ability to look at all code dependent libraries, binaries, and changes in the libraries across other versions of code.
  - Scanning the deployment manifests of the Function-based Serverless and Container Image-based Serverless to look for issues in the configuration.
- Results accuracy - need the ability to control False Positives and False negatives to remove the noise of the issues found.
- Timeout validation - Functions have a limited time span and will be stopped if they exceed a time frame, which can cause issues for them.. Static scanning controls should identify and rectify issues in code like code portions that may prevent the code from being preempted abruptly.
- Easy integration with development environments and IDEs (e.g.,Visual Studio, Eclipse, IntelliJ.) Developers use the tools to develop functions, while easy integration provides developers with early warnings of issues in their code to fix them.
- Support for **early scanning and periodic or on-demand rescanning** in case new issues are found. This is essential as new vulnerabilities in components of code are found regularly
- Open Source issues - Open source components and an inventory of all open source libraries in accounts need to be done in code scanning. This will help fix issues when needed.
- Broad support of programming languages - need to support both compiled and interpreted languages. Developers of functions work in different languages, and varied binding is essential for static scanning.
- Access to source code and binaries - Integration with source code tools like git, svn, GitHub, bitbucket
- The simplicity of integration in SDLC and **CI/CD pipelines** with tools like Jenkins, TeamCity, Bamboo Maven, Ant, etc. This is the best place to check for dynamic bindings of software.
- Easy integration with bug tracking systems like Jira and ServiceNow. This provides for easy integration of results when code is scanned.
- Ability to control the kinds of checks the tools perform; BufferOverflow, SQL Injection flaws, Insecure deserialization, etc.

Config Template scans

When functions are deployed, it is not done automatically but through Infrastructure-as-Code (IaC) templates. These templates like CloudFormation, Terraform, can be scanned to make sure the functions are securely deployed without configuration issues which results in a compromise in security.

The templates can also look at the input and output of the functions, the way the functions are stitched, the ports open and compare this against known bad behavior and help remediate the issues. When the input is open to the internet, strict checks need to be applied to the code and risk highlighted.

Policy enforcement

Scanning code templates is critical for Serverless security. Code scan and the related policies can be enforced at different stages. When this happens, there are two workflows:

- Forcing policies in the CI/CD pipeline will be successful with easy enablement of developers to function code scanning tools. This helps developers uncover and fix issues faster; however, security teams in many cases do not know of CI/CD pipelines nor can control them -- a good first step but does not guarantee that all functions deployed are scanned.
- Deployment controllers - Deployment controllers are CloudFormation controllers, Knative controllers, Terraform controllers. All code that needs to get pushed goes through these controllers. Security teams can enforce policies in this place. Where all functions can be checked independently of where the CI/CD pipelines run.

Policies can be in enforce mode (e.g. a policy that disallows any user from accessing a function), where they prevent anything that does not meet the policy from being deployed. Monitoring mode is where alerts are raised but the functions can still be run. This could lead to incidents being raised andresult in tickets in CI/CD tools like Jira.

Different policies are necessary for various environments like Dev, Test, pre-production, and production environments. The production environments may have stricter controls compared to other environments and clear segregation from other environments.

Identity and access management controls for the functions and services that access the functions may be applied.

As security teams have limited control over Serverless functions, one of the most critical things in **Serverless** security is understanding and managing IAM permissions. There are two parts to the IAM:

a. IAM of the User or Service/ role used to create, deploy and delete the functions
b. IAM role/ permissions of the deployed Function/ Container Service itself

Managing IAM permissions is hard in very dynamic and fast-changing environments in which new functions are created, deleted, and deployed. The following should be performed:

a. The first step to security for **FaaS** is the visibility of the IAM roles and permissions (due to rapid speed of change), each function has and the permissions needed to deploy the Function. You cannot control what you cannot see. Functions need to use the least privilege model for security.
b. The next step is to create a guardrail of IAM permissions, the set of permissions allowed within an environment to a function. Any function deployed should be set up with that permission. Any function with an IAM permission higher than that should be prevented, and is taken through an exception path.

As defined above in the **Policy Enforcement** section, functionality can be achieved by putting controls in deployment pipelines as business logic is deployed or in the controller deploying the functions.

Gateway and Interface controls

External interfaces (APIs) in code increase the attack surface. All-access to the FaaS and Container Image-based Serverless environment externally is through Gateways. This makes the gateway critical for the security of the environment.

To protect external access, the gateway should support controls and remediation for "OWASP top 10" risks. *See section 6.3.7.2 API Security (OWASP top 10) for details.*

Data protection and integration with encryption/KMS services to protect data at rest and in transit.

With the data protection regulations, especially in highly regulated industries (GDPR), FaaS can leverage encryption and KMS to protect the data at rest and in transit. If we look at the shared responsibility model, data and its accountability is a customer's responsibility.

Function orchestration controls and validation,

'Step functions' help orchestrate function chaining. Security controls are needed to ensure that if one step fails, the whole chain fails.

Detection and response

- Failure detection
- Policy violation detection
- threat/Compromise detection
- Automate response
- Runtime detection and policy enforcement

CSP's capabilities vary significantly, with some third-party vendors providing serverless failure detection. One good practice would be to assess the CSP's native capabilities and see what else would bring added value from the third-party world.

6.4 Delta/Additional Controls for Container Image Based Serverless

Cloud-native computation is a highly complex and continually evolving construct. Without core components to make compute resource utilization occur, organizations cannot ensure workloads are secure. Considering that containers make it easy for developers to get the most use out of servers by enabling them to deploy multi-tenant applications on a shared host, spinning up and shutting down individual containers as per the need. Tobe able to support these needs, the developers need the right environment for running containers.

Since containers provide software-based virtualization, it is important to use a container-specific operating system which is a read-only OS with other services disabled, this helps reduce the attack surface. This also provides isolation and resource confinement that enables developers to run sandboxed applications on a shared host kernel.

In the Container Image-based Serverless platforms where tenants are provided the choice to bring their own OS images, or private Container Image-based Serverless platforms, it is pertinent to ensure the operating system is secure, follows security configurations, uses minimalistic OS configurations, and secure system calls from abuse by using Seccomp. Seccomp stands for secure computing mode and has been a feature of the Linux kernel since version 2.6.12. It can be used to sandbox the privileges of a process, restricting the calls it is able to make from user space into the kernel. Orchestration engines let the user automatically apply seccomp profiles loaded onto a Node to respective Pods and containers. Similarly, on Windows it is important to Sandbox Syscall and Filtering Kernel Isolation in Windows Per-Process, Limit and Blacklist Targets in Win32k Per-Process and apply Windows file system filters.

6.4.1 Managing API Access to Access Container Image-based Serverless Service

Kubernetes is an open-source orchestrator for deploying containerized applications. As Kubernetes is entirely API-driven, controlling and limiting who can access the cluster and what actions they can perform is the first line of defense. A Kubernetes cluster provides an orchestration API that enables applications to be defined and deployed with simple declarative syntax. Kubernetes API exposes concepts like Deployments that make it easier to perform zero-downtime updates of your software and Service load balancers that make it easy to spread traffic across a number of replicas of your service. Additionally, Kubernetes provides tools for naming and discovery of services so that you can build loosely coupled microservice architectures. Kubernetes expects all API communication to be encrypted by default with TLS. All API clients must be authenticated, even those that are part of the infrastructure like nodes, proxies, the scheduler, and volume plugins. These clients are typically service accounts or use x509 client certificates, and they are created automatically at cluster startup or are set up as part of the cluster installation. Once authenticated, every API call is also expected to pass an authorization check.

Kubernetes Control plane components description can be found in the Kubernetes.io website.

6.4.2 Container Image-based Serverless Configuration and Policy Enforcement

Kubernetes Control Plane:

The main components of the control plane include the Kubernetes API server, which provides a REST API for controlling Kubernetes. A user with full permissions on this API has equal root access on every machine in the cluster. The kubectl, which is a command-line/client for this API, can be used to make requests of the API server to manage resources and workloads. A User who has write-access to this Kubernetes API can control the cluster as a root user. Any ports on which the APIs server listens for requests must be closed, and appropriate authentication and authorization of users should be in place. See details in section 6.3.5 Access management controls. Kubernetes role-based access controls are used for configuration of flexible authorization policies for managing permissions around Kubernetes resources.

The Kubernetes API endpoint should not be exposed to the internet. The endpoint should only be accessible within the network that the Kubernetes cluster is deployed on. The Kubernetes API server should be configured to scale based on the number of requests to ensure that the cluster can handle large spikes in traffic.

