altice labs

Security guidelines applied to microservices cloud architectures

Security; Cloud Computing; Microservices

White paper

Version 1.0, June 2024

Introduction

In recent years, microservices architecture has become preferred for designing and implementing application systems in cloud and on-premises infrastructures. This type of architecture enables scalability, flexibility, and resilience, making it an optimal solution for modern software development. Despite the advantages, it also poses new security challenges compared to traditional models.

This paper aims to provide comprehensive guidelines and best practices for securing generic microservice applications deployed in a cloud environment. In the following sections, we will explain basic concepts related to microservices and cloud computing and highlight their key features and benefits. Furthermore, we will provide practical recommendations for dealing with everyday security problems in microservices architecture based on credible literature and our own experience.

Context

Microservices[1] and cloud-native applications[2] are two different concepts that are often used together in the context of software architecture design and development. Microservices refers to a software architectural style in which an extensive application is broken down into several small and independent services that communicate with each other over well-defined interfaces, such as APIs. Each microservice is designed to perform a specific function and can be developed, deployed, and scaled independently of the other services. A microservices architecture aims to enable flexible and efficient development and deployment of large, complex applications by breaking them into smaller, modular components. Cloud-native applications are designed to be deployed and run in a cloud computing environment. They are built with technologies and practices that take advantage of the characteristics of cloud computing, such as on-demand scalability, elasticity, and the ability to deploy and update applications quickly. Cloud-native applications are often developed using microservices architecture, but they can also be built using other models, such as monolithic or serverless[3].

On the other hand, microservices do not have to run in the cloud. Developers can use platforms like Kubernetes[4] to deploy them on-premises. However, experienced developers already know why microservices are central to cloud-native apps and why they are beneficial. But, to actually and adequately build cloud-native applications based on a microservices architecture, developers must be well-versed in specific tools, programming languages, development techniques, and security know-how. This paper gives an overview of these concepts and, due to its relevance, focuses on the security aspects of microservices architectures.

Article

Microservices architecture has been around longer than cloud-native computing. It started to become famous about a decade ago, whereas the term ***cloud-native*** emerged around 2015[5]. However, cloud-native applications are based on cloud computing principles that extend back to the 1960s. So, let's start from the beginning and go a bit deeper into these concepts to understand the advantages and possible disadvantages of each one of these technologies and how they can benefit from each other.

Principles of Cloud Computing

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction[6] [7]. This cloud model promotes availability and has five essential characteristics: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. In the past, organisations relied on local data centres to provide infrastructure for their applications and workloads. However, these infrastructures often require the purchase of underutilised hardware and do not offer the necessary elasticity for their workloads [1]. As a result, many organisations have started to host their applications in the cloud, which automatically solves those issues.

Besides offering users more flexibility in choosing their resources and providing quick and easy access to them, clouds have numerous other advantages over conventional IT infrastructures. Nevertheless, these advantages can also become disadvantages if the necessary precautions are not taken. Therefore, both should be covered simultaneously when presenting the features related to the cloud to understand their positive and negative aspects clearly.

The most significant advantage is a ***faster time to market***. New instances can be created or removed from the cloud in seconds, thus speeding up deployment times. Also, by following an ***on-demand self-service approach***, customers can automatically request services based on their needs. Companies can also count on ***rapid elasticity*** since cloud computing allows them to rapidly scale resources, launching more instances during peak loads and quickly downsizing while on periods of low activity without investing in physical infrastructure.

Clouds can lead to substantial ***cost savings***, considering that companies do not need to invest in their infrastructure as it is managed and maintained by the cloud provider. Clients only pay for the resources and services used, regardless of the service model. However, this is not a linear advantage since costs can become very high if resource management is not done carefully.

Resiliency is usually a plus; cloud environments offer backup and disaster recovery features. Cloud backup can be online or remote, and it's a strategy for sending a copy of a physical/virtual file or database to a secondary, off-site location. It ensures data safety and prevents loss, even in hardware failure, cyber-attacks, and user error scenarios. Cloud DR (Cloud disaster recovery) is a broader concept. It combines services and strategies to store backup data, apps, and other resources in cloud storage through public clouds or a dedicated service provider. It is important to be deployed across multiple data centre locations, and these factors contribute to enhancing resilience. Cloud providers can also offer high ***availability*** (% of the operational time) and ***reliability*** (high performance and correct output) environments (i.e. SLAs - Service-Level Agreements - of 99,9%).