The kubelet is the agent on each cluster node responsible for interacting with the container runtime to launch pods and report nodes and pod status and metrics. Kubelets in the cluster also have an API; where other components integrate and provide instructions such as start and stop pods, etc. Suppose unauthorized users get access to this API on any node to execute code on the cluster. In that case, it is possible to gain control of the entire cluster. It is important to restrict access to Kubelet API and Limit the permissions of kubelets by enforcing NodeRestriction in the --admission control settings[1] on the API. This helps restrict kubelet so that it can modify only pods that are associated with it. Kubelets also need client certificates to be able to communicate with the API server. These certificates should be rotated periodically and may be set to rotate automatically.

Kubernetes stores configuration and state information in a distributed key-value store called etcd. Any user that can write to etcd can effectively control the Kubernetes cluster. Even just reading the contents of etcd could easily provide helpful hints to a would-be attacker. All Kubernetes Secrets should be encrypted prior to being stored in ETCD. Secrets should not rely on ETCD encryption at rest for security. The Kubernetes API should be deployed in a highly available and fault-tolerant deployment. A highly available deployment fulfills the availability requirement of information security, ensuring that the cluster can be administered during an outage or attack/incident, and quality of service can be maintained. The ETCD nodes should be configured with security to only accept traffic from the API Server on port 2379. The ETCD nodes should also be configured to only accept traffic from other ETCD nodes on port 2380. The ETCD nodes should be deployed behind a highly available load balancer. This will ensure that traffic is evenly balanced across nodes and so the ETCD nodes are not directly exposed to the API Server. The ETCD nodes should be deployed in an odd number so that they can properly elect a leader in the cluster. This is required for high availability. The ETCD nodes should be deployed to separate instances than the API Server and other Kubernetes components. The Kubernetes Audit, Authenticator, API, Controller Manager, and Scheduler logs

[1] http://bit.ly/2IgwP7G

should be stored and monitored for issues. Common issues are repeated authentication failures. Basic authentication using static passwords should be disabled. The Kubernetes cluster should be integrated with the enterprise identity and access management. Kubernetes Hosts Should Not Have Public IP Addresses. Kubernetes Hosts Should be Running on an Approved operating system. [Kubernetes, 2021]

Data plane

Pod Security Policies should be defined at the cluster level and consumed by all pods in the cluster. By default, pods should not be allowed to run as privileged users and should not escalate permissions. Kubernetes worker nodes should not have public IP addresses assigned and should only serve traffic behind a load balancer or proxy server.Pods are dynamically provisioned least privilege credentials based on the role. These credentials should be generated by a Kubernetes webhook that is integrated with the backend provider. Pods should not inherit the credentials of the underlying host. Pods should have no privileges on the underlying cloud provider by default unless explicitly assigned by a Service Account.

All pods should be assigned a Service Account or default Service Account with the least privilege by default.

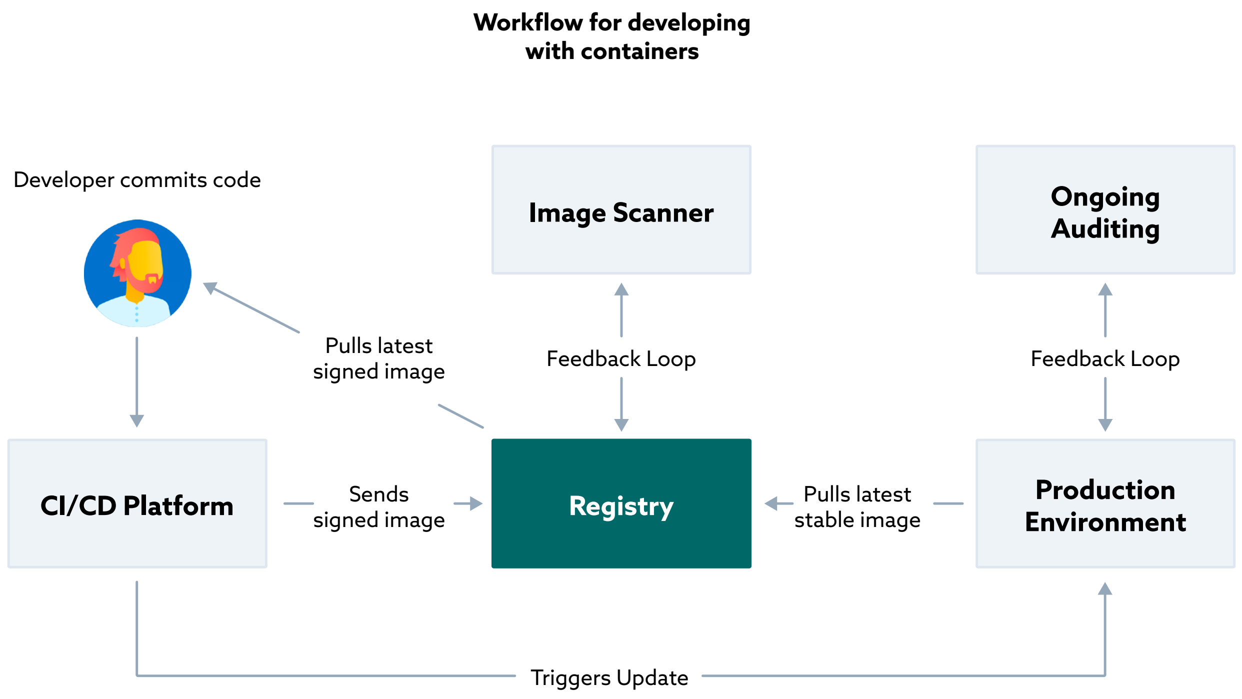
Securing pods and the containers that run as part of pods is a critical aspect of protecting container environments. These are individual units of compute that can be subject to attacks that may be used to attack Kubernetes clusters. For defense in depth, applying security at the lowest component unit level ensures greater fine-grained controls implemented on an individual application component. Kubernetes and other container orchestration systems provide capabilities that enable users to harden and secure pods e.g., Kubernetes security context and security policies such as Pod Security Policies.

There are other open-source tools such as, Open Policy Agent (OPA), which can also enforce security policies.

There may be specific requirements where deviation from those policies is needed. Hence, it is essential to apply security policies to all containers within a given pod, so a security context for individual containers can be specified. Hence the security context field must be included in the container manifest; the constraints for an individual container will override those specified for the pod when there is overlap or conflict.

6.4.3 Base Image Mgmt and Hardening

Base images



Base images are the foundation for all images and could be considered equivalent to base-level operating systems. Base images could be owned by cloud development teams, or the operating system team within the organization. These images are pulled from various official repositories (Docker, Quay etc.) and stored in the organization's image registry/repository which should be made available to engineering teams as approved and authorized versions for use within the organization. As per NIST 800-190, organizations should use minimalistic container-specific host OSs, with all other services and functionality disabled whenever possible. The organizations should harden these images (via patching and hardening) based on industry best practices to ensure that only the ports, protocols, and services necessary to meet business needs are provided based upon hardening benchmarks (Centre for Internet Security - CIS Benchmarks).

Use a CI/CD pipeline to automatically build, tag, and test the base image. Supplement this by requiring that the development, testing, and security teams sign images before deploying them into production.

A few Base images Security hardening best practices:

Remove or disable default Accounts	Default account names and passwords are commonly known in the attacker community.
Configure OS User Authentication	The base images should be configured to authenticate a prospective user/ service and should be used with encryption. MFA should be used whenever possible. Organizations should implement TLS to protect sensitive authentication information during transmission over untrusted networks.
Limit access permissions	Configure RBAC and ABAC access controls and ensure that all users have the necessary privileges to carry out their tasks—no less and no more.
Repudiation & Integrity	Container images (via an image manifest) should be protected from modification using digital signatures securely stored in a notary.
Configure Image registry	Upstream versions of base images should undergo security hardening and then should then be made available to tenant engineering teams as approved and authorized versions for use within the organization.
Address Image vulnerabilities	Patches and other mitigating controls should be deployed as soon as possible (based on organizations' business/compliance needs) whenever new security vulnerabilities are discovered and include immediate testing for the vulnerability to confirm the risk has been addressed.
Securing the Software Supply chain	An important aspect of the deployment pipeline is ensuring that the deployments follow the organization's approved processes. A few methods for establishing this are the ones with Binary Authorization, tags, attestations, etc. Tagging the container images with the corresponding commit hash makes it easy to trace back the image to a specific point in history. Attestors can state whether an image meets a particular criterion by adding notes about images in Container Registry at each of the critical stages of your deployment pipeline. With binary authorization, a policy can be configured that requires specific attestations on the image before the image can be deployed across clusters. Images can be undeployed if the defined policy is not satisfied.

6.4.4 Kubernetes configurations and service mesh policies enforcement

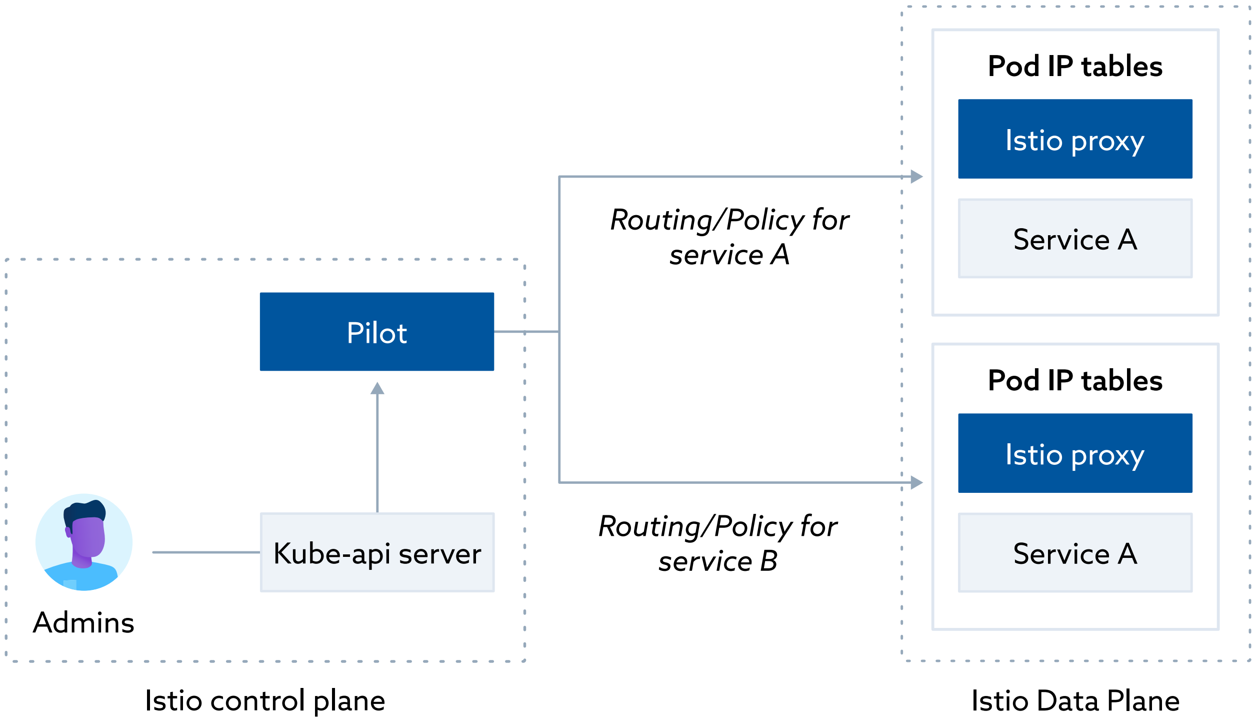
Kubernetes (K8s) is an open-source orchestration platform that automates the deployment, scaling and management of containerized applications. The Kubernetes control plane stores the state and configuration data for the entire cluster in ectd, a persistent and distributed key-value data store. Each node has access to ectd, and through it, nodes learn how to maintain the configurations of the containers they are running.

K8s control plane communicates with the components in the cluster through the kube-Apiserver. It ensures that configurations in etcd match with configurations of containers running in the cluster. This is carried out via declarative API. However, K8s does not specify a standard (or mandate) for configuration language/system (JSON, YAML).

The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end-users, different parts of your cluster, and external components communicate with one another. The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example, Pods, Namespaces, ConfigMaps, and Events). Most operations can be performed through the kubectl command-line interface, which uses the API via REST calls. Network Policy is a Kubernetes feature to configure how groups of pods are allowed to communicate with each other and other network endpoints. Network Policy operates at Layers 3 (Network) and 4 (Transport) in the OSI model. Kubernetes network policy specifies how groups of Kubernetes workloads, which are hereafter referred to as pods, are allowed to communicate with each other and other network endpoints. Network policy resources use labels to select pods and define rules which specify what traffic is allowed to the selected pods.

Pod IP addresses are not durable and will appear and disappear in response to scaling up or down, application crashes, or Node reboots. Services were built into Kubernetes to address this problem. A Kubernetes Service manages the state of a set of Pods, allowing you to track a set of Pod IP addresses that are dynamically changing over time. Services act as an abstraction over Pods and assign a single virtual IP address to a group of Pod IP addresses. Any traffic addressed to the virtual IP of the Service will be routed to the set of Pods that are associated with the virtual IP. This allows the set of Pods associated with a Service to change at any time — clients only need to know the Service's virtual IP, which does not change.

Istio service mesh is a security as a service layer that offers policy-based routing and policy-based authorization for supported protocols. Istio policy operates at the "service" (OSI Layer 7). The service mesh will inject the proxy and node agent container into the application's deployment specification, sharing the pod's network namespace and transparently intercepting traffic to apply policy. The node agent will use the pod's service account token to acquire platform certificates for the proxy and serve the certificate/key to the proxy using the proxy's secret discovery service (SDS). For service mesh enabled pods, all TCP Traffic into and out of the pod will be redirected through the envoy proxy running as a sidecar container in the pod and can be extended to support external services as shown in the diagram below.



Kubernetes Policy Enforcement Using Istio

Service-to-Service Inside Istio:
Both services are inside the service mesh in a single cluster and are deployed as Kubernetes native services and the istio control plane can automatically discover their definitions and configure the proxies appropriately.

External Service to Istio:
This represents a client (e.g., curl) or service outside the mesh connecting to a service inside the mesh (shown as SvcA-to-SvcB in the diagram above.) For this communication to work, the client needs a client certificate, and SvcB must also be integrated with the Istio ingress gateway

Istio to External Service:
In this case, a client inside the mesh needs to initiate connections to an external service shown as SvcC-to-SvcD in the diagram.

In addition to the Istio Proxy/Envoy sidecar, application pods could include an OPA sidecar as suggested in the Github blog. When Istio Proxy receives API requests destined for your microservice, it checks with OPA to decide if the request should be allowed.

6.4.5 Access Management Controls

The Apiserver is configured to listen for remote connections on a secure HTTPS port (443) with one or more forms of client authentication enabled. One or more forms of authorization should also be enabled.

Authentication/Authorization

All communications in the control plane are via mutually authenticated TLS. A firewall rule is configured to allow external HTTPS access to the API. Users access the API using kubectl, client libraries, or by making REST requests. Both human users and Kubernetes service accounts can be authorized for API access. When a request lands at the API server, it performs a series of checks to determine whether to serve the request or not and, if it does serve the request, whether to further validate via **defined policy**. Authorization is usually implemented by the K8s RBAC authorization module. However, they can be implemented via alternate methods such as advanced authorization policies via Open Policy Agent (OPA) by leveraging the Webhook authorization module.

Admission controller/Pod Security Policy

Once an API request has been **authenticated and authorized**, the resource object can be subject to validation or mutation before it's persisted to the cluster's state database, using admission controllers. An admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to the persistence of the object, but after the request is authenticated and authorized. Some Admission Controllers are provided below:

- **DenyEscalatingExec** — if it's necessary to allow your pods to run with enhanced privileges (e.g., using the host's IPC/PID namespaces), this admission controller will prevent users from executing commands in the pod's privileged containers.
- **PodSecurityPolicy** — provides the means for applying various security mechanisms for all created pods. The PodSecurityPolicy objects define a set of conditions that a pod must run with to be accepted into the system, and defaults for the related fields.
- **NodeRestriction** — an admission controller that governs the access a kubelet has to cluster resources.
- **ImagePolicyWebhook** — allows for the images defined for a pod's containers, to be checked for vulnerabilities by an external 'image validator.'

Open Policy Agent Gatekeeper (CRD)

Kubernetes allows decoupling policy decisions from the inner workings of the API Server by means of admission controller webhooks, which are executed whenever a resource is created, updated, or deleted. Gatekeeper is a validating webhook that enforces CRD-based policies executed by Open Policy Agent, a policy engine. Additionally, Gatekeeper's audit functionality allows administrators to see what resources are currently violating any given policy. Finally, Gatekeeper's engine allows administrators to detect and reject non-compliant commits to an infrastructure-as-code system's source of truth, further strengthening compliance efforts.