However, these environments are complex in terms of ***security and privacy*** since cloud computing has brought many advantages while creating new problems. The ownership by third parties is considered one of the primary reasons some organisations hesitate to adopt cloud computing [2]. Some other common issues commonly associated with cloud computing include the loss of physical control of data, sharing resources with other users (multi-tenancy) [3], and greater data exposure due to being accessible via the Internet. Nonetheless, public cloud providers nowadays provide various security features such as encryption, access control, monitoring and private connections, which can be utilised to protect systems and data. However, there is still the risk of vendor lock-in; each cloud provider gladly offers different types of services that can be useful and avoid extra development time, but they create dependencies, making the migration to other cloud providers more difficult.

Several options are available to deploy cloud clients' systems, also known as **Cloud Deployment Models**, to minimise some risks and build a solution that fits different needs. Based on selected requirements, clients can choose to deploy their systems on a private cloud[8] [9] (on-premises or by a third party - managed or virtual), public cloud, community cloud (cloud shared by multiple organisations or clients forming a community), or a hybrid cloud (where the system is deployed in a mix of two or more of the previous deployment models).

There are various **Cloud Service Models** to choose from (**Figure 1**), such as infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS), based on the level of control, flexibility, and management needed..

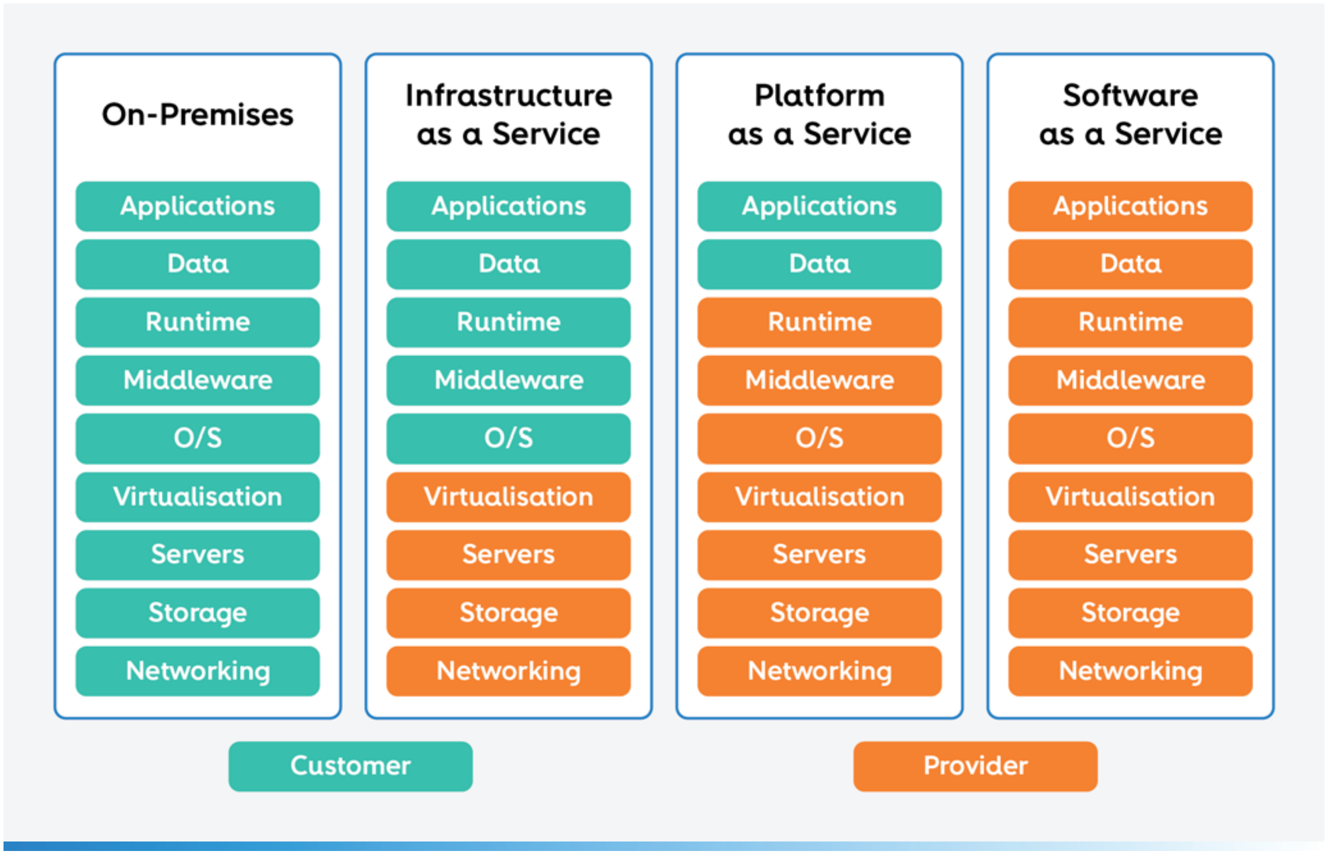
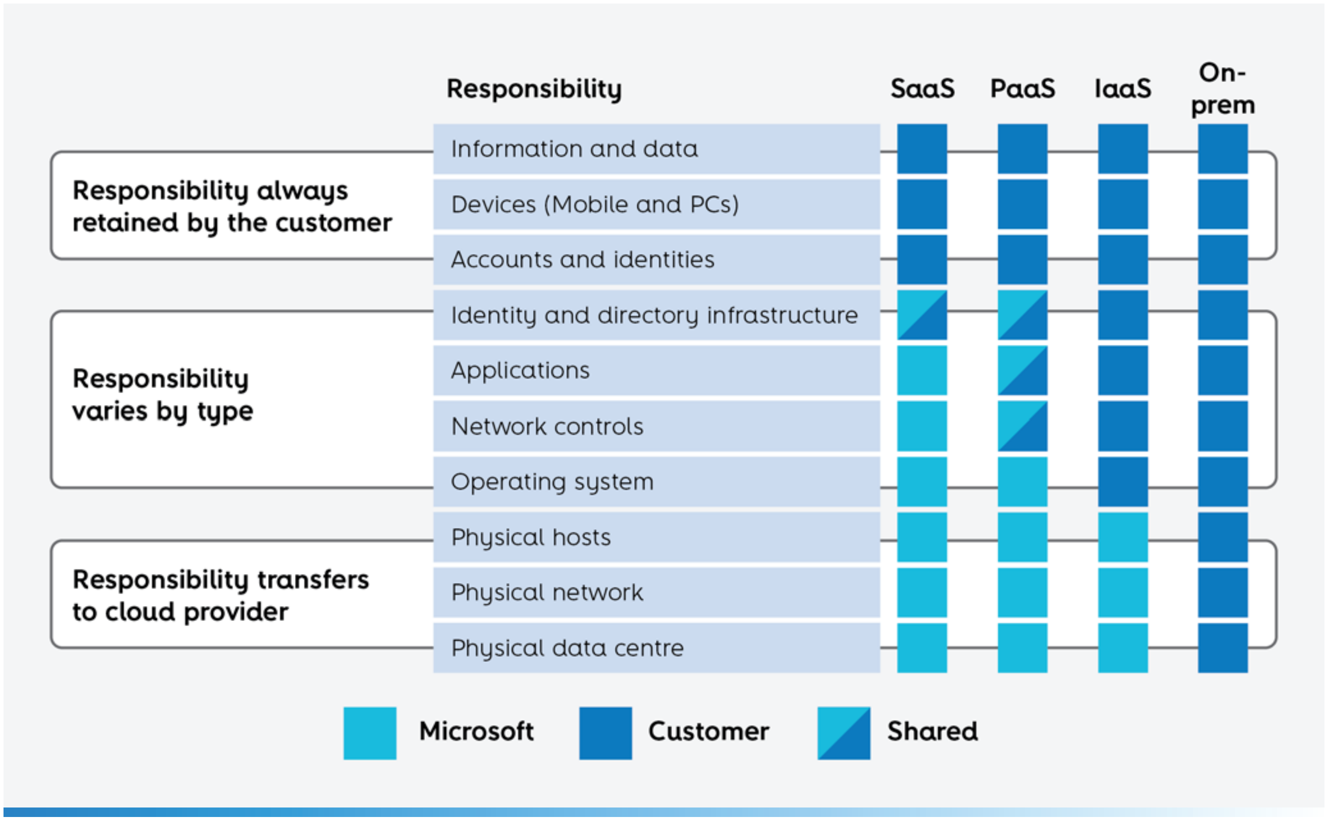


Figure 1 – Customers' resource control in the different cloud services models comparing with the on-premises deployment model

- Iaas offers more flexibility over the IT resources. In this case, it provides their consumers with the fundamental computing resources (e.g., storage, networks, processors) to deploy their applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications. Amazon (AWS), Google (GCP), Microsoft (Azure), IBM (Cloud), Cisco (Metacloud), and ORACLE (cloud) are some of the big companies that offer IaaS[10].

- Paas offers a platform where consumers can execute their applications. Unlike the previous model, the consumer does not have control over the underlying cloud infrastructure, operating systems, or storage, only the deployed applications. App Engine (Google), App Service (Azure), and Elastic Beanstalk (AWS) are some examples of PaaS. Also, Kubernetes Managed Services such as GKE (Google), AKS (Azure), and EKS (Amazon).
- Saas offers applications deployed in the cloud that customers can access through a thin client interface, such as a web browser. In this model, the consumer does not control the underlying cloud infrastructure or the application itself. Google (Apps), Microsoft Office 365, and Dropbox are good examples of SaaS implementations.

Cloud technology has become popular and widely used. However, it is essential to note that the cloud, like any other technology, is vulnerable to multiple attacks. Hence, ensuring cloud security is crucial and is one of the primary challenges in adopting cloud systems. While cloud providers are responsible for the management of infrastructure, security and data protection are a **Shared Responsibility** of both the customer and the provider. The level of shared responsibility may differ depending on the provider and the service model used. Generally, the more control the client has over the workloads, the greater the responsibility to protect them. Therefore, it is essential to have a clear understanding of the responsibilities and take the necessary measures to ensure the security of the system. **Figure 2** illustrates an example of the security responsibilities of clients and providers for a specific cloud provider.



	Responsibility	SaaS	PaaS	IaaS	On-prem
Responsibility always retained by the customer	Information and data	■	■	■	■
	Devices (Mobile and PCs)	■	■	■	■
	Accounts and identities	■	■	■	■
Responsibility varies by type	Identity and directory infrastructure	◩	◩	■	■
	Applications	□	◩	■	■
	Network controls	□	◩	■	■
	Operating system	□	□	■	■
Responsibility transfers to cloud provider	Physical hosts	□	□	□	■
	Physical network	□	□	□	■
	Physical data centre	□	□	□	■

□ Microsoft ■ Customer ◩ Shared

Figure 2 – Azure's Shared Responsibility Model

Principles of microservices architecture

Microservices is a software architecture model that has recently gained popularity because of its many benefits over other existing ones. The term “microservices” was first used in 2011 in a workshop to describe the participants’ common ideas in software architecture patterns [4]. Although there is no precise definition of this type of architecture, in 2014, Martin Fowler and James Lewis published their “Microservice” article[11], which became a de-facto standard for defining microservices. This article describes Microservices Architecture as “an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and are independently deployable by fully automated deployment machinery.”