Istio Service Mesh - Authentication

Istio supports mutual TLS authentication between services where the client and server proxies exchange certificates. The certificates are verified against the CA by the proxy, and the Secure Production Identity Framework for Everyone (SPIFFE) URI from the cert is extracted and used as the identity of the peer. This standard authentication mechanism allows us to interoperate with clients and servers outside of the service mesh integrated with the CA. Enabling mutual TLS can be configured at the cluster, namespace, and service level. An authentication policy can be used to enforce client certificate authentication. Istio supports additional client authentication methods as well (e.g., JWT)

Istio Service Mesh - Authorization

Istio's authorization feature provides namespace-level, service-level, and method-level access control for services enabled with service mesh. It features:

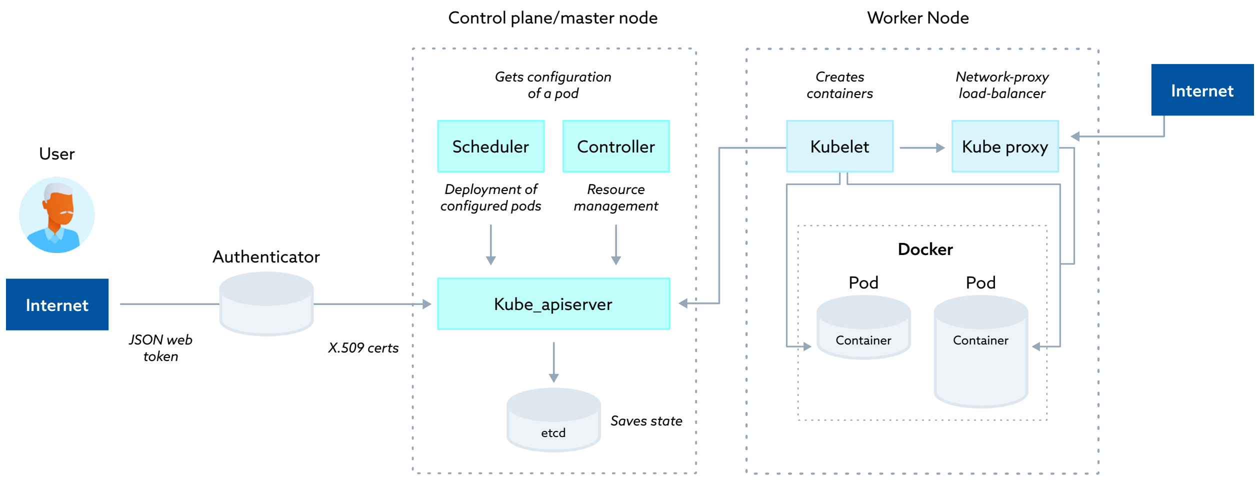
- Role-Based semantics, which are simple and easy to use
- High-performance operation, as authorization is enforced natively in the local envoy proxy
- Support for HTTP and GRPC protocol attributes and plain TCP connections

Namespace administrators can configure authorization policies for services in their namespace. The policies are configured as k8s custom resource definitions (CRDs) using the k8s API. Like any other k8s CRD, the authorization policies are namespaced, meaning the service admin must have write access to the policy CRDs in the service's k8s namespace to affect the service's authorization policy.

When authorization is enabled for a service, all of the inbound requests to a service will fail unless it is permitted by a policy (*who* can do *what* under *which conditions*).

6.4.6 Kubernetes Risks & Controls:

Example: API server access



The API server is the hub of all communication within the cluster. It is on the API server where the majority of the cluster's security configuration is applied. Suppose somebody or something gains unsolicited access to the API. In that case, it may be possible for them to acquire all kinds of sensitive information and gain control of the cluster itself. Carefully managing access to the API server is crucial as far as security is concerned.

The API server controls access to all workloads on K8s clusters and the resources they use. It is on the API server where the majority of the cluster's security configuration is applied. If a bad actor gains unauthorized access to a privileged resource via K8s API server, they can access any cluster resource which is secured by k8s RBAC (e.g. Secrets), access any file on cluster nodes, etc. Carefully managing access to privileged Resources is therefore crucial to maintain the integrity of a cluster. One of the most basic needs of a new application that's deployed on Kubernetes is to expose a couple of web endpoints on the public internet with different URLs and secure them with SSL. It is important to manage the endpoint security for this.

The list below contains examples of exploits that can happen if access to API server is not protected:

- A Denial of Service attack (DOS) is when the victim service is overloaded with many fake requests. As a result, the service is no longer capable of responding to legitimate requests. In extreme cases, the attack leads to depleting the system resources of the entire machine (CPU, memory, network bandwidth, disk I/O, etc.), causing even more damage.
- Netflix announced eight vulnerabilities in their advisory that affect HTTP/2 implementations; two of them affect the Go net/http library: CVE-2019-9512 "Ping Flood" and CVE-2019-9514 "Reset Flood." Any application written in Go that uses the net/http package to listen for HTTP/2 requests is vulnerable to DoS attacks, including Kubernetes.
- CVE-2019-9512 "Ping Flood": the attacker hammers the HTTP/2 listener with a continuous flow of ping requests. The recipient - to respond to each request - starts queuing the responses one after the other. That queue could grow, allocating more memory and CPU until the application crashes.
- CVE-2019-9514 "Reset Flood": this attack has a similar theme as the first, except that it exploits the RST_STREAM frame of HTTP/2. RST_STREAM is simply a frame type that - when sent from a peer - signals to the other peer that the connection needs to be canceled. So, a DoS attack can be crafted by opening several streams to the server and sending invalid data through them. Having received invalid data, the server sends RST_STREAM frames to the attacker to cancel the "invalid" connection. With lots of RST_STREAM responses, they start to queue. As the queue gets more massive, more and more CPU and memory get allocated to the application until it crashes.
- CVE-2018-1002105 - incorrect handling of error responses to proxied upgrade requests in the kube-apiserver.
- CVE-2016-7054 (OpenSSL advisory) [High severity] - TLS connections using *-CHACHA20-POLY1305 ciphersuites are susceptible to a DoS attack by corrupting larger payloads.
- According to our 2018-2019 Global Application & Network Security report, HTTPS flood attacks or DDoS attacks that exploit SSL/TLS were the most commonly reported form of application-layer attacks in 2018. By using SSL/TLS, we enjoy authenticity, integrity and confidentiality. However, when it comes to the destination server, the SSL/TLS connection requires large amounts of allocated resources – 15 times more than from the requesting host, to be exact.

Example : Misconfiguration

Kubernetes is a complex system that offers multiple configuration options. These configurations should be carried out securely. Security misconfigurations can expose sensitive user data and system details that may lead to full server compromise. Security misconfigurations are commonly a result of insecure default configurations, incomplete or ad-hoc configurations, open cloud storage, misconfigured HTTP headers, unnecessary HTTP methods etc. Attackers will take advantage of these misconfigurations to exploit these unpatched flaws, common endpoints, or unprotected files and directories to gain unauthorized access.

Mitigation strategies:

- Establish repeatable hardening and patching processes.
- Disable unnecessary features.
- Restrict administrative access.
- Define and enforce all outputs, including errors.
- Ensure the host is secure and configured correctly. Check your configuration against CIS Benchmarks.
- Autochecker assesses conformance with these standards automatically.
- Establish a repeatable hardening process leading to the fast and easy deployment of a properly locked-down environment.
- Kubernetes stores configuration and state information in a distributed key-value store called etcd. Any user that can write to etcd can effectively control the Kubernetes cluster.
- Review and update configurations across the entire API stack. The review should include-orchestration files, API components, and cloud services.
- Continuously assess the effectiveness of the configuration (configuration flaws) in settings in all environments via an automated process.
- Configure custom dashboards and alerts, enabling suspicious activities to be detected and responded to earlier.
- Continuously monitor the infrastructure, network, and API functioning (possibly via automation).

Example: Vulnerability scanning

Scanning an image for vulnerabilities throughout its lifecycle is crucial. It is also important for weighing your organization's risk tolerance against maintaining the velocity. An organization will need to generate its own policies and procedures for handling image security and vulnerability management. Deploying container images with vulnerabilities can lead to attacks carried out by threat vectors looking to exploit these vulnerabilities.

Best practices:

Start by defining your criteria for what constitutes an unsafe image, using metrics such as:

- vulnerability severity
- number of vulnerabilities
- whether those vulnerabilities have patches or fixes available
- whether the vulnerabilities impact misconfigured deployments

6.4.7 Additional Security

6.4.7.1 Kubernetes Security Best Practices

Kubernetes is an open-source container orchestration engine for automating deployment, scaling, and management of containerized applications. Kubernetes K8s cluster consists of worker nodes/ pods that host applications. Kubernetes control plane manages the pod network in the cluster. I am covering the Kubernetes and container security best practices in the four sections listed below:

Part One

TLS everywhere
TLS should be enabled for every component that supports it to prevent traffic sniffing and to authenticate the identity of both sides of a connection.

Run a service Mesh
A service mesh is a web of encrypted persistent connections made between high-performance "sidecar" proxy servers, Envoy. It adds traffic management, monitoring, and policy - all without microservice changes.

Use Network policies
By default, Kubernetes networking allows all pod-to-pod traffic; this can be restricted using a Network Policy.

Use Open Policy Agent (OPA)
With OPA you can enforce custom policies on Kubernetes objects without recompiling or reconfiguring the Kubernetes API server.

Logging & Monitoring
You need this to detect anomalies on the application and infrastructure levels. If there are any attacks or anomalies – high usage or potential compromises – logging and monitoring will detect them. Application performance monitoring will detect other breaches such as DDOS attacks.

Consider using a Bastion host
This is a special-purpose computer on a network specifically designed and configured to withstand attacks. Access to masters should only be done through bastion hosts, which can be hardened and can monitor users who have privileged access to Kubernetes.

Private networking
Both Kubernetes master and worker nodes need to be deployed on private subnets to ensure secure connectivity with corporate networks, prevent direct reachability from the Internet and reduce the overall attack surface.

Use Linux Security Features
The Linux kernel has several security extensions (SELinux) that can be configured to provide the least privilege to applications. The Linux kernel has many overlapping security extensions (capabilities, SELinux, AppArmor, seccomp-bpf) that can be configured to provide less privilege to applications.

Cluster node images
When you are building in a Kubernetes cluster, you will be using a Linux image. It needs to be CIS benchmarked to make sure all the Linux security controls are in place. Not doing the operating system hardening process can open up the infrastructure to software vulnerabilities. It is also important to consider the entire Software supply chain security to ensure that the software chain of custody is being followed according to security best practices. Binary Authorization, tags, attestations, etc. are examples of how this can be implemented.

Separate and Firewall your etcd cluster
etcd stores information on state and secrets and is a critical Kubernetes component - it should be protected differently from the rest of your cluster.

Rotate Encryption keys
A security best practice is to regularly rotate encryption keys and certificates to limit the "blast radius" of a key compromise.

Part Two: Container Security

Use PodSecurityPolicies
Policies are a vital but often overlooked piece of security that simultaneously functions as a validating and mutating controller. PodSecurityPolicy is an optional admission controller enabled by default through the API; thus, policies can deploy without the PSP admission plugin enabled.

Statically analyze YAML
Where PodSecurityPolicies deny access to the API server, static analysis should be used in the development workflow to model an organization's compliance requirements or risk appetite.

Run containers as a Non root user
Containers that run as the root frequently have far more permissions than their workload requires which, in case of compromise, could help an attacker further their attack.

Part Three: Automation Security

Image Scanning
All the deployments should be controlled through an automated CI/CD environment. At a high level, everything in Kubernetes is deployed as a container image. When someone creates an application, it becomes a container image and gets added to the container registry. The container image scanner needs to be part of the CI/CD pipeline. When someone creates a container image, it needs to be continuously scanned for vulnerabilities. You can do image whitelisting through an admission controller in Kubernetes. If your application uses certain images, those need to be approved.

Secret management
Your clusters also need to be integrated via secret management systems. This ensures application pods will automatically receive required passwords and secrets at runtime based on the AppRoles attached to the pods.

Code analysis
Code scanning and static code analysis are also integral parts of automation security. When working on any application code in Kubernetes, you should scan the source code to ensure it does not have any vulnerabilities or any hard-coded anomalies.

Third-party vulnerability scanning
Vulnerability assessments are a core requirement for organizations. This is especially necessary when using upstream components which could be vulnerable. Third-party vulnerability scanning (BlackDuck, Tenable, etc.), when conducted continuously, can help organizations stay ahead of threat agents. These standard vulnerability scanning tools scan open-source libraries for known vulnerabilities published in various vulnerability databases (such as National Vulnerability Database).

Once the vulnerabilities are identified, the remediation of these vulnerabilities will depend on how these might impact an organization's operations, business, regulatory requirements, or reputation. Example: PCI DSS 3.2.1 requires that known vulnerabilities of high/critical type should be remediated within 30 days.

DevSecOps (CI/CD)
Security should be built into the entire DevSecOps process. The Agile process that feeds DevSecOps must also be secure and security user stories must be in the backlog. Embed security throughout the software lifecycle to identify vulnerabilities earlier, perform faster fixes, therefore, reduce cost. Continuous monitoring to ensure devices, tools, accounts, etc., are continuously discovered and validated.

Part Four: Identity and Access Management

Enable RBAC with least privilege
Role-based access control (RBAC) provides fine-grained policy management for user access to resources, such as access to namespaces. Centralizing authentication and authorization across an organization (aka Single Sign On) helps onboarding, offboarding, and consistent permissions for users.

Use Third-party Auth for API server
Kubernetes - controlling access: To secure and manage access to Kubernetes APIs, administrators need to create detailed authentication and authorization policies and implement advanced, full-featured screening technologies.

6.4.7.2 API Security (OWASP top 10)

APIs are critical to automating container management at scale. APIs are used to:

> - Validate and configure the data for pods, services, and replication controllers.
> - Perform validation on incoming requests & invoke triggers on other components.

API threats are considered the most common attack vector. Threat agents exploit these vulnerabilities in the APIs to breach the applications. Please refer to OWASP Top 10 threats and mitigations

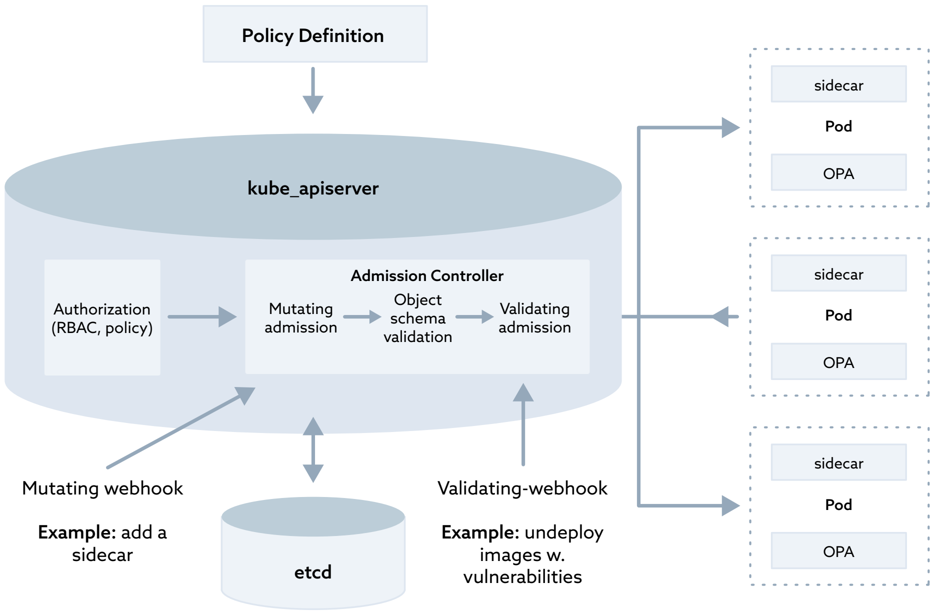
6.4.7.3 Kubernetes Policy

Kubernetes allows decoupling policy decisions from the inner workings of the API Server by means of Admission Controller Webhooks. Admission webhooks are HTTP callbacks that receive admission requests and do something with them. There are two types of admission webhooks, validating admission webhook and mutating admission webhook. Mutating admission webhooks are invoked first and can modify objects sent to the API server to enforce custom defaults. After all object modifications are complete, and after the incoming object is validated by the API server, validating admission webhooks are invoked and can reject requests to enforce custom policies.

Open Policy Agent (OPA) is an open-source, general-purpose policy engine that unifies policy enforcement across the stack. It provides a high-level declarative language that allows you to specify policy as code and use simple APIs to offload policy decision-making from your software. OPA enables you to enforce policies in microservices, Kubernetes, CI/CD pipelines, API gateways, and more. Gatekeeper is a validating (mutating) webhook that enforces CRD-based policies executed by Open Policy Agent. Gatekeeper's audit functionality allows administrators to see what resources are currently violating any given policy.

Example of validating webhooks (LINK):

- Disallow running of privileged containers
- Disallow shared usage of host namespaces
- Restrict all usage of host networking and ports
- Restrict any usage of the host filesystem
- Restrict Linux capabilities to the default set
- Restrict usage of defined volume types
- Privilege escalation to root
- Restrict the user and group IDs of the container
- Restrict allocating an FSGroup that owns the pod's volumes
- Requires seccomp profile



Kubernetes policy best practices:

- All network connections should be subject to enforcement via policies
- Establishing the identity of a remote endpoint is always based on multiple criteria including strong cryptographic proofs of identity.
- Network-level identifiers like IP address and port are not sufficient on their own as they can be spoofed by a hostile network.
- Compromised workloads must not be able to circumvent policy enforcement.
- Use encryption to prevent disclosure of data to entities snooping network traffic.
- Start by applying the "default-deny-all" network policy. Only connections explicitly whitelisted by other network policies should be allowed.
- Network policies are namespaced and are created for each namespace.
- In order to receive traffic from outside sources, designate labels applied on pods to allow access from the internet and to create network policies that target those labels.
- For a more locked-down set of policies, you would ideally want to specify more fine-grained CIDR blocks as well as explicitly list out allowed ports and protocols.