The main characteristics of microservices architecture can be summarised as follows:

- *Componentization via Services:* in a microservice architecture, services are software components that can be upgraded or replaced independently, meaning that if a service is modified, there is no need to redeploy the entire application, but only the specific service that has been changed.
- *Organized around Business Capabilities:* traditional architectures are organized around the technology layer, with each team in charge of a specific aspect, such as the UI, database, service logic, etc. However, a simple change in the application could require the involvement of multiple teams, resulting in time and budget constraints. In contrast, microservices architecture is structured around business capabilities, with each service managed by a team responsible for all aspects of its development. This approach allows teams to work independently and at their own pace, creating more cohesive and loosely coupled services.
- *Products not Projects:* the development team aims to deliver working software in traditional software projects. Once the software is provided, it is handed over to a support and maintenance team, and the development team moves on to the next project. In microservices, the development team should be responsible for owning a product throughout its lifetime. It means developers can see how the software behaves in production and interact with customers more frequently, resulting in better customer satisfaction.
- *Smart endpoints and dumb pipes:* the communication between components can be accomplished using different approaches. Some methods use complex mechanisms such as ESB (Enterprise Service Bus), which offer advanced features like message routing, choreography, transformation, and business rule application. However, such approaches can be complicated to maintain. In contrast, microservices favour “dumb pipes,” which are simple, lightweight protocols such as REST (Representational State Transfer) over HTTP. This approach leaves the complexity to the service logic itself, resulting in a more manageable and maintainable system.
- *Decentralized Governance:* in traditional projects, there is usually a set of standards for software development. For example, which programming language to use, what type of database to use, and how to create logs, among others. In microservices, each team should be entirely responsible for the lifecycle of their service and can make their own decisions, such as what technology to use.
- *Decentralized Data Management:* traditional systems have a single shared database, which stores all the system’s data from all components. With microservices, each service has its database, the technology of which may differ from the others (i.e., Polyglot Persistence). It allows developers to choose the right technology for the right problem.

- *Infrastructure Automation:* deploying microservices manually can be challenging due to the amount of service possibilities. Automation techniques like automated testing and deployment can reduce the complexity and time required for building and deploying microservices.
- *Design for failure:* in a microservice architecture, there are a lot of processes and network traffic, which increases the likelihood of failure. Since it may occur anytime, detecting and handling this failure as quickly as possible is necessary. Therefore, monitoring and logging setups for each service should be in place.
- *Evolutionary Design:* is defined as the capacity to change an application over time. That can happen for multiple reasons: fix bugs, add new features, bring in new technologies, or make existing systems more scalable and resilient. There may also be a need to add some services for some period and discard them later. Microservices allow us to manage this process more efficiently.

Given this list of characteristics, it is fair to say that the strongest virtues of microservices architecture are 1) ***the organizational alignment***: microservices allow alignment between the architecture and the organisation, where the services can be altered according to organisational changes; 2) ***independent deployment***: services can change and be deployed independently without deploying the entire system. It allows developers to deploy the code more frequently and quickly put new functionalities on the market; 3) ***independent scalability of components***: in a microservices architecture, individual services can scale without scaling the entire system, unlike traditional architectures; 4) ***robustness***: microservices are designed to be isolated, so if one fails, the others should continue to function normally. It enhances the system's resilience and reduces the impact of any failures; and finally, 5) ***com-posability***: in microservices, the services are independent and have well-defined boundaries to reuse in different projects. It prevents developers from writing the same functionality again, allowing them to build new applications more quickly.

Although some microservices' characteristics present advantages over traditional architectures, they also raise some concerns, such as a) ***technology heterogeny*** - different technologies can be used for each service. It enables the development team to choose the most appropriate tool for each task. For instance, if one part of the system requires enhanced performance, a different technology might achieve the desired performance levels. This approach encourages the adoption of new technologies into the system. However, attempting to adopt too many new technologies simultaneously can lead to overload and reduce the time needed to deliver new functionalities to the users. Therefore, there is a need to balance the advantages of technology diversity with the disadvantages of complexity.

Also, b) ***the cost*** - choosing a microservices architecture might not be the best option for those who aim to reduce costs because costs might increase rapidly, as there is the need to maintain multiple services and have additional resources like storage, processors, and machines to run them. Implementing this type of architecture takes time efficiently, which could delay the delivery process.

But that is not all. As already known, c) ***monitoring and troubleshooting*** are essential for any system in production, but in particular to microservices architectures; they help detect issues that may arise in the system and understand what happened and how to prevent these problems from happening again. However, monitoring becomes more challenging to implement when dealing with microservices, mainly due to their distributed nature. As the system's complexity increases, the number of services that need to be monitored also increases, and so do the chances of having to support multiple logging solutions. If the services have various copies distributed across different regions, a strategy is needed to aggregate their logs, metrics, and traces to overview what is happening comprehensively. It also presents more challenges since system managers may need help collecting up-to-date data about their status.