6.5 Compliance and Governance

When we leverage the Cloud computing model, the organization is able to not only reduce investments in hardware, facilities, utilities, and data centers, but in theory transferring this risk to the CSP should reduce overall risk. There is a broad assumption being that the Cloud provider is making ongoing investments in platform security and managing those areas of risk, but how do we know that? Ongoing and regular assessment of the cloud provider's performance and quality of service is an essential part of the organization's security assurance program. The organization's sourcing team should regularly analyze and assess the state of the cloud provider to ensure that contractual obligations are being met.

The security team's focus should be directed to identifying risk themes related to Serverless applications being deployed by development teams. An analysis of the risks or risk themes related to Serverless computing can be examined and used to develop a better understanding of overall risk. We can then form risk statements that can be used to determine the likely harm resulting from a particular risk and begin an assessment of what to do about the risk. Any residual risks can then be recorded in the organization's GRC system.

It is hard to protect and govern an environment without having a view of all the assets.

6.5.1 Asset Management for Serverless

The asset management system requires a mechanism designed to handle the ephemeral nature of Cloud compute. The ephemeral nature of Cloud compute creates a challenge in asset management and incident response. One of the best approaches is to integrate automated asset tracking within your CI/CD pipeline. This way, any time a new serverless function is deployed or changes, the corresponding change can be recorded, updated, and the asset tagged accordingly. Monitoring, alerting, and audit actions and changes to the environment are updated in real-time. This information can then be integrated with systems to automatically respond and take action.

This way, gaps in visibility are closed, and confusion or incorrect response during an actual incident can be avoided. Alternatively, organizations can also adopt a polling approach towards gathering asset inventory. In serverless, APIs activated for the organization are polled a certain number of times per day. The information is updated in the organization's asset inventory by taking delta of two consecutive polling runs.

When we leverage the Cloud computing model the organization can reduce investments in hardware, facilities, utilities, and data centers, and in theory, transferring this risk to the CSP should reduce overall risk. There is a broad assumption that the Cloud provider is making ongoing investments in platform security and managing those areas of risk, but how do we know that? Ongoing and regular assessment of the cloud provider's performance and quality of service is an essential part of the organization's security assurance program. The organization's sourcing team should regularly analyze and assess the state of the cloud provider to ensure that contractual obligations are being met.

The security team's focus should be identifying risk themes related to Serverless applications deployed by development teams. An analysis of the risks or risk themes related to Serverless computing can be examined and used to better understand overall risk We can then form risk statements that can determine the likely harm resulting from a particular risk and begin an assessment of what to do about the risk. Any residual risks can then be recorded in the organization's GRC system.

Due to the nature of serverless computing, developers no longer have to worry about infrastructure, network, or host security. However, new attack vectors have emerged for the full list; refer to the Cloud Security Alliance's "Top 12 Risks for Serverless Applications".

6.5.2 Serverless Governance

From a governance point of view,function/container inventory should be mentioned somewhere in this document, as it's something unique to track that users have not had to do in the past, and requires new tools/thoughts.

Elements of Governance:

- Asset Inventory - Leveraging a combination of your CI/CD pipeline, CMDB, and asset metadata history maintained by the CSP to develop a complete view of assets that you need to protect
- Shared Responsibility - Customer or Application Owner & CSP - Stakeholders & Regulators Mandate
- RACI - Responsibility, accountability, Consulted, Informed
- Metrics - How we measure
- Automation
- Performance - Efficiency & effectiveness
- Service Agreement - SLA - TOR

1. Develop single-purpose functions that are stateless:
Since functions are stateless and persist for a limited duration only, writing single-purpose codes for a function is recommended. This limits the execution time of a function which has a direct impact on cost. In addition, single-purpose codes are easier to test, deploy and release, thus improving enterprise agility. Finally, even though statelessness may be perceived as a limitation, it provides infinite scalability to a platform for handling an increasing number of requests, which would not have been possible otherwise.

2. Design push-based, event-driven patterns:
Designing push-based and event-driven architecture patterns where a chain of events propagate without any user input imparts scalability to an architecture.

3. Incorporate appropriate security mechanisms across the technology stack:
Appropriate security mechanisms must be incorporated at the API Gateway layer and also at the FaaS layer. These security mechanisms include access controls, authentication, identity and access management, encryption and establishing trust relationships, etc.

4. Identify performance bottlenecks:
On-going measurement of performance bottlenecks in terms of identifying which functions are slowing down a particular service is critical to ensure the optimal customer experience.

5. Create thicker and powerful front-ends:
Executing more complex functionality at the front-end, especially through rich client-side application frameworks, helps reduce cost by minimizing function calls and execution times. Completely decoupling back-end logic from the front-end while not compromising on security is one way of doing it. This also allows more services to be accessed from the front-end resulting in better application performance and a richer user experience.

6. Leverage third-party services:
Serverless being an emerging field, existing enterprise tools for various services like logging, monitoring, etc. may not be compatible. Choosing the right third-party tools for executing the task at hand will be key for enterprises to ensure the benefits of serverless are utilized to the fullest.

6.5.3 Compliance

Perform continuous near-time monitoring to detect and record if any vulnerability drift or security or compliance drift occurred.

Organizations that don’t focus on compliance can be endangered by legal issues. The compliance process is often highly detailed, requiring extensive diagramming and documentation and explaining it to auditors and regulatory bodies on how the organizational systems are compliant. Because serverless applications remove the ability to identify a physical machine as evidence of security controls, this complicates regulatory compliance.

For example, SOX compliance:

Enforced software engineering practices and DevOps policies cover a large part of the SOX compliance. However, it is pertinent for organizations to have a strategy for dealing with personally identifiable information to achieve full compliance and demonstrate that to regulators. For example,for AWS Lambda functions, this means that significant enforcement of SOX will be in AWS configuration and development pipeline policies and controls to resources.

Similarly, for GDPR, vital elements of the regulation require proper security implementation, data access restrictions, change control management, data retention, and destruction policies -- all the items expected of a publicly traded company generally implemented as part of an effort for an organization to become SOX-compliant.

7. Futuristic Vision for Serverless Security

As we have seen, Serverless brings many benefits and challenges along. In this section, we will focus on areas that will become vital for security in the upcoming years in the realm of operations, applications security, and encryption/privacy. The serverless tools are still a challenge and are tied to CSP's as the technology matures. Deployments and configuration have improved over the last few years by the Cloud providers. However, the experience is still very fragmented as some cloud providers are more mature than others.

Full adoption will require the same level of maturity across the board for consumers to create multi-cloud strategies or a level of abstraction on top of the CSPs. The Cloud providers also need to bring the continuous delivery approach to serverless to inspect and mitigate security defects and provide operational best practices on deployment like AB testing, blue/green deployments, and canary releases built into the platform. As serverless has increased cardinality, it requires similar constructs as containers. Currently, the tooling to introspect those systems is still an open problem.

Alongside the operational challenge, there is the application's quality from a security perspective and its dependencies. Topics that focus on the application runtime, sandboxes, and AI/ML methods to identify and mitigate risk, will grow in the upcoming years as serverless gains adoption. Developers usually are not trained in security. As such, novel programming languages, more robust SDLC integration, and formal methods will enable developers to be agile with the proper guardrails.

Finally, serverless encryption and privacy aspects will require breakthroughs; although most CSP's adopted secure enclaves for specialized workloads, it also needs to be extended to serverless products. The privacy aspect and the ability to do operations on encrypted data due to the transient and boundless nature of serverless will require better privacy frameworks, and in some cases, the ability to work over encrypted data, which several frameworks from the major cloud providers (SEAL) have been published.

As serverless gets widespread, adoption operations will require and push the CSPs for more maturity and consistency across different environments. The application security lifecycle will be more streamlined and integrated with the Cloud providers. DevOps tools and supply chain integration, especially with the recent hacks that took advantage of these tools, will be a major area of focus. Finally, despite being the more far-reaching goal, the privacy aspect -- major cloud providers have been publishing R&D in this area, highlighting the roadmap and their focus.

7.1 The Foad for the Future of Serverless

7.1.1 Movement towards serverless Container Image-based Serverless

All enterprises today are embracing digital transformation for continued growth and to gaining competitive edge.

Containers are now broadly deployed, mainly using the Kubernetes orchestration system. Cloud-native technologies such as Kubernetes provide the automation, visibility and control necessary to manage applications at scale, and high innovation velocity.

To a casual observer, it may appear that the Kubernetes-led automation minimizes operational tasks in the deployment pipeline, and during the deployment and operations processes. A deeper look unveils why that is not so. While applications can be flexibly deployed using a Kubernetes configuration, understanding replicas, scale, taints, and tolerations are not top of mind for many developers. This leads to much unnecessary complexity for developers, which explains why the future of Containers is Serverless.

The Serverless model of Container Image-based Serverless abstracts a lot of the management of a lot of the functionality like scaling, replicas, managing the control plane to the Service provider. This enables developers to focus on their tasks.

As we move forward, we will see more and more Container Image based Serverless models providing greater flexibility in what a developer has to do, even as we move towards lesser and lesser overhead for developers.

7.1.2 Virtualization changes

Security is a big concern for Containers. Virtual machines provide very strong isolation on the host. Containers, however, use Linux namespace to separate, restrict and isolate resources from one container to another. This isolation can be compromised, and containers cannot act as security boundaries, just like VMs. The need for compromise between the capabilities of fast deployment, easy component assessment, and strong security is leading to many different models of OS:

Unikernels

Unikernels are non-general purpose, built using the library operating system, and use a single address space machine. The OS greatly reduces the attack surface and resource footprint.

Kata containers

Kata is a secure container runtime with lightweight virtual machines that feel and perform like containers, but provide stronger workload isolation using hardware virtualization technology as a second layer of defense.

gVisor

gVisor provides a virtualized environment to sandbox containers for greater security and isolation. The system interfaces normally implemented by the host kernel are moved into a distinct, per-sandbox application kernel to minimize the risk of a container escape exploit. gVisor does not introduce large fixed overheads and still retains a process-like model with respect to resource utilization.

7.1.3 FaaS evolution

As the adoption of FaaS grows, so does the need to provide greater flexibility, transparency, more robust security, and more controls for enterprises' deployments of their FaaS applications. FaaS is built on top of Container Image-based Serverless platforms.

As such, it allows FaaS to grow in different aspects such as:

Knative

As Kubernetes becomes the layer of abstraction, Knative is an open-source platform, which adds components for deploying, running, and managing serverless, cloud-native applications to Kubernetes. This provides the infrastructure administrators the ability to build on existing infrastructures while enabling the platform to be more flexible and transparent. This allows customers to run Functions on the private cloud too.

OpenWhisk

With the growth of FaaS, there is a need to support stateful FaaS. OpenWhisk provides a distributed Serverless platform that executes functions in response to events at any scale. OpenWhisk manages the infrastructure, servers, and scaling using Docker containers so you can focus on building amazing and efficient applications. OpenWhisk allows for Stateful Functions. This allows customers to run Functions on the private Cloud too.

OpenFaaS

Like Knative, OpenFaaS is a framework for building serverless functions with Docker and Kubernetes which has first class support for metrics. Any process can be packaged as a function enabling you to consume a range of web events without repetitive boiler-plate coding.

Similarly to OpenWhisk, this would also allow the functions to run Functions on private Cloud too.

Closed FaaS

There is a growing trend towards allowing the ability to run monitoring and security logic within the containers running the functions to enable enterprises to control their FaaS deployments, even for closed FaaS.

7.2 Serverless Security

Our understanding of how and when to use Serverless architectures is still in its infancy. We're starting to see patterns of recommended practice occur, and this knowledge will only grow." (Martin Fowler)

Above are the words of practical wisdom and challenges, as Serverless is still at an early stage and knowledge. Practices and Architecture patterns will keep growing in the next few years trying to learn the best approaches to better develop, deploy, monitor, deliver performance and more secure applications.

1. Serverless is left focused towards the developer - code first. There is change and challenge presently towards the support of languages across cloud providers. **We find solutions with the use of frameworks.** Some tasks required to configure and deploy a serverless application are too time-consuming to do without the help of a serverless framework. A developer gets no benefit from taking care of these tasks manually, so it makes total sense to take advantage of a framework. **Frameworks will continue to evolve. Enhancing support for languages, continuous development and integration, bundling and packaging code for deployment and the actual deployment process (which may take multiple steps in itself), setting environment variables, managing secrets, configuring endpoints to expose your serverless service as a public API, taking and properly setting the permissions needed by the function, etc.**

Presently, the ecosystem for serverless frameworks is quite rich and diverse. Some are focused on specific programming languages and/or cloud providers. Others are runtime and cloud-agnostic. It is better to shift to supporting more cloud-native, more support of developer languages, and becoming cloud-agnostic.

2. Standards are evolving, and effort is made towards the **harmonization of security controls** in serverless environments.

3. Logging and debugging - developers need to have control over the platform that is running code and need metrics that can be relied upon, generated by background processes or daemons. Code could be instrumented to send metrics to third-party services in real-time, but that means latency will impact the overall execution. Specific information is critical to log in to serverless applications, so that they are available when it comes the time to act on security breaches. Having critical logs will help, for example, understand which security flaws attackers explored and how to fix them, or build a blacklist of IP addresses, or identify compromised customer accounts.

To properly conduct logging and debugging of serverless applications we must rethink how we approach these activities. Access to information on desired parameters such as **Invocation/Event Inputs, Response payload, Authentication requests** and its integrity availability, open integration, and this needs to evolve.

4. Moving towards stateful and tracing - Serverless functions are ephemeral and will almost always interact with external services, mainly because they intrinsically cannot persist data.

Whether it is a database, object storage, a data lake, or else, functions need external storage to accomplish tasks that rely on the statefulness of data. Other services interacting with functions may be message queues, data streaming processing, authentication mechanisms, machine learning models, etc.

While tracking the entire lifecycle of a serverless function, runtime execution is essential for performance improvement, security monitoring, debugging, etc. All these external interactions pose difficulties for that. One crucial thing to consider when choosing a tracing system for instrumentation is its scalability and availability. Since serverless platforms, by definition, can scale very quickly and offer high availability, the tracing system must be able to cope with the elastic demand of serverless functions. For this reason, solutions specially tailored for instrumenting serverless functions need to evolve.

5. Performance: Although serverless functions offer virtually infinite elasticity in terms of scalability, it does not mean they should run unmonitored. It is paramount to set up thresholds of performance expectations so that it is possible to determine when something requires attention.
Some of the key measures to look for are:

- Invocations count
- Count of runtime crashes, application failures, cold starts, retries
- Memory utilization
- Duration of executions

Cold starts happen when our function does not have enough containers to serve the number of requests coming in at a given point in time. This forces the underlying serverless platform to spin up a new container – which may take from a few hundred milliseconds to several seconds – while the requester is waiting for a response. There are many scenarios where this is undesirable. If that is the case for our application, we need to detect and monitor cold starts in our stack. Cloud services usually will not provide this information directly, but monitoring services need to evolve.

6. IAM Serverless Functions can integrate with services in a domain and potentially cross trust domains. This makes trust management and access management complex for Container Image-based Serverless and FaaS. Having uniform service trust and user identity available across platforms and domains is critical as the use of serverless grows. There is work in progress in the industry for developing SPIFFE and the SPIFFE Runtime Environment (SPIRE) for cross-platform authentication and access management. SPIFFE is still not a formal standard yet but is in the process of being ratified. SPIFFE/SPIRE also help to improve observability, monitoring, and ultimately Service Level Objectives (SLO). By normalizing software identity across various systems (not necessarily just containerized or cloud native), and providing an audit trail of identity issuance and usage, SPIFFE/SPIRE can greatly improve situational awareness before, during, and after an incident occurs. More mature teams may even find that it improves their ability to predict problems before they impact service availability.

7. Supply Chain Developers may use various libraries in their functions. Using libraries saves the developer time by leveraging and reusing existing functionality rather than writing it. These libraries may be developed by third parties or could be open source. There is typically no guarantee that the library is efficient and without vulnerabilities. The developer of the library may have used other libraries. The number of libraries may be several layers deep in their supply chain.

When a piece of code uses a library in the logic it executes, that library is technically a dependency. A layer of code is introduced as a dependency in the code. As dependencies use other dependencies, these layers get deeper. A vulnerability may exist at any layer or branch, and it may affect the serverless function depending on the severity and difficulty to exploit. Therefore, it is essential to understand the dependency tree and keep it as short and narrow as possible. Periodically checking your dependencies for vulnerabilities is a good practice to maintain the security posture of your application. Also, there is much work being done around supply-chain security of the Cloud-native technologies and functions after a series of compromises like Node.JS and recent Solarigate breach. These best practices will provide mechanisms for establishing trust in the dependencies from supply chain software and keep serverless functions secure from vulnerabilities in the dependencies from third-party libraries and open source-software.

Serverless is still in sort of an early stage compared to other technologies, and knowledge will keep growing in the next few years about the best approaches to better develop, deploy and monitor applications. (Martin Fowler)

7.3 Serverless Advances for Data Privacy

As serverless is poised to become the biggest driver in shaping how organizations build, consume and integrate with the Cloud-native capabilities, (due to the way Cloud providers want to increase stickiness and have been increasing their BaaS offerings) there needs to be mechanisms in order to safeguard data privacy.

In the last year, we have seen a rise in confidential computing[2] from the most prominent vendors, e.g. (AWS Nitro, Azure confidential computing, GCP confidential computing). The trend is that some of these capabilities will also be ported to serverless [Gartner, 2021].

As more workloads shift to serverless, encrypting and working on data, state-data, and metadata becomes a requirement. If we look at some examples that probably will make it into the serverless offering of the CSP's:

Self-protection: functions will, in real-time, evaluate and adapt the security and micro-segmentation around each resource, even by using predictive analytics(AI/ML) to evaluate the security posture and define new alerting mechanisms.

Pre-defined compliant functions: A set of functions that developers could leverage and use, according to the guardrails established and type of data required to be used. This could be a natural extension to set up guardrails, a kind of catalog for serverless. This would have the benefits of creating guardrails without reducing agility.

[2] "By 2025, 50% of large organizations will adopt privacy-enhancing computation for processing data in untrusted environments and multiparty data analytics use cases."- Gartner.

Secure enclaves for FaaS: With the scrutiny and assurance that some regulated industries want from the Cloud environments, organizations need to leverage mechanisms to be compliant with ever-increasing regulations.

We have seen that trusted execution environments have been extensively used to maintain the confidentiality, integrity of the data, code, and computation. One example of this would be the European General Data Protection Regulation (EU GDPR), which is globally relevant. These mechanisms would benefit from having available cloud environments specifically in Faas functions as they start to be used in more traditional IaaS environments.

This technology would allow the code, data, and processing to be better isolated from other customers, and from the CSP. Especially important when it contains personal or sensitive data.

Some libraries already exist (i.e., IEEE Xplore[3], Graphene[4]) that offer enclaves as a service and include the natural extension of making their way into FaaS.

8. Conclusions

IT organizations in nearly all industries feel the pressure to deliver value faster, get ahead of the competition and provide new experiences to customers at a rapid rate. Serverless platforms allow application teams to deliver value without managing the infrastructure the application runs on. As this movement gains steam, we will see a proliferation of serverless platforms and more high-value applications being put on these platforms. Security concerns on Serverless platforms are going to grow from here.

As the CSA Serverless Working Group, we have captured all aspects of Serverless Security, including both the Container Image-based Serverless and FaaS platforms as they exist today and as we envision they will evolve. We will provide new revisions of the document when we see more significant changes in the Serverless platform.

[3] https://ieeexplore.ieee.org/document/7163017
[4] https://grapheneproject.io/

9. References

[Ananthanarayanan et al., 2011]	Ananthanarayanan, G., Ghodsi, A., Shenker, S., Stoica, I. (2011). *Disk-Locality in Datacenter Computing Considered Irrelevant. University of California, Berkeley.* https://people.eecs.berkeley.edu/~alig/papers/disk-locality-irrelevant.pdf
[Berkeley, 2019]	Berkeley. (2019). Electrical Engineering and Computer Sciences University of California at Berkeley. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf
[CSA, 2019]	Cloud Security Alliance. (2019). *The 12 Most Critical Risks for Serverless Applications.* https://cloudsecurityalliance.org/artifacts/the-12-most-critical-risks-for-serverless-applications
[CSA, 2013]	Cloud Security Alliance. (2013). Top Threats Working Group. *The Notorious NineCloud Computing Top Threats in 2013.* https://downloads.cloudsecurityalliance.org/initiatives/top_threats/The_Notorious_Nine_Cloud_Computing_Top_Threats_in_2013.pdf
[Forrester, 2020]	Forrester research. (2020). The Forrester New Wave™: *Function-As-A-Service Platforms, Q1 2020. The Nine Providers That Matter Most And How They Stack Up.* https://reprints.forrester.com/#/assets/2/108/RES155938/reports
[Gartner, 2021]	r3. Gartner Report. (2021). Top Strategic Technology Trends for 2021: Privacy- Enhancing Computation. https://www.r3.com/gartner-2021-privacy-enhancing-computation/
[IEEE Spectrum, 2020]	Fahmida, R. (2020). IEEE Spectrum. What is Confidential Computing?. https://spectrum.ieee.org/what-is-confidential-computing
[Jericho Forum-White Paper, 2007]	The Open Group Library. Jericho Forum - White Paper. (2007). *Business rationale for de-perimeterisation.* https://collaboration.opengroup.org/jericho/Business_Case_for_DP_v1.0.pdf
[Jericho Forum Commandments, 2007]	The Open Group Jericho Forum. (2007). *Jericho Forum Commandments.* https://publications.opengroup.org/w124
[Kubernetes, 2021]	Kubernetes. (2021). Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/
[Manral, 2021]	Manral V. (2021). The Evolution of Cloud Computing and the Updated Shared Responsibility. *Cloud Security Alliance Blog Article.* https://cloudsecurityalliance.org/blog/2021/02/04/the-evolution-of-cloud-computing-and-the-updated-shared-responsibility/

[NIST SP 800-123, 2008]	NIST Special Publication 800-123. (2008). Guide to General Server Security. Recommendations of the National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-123.pdf
[Node-RED, 2020]	Node-RED. (2020). Low-code programming for event-driven applications. https://nodered.org/
[OpenWHISK, 2020]	OpenWHISK. (2020). Open Source Serverless Cloud Platform. *Apache OpenWhisk*. https://openwhisk.apache.org/
[OWASP, 2017]	OWASP. (2017). OWASP Top 10. *Interpretation for Serverless.* https://owasp.org/www-pdf-archive/OWASP-Top-10-Serverless-Interpretation-en.pdf
[OWASP, 2019]	OWASP. (2019). OWASP API Security Project. *API Security Top 10 2019.* https://owasp.org/www-project-api-security/
[PCI, 2018]	PCI. Security Standards Council. (2018). DSS 3.2.1 Standard. Requirements and Security Assessment Procedures. https://www.pcisecuritystandards.org/document_library https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf?agreement=true&time=1615400173449
[Rahic, 2020]	Rahic, A. (2020). Fantastic serverless security risks and where to find them. *Serverless*. https://www.serverless.com/blog/fantastic-serverless-security-risks-and-where-to-find-them
[Shankar et al., 2018]	Shankar, V., Krauth, K., Pu, Q., Jonas, E., Venkataraman, S., Stoica, I., Recht, B., and Ragan-Kelley, J. (2018). *numpywren: Serverless Linear Algebra.* https://arxiv.org/pdf/1810.09679.pdf
[Veryard, 2011]	Veryard, R. (2011). Architecture, Data and Intelligence. Service Boundaries in SOA. https://rvsoapbox.blogspot.com/2011/07/service-boundaries-in-soa.html

Appendix 1: Acronyms

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

ABAC	Attribute-based access control
API	Application Program Interface
BaaS	Backend as a Service
CICD	Continuous integration, continuous delivery
CISO	Chief Information Security Officer
CPU	Central processing unit
CRUD	Create, Read, Update, Delete
CSP	Cloud Service Provider
DevOps	A portmanteau of "development" and "operations."
DLP	Data Loss Prevention
FaaS	Function as a Service
GDPR	General Data Protection Regulation
HIPAA	The Health Insurance Portability and Accountability Act of 1996
IaaS	Infrastructure as a Service
IT	Information Technology
KMS	Key Management System
NIST	National Institute of Standards and Technology
OPA	OPA (Open Policy Agent) is an open-source, general-purpose policy engine that unifies policy enforcement across the stack. OPA provides a high-level declarative language that lets you specify policy as code and simple APIs to offload policy decision-making from your software.
OS	Operating System
PaaS	Platform as a Service
PCI	Payment Card Industry
SLO	Service Level Objectives
SPIFFE	Secure Production Identity Framework for Everyone
SPIRE	SPIFFE Runtime Environment

STS	Security Token Service
VNet	Virtual Network
VPC	Virtual Private Cloud

Appendix 2: Glossary

Service boundaries	Service boundaries are defined by the declarative description of the functionality provided by the service. A service - within its boundary - owns, encapsulates and protects its private data and only chooses to expose certain (business) functions outside the boundary.