Other significant concerns related to its distributed nature are d) testing, which is crucial for ensuring that the system performs as intended, but it becomes more complex in a microservice architecture; multiple components must be tested independently and as part of the more extensive application, increasing the number of unit tests and the scope of end-to-end tests. As the system grows, the time and complexity required to run the tests increases, demanding efficient testing strategies. Also, e) ***latency*** can become a problem and should be handled differently from traditional architectures; microservices may cause some delay because processing is often distributed across multiple independent services instead of a single process. Since microservices are a distributed architecture, the data needs to be serialised, transmitted and deserialised over networks, which can cause an increase in the time it takes to complete operations. And last, f) ***Data consistency***; unlike the traditional architectures, which are managed in a single database, microservices will have different databases managed by other processes. It might lead to data consistency issues and increased complexity since the traditional transactions used to manage the database state must be distributed and coordinated among several services [5]. Therefore, implementing and coordinating distributed transactions, as well as avoiding data duplication, is a challenging task.

However, on top of these concerns, ***security*** is often the biggest. Microservices architectures present significant security challenges due to their heterogeneous and distributed nature. As the number of microservices increases, there are more interactions between components, more communication links to be protected and more attacks to exploit. This architecture also deals with increasing data flow over networks, and the infrastructure quickly becomes more complex. Each microservice has its entry points, significantly expanding the system's total number of entry points, thus contributing to an ever-increasing attack surface. Therefore, it is essential to ensure as a baseline that each microservice entry point is protected with equal strength, as the security of the entire system is only as strong as its weakest link. However, microservices architecture also presents more opportunities to implement a defense-in-depth strategy compared to conventional architectures [5]; since the system's functionality is divided into distinct components, it is possible to limit each component's actions and apply different levels of security depending on the sensitivity and importance of the microservices [5] [6].

Security aspects and recommendations about microservices

The truth is that ensuring the security of a microservices system is indeed a challenging task. Specific standards can assist developers in securing their applications in general and in particular when developing microservices architectures. The following fundamental principles should guide the design process throughout its lifecycle.

Least privilege

This principle states that permissions of users or services should be restricted to only those necessary for them to complete their tasks within a specific period. This approach offers several benefits, such as limiting the actions that can be taken by an attacker who has acquired valid credentials and reducing the potential damage caused. The best practice is to start with a deny-by-default policy - users or services initially have no permissions - and then to grant them only the ones required to complete their tasks.

Defense-in-depth

This principle is based on the idea that the security controls of our system may fail. Therefore, it is essential to create multiple layers of security. This approach ensures that the other layers can help mitigate attacks if one fails. In a microservice architecture, where the system's functionality is divided among different services, traditional security models are ineffective in protecting other services once a single one is compromised. Hence, it is crucial to implement defense-in-depth strategies for each microservice.

Zero Trust

A security model that relies on the principle of "Never trust, always verify." Traditional security models only authenticate users during their initial access, allowing them broad access to resources within the internal network. However, this model is vulnerable to internal attacks and lateral movements despite being protected from external attacks by solid firewalls. In contrast, Zero Trust has no implicit trust, meaning everything must be verified, even for internal users. In a zero-trust model, it is assumed that the environment has already been compromised. The core principle is that Trust should be earned from a user or another system. Therefore, Trust should only be established if there is enough evidence that it is legitimate, such as proof of solid authentication and authorisation.

When it comes to securing microservices, there is no universal solution. The approach to security will depend on several factors, such as the system's requirements, the sensitivity of the data it handles, the organisation's policies, the budget available, etc. Some of the most common problems in this type of architecture are described. Then, several possible solutions/recommendations to mitigate them are suggested to help developers improve their design and implementation efforts. However, it is essential to note that the recommendations provided here are generic and should only be used as a starting point. As a regular practice, it is highly recommended to establish a threat model at the beginning of the architecture design process to identify and address specific vulnerabilities and apply the appropriate countermeasures. Additionally, conducting regular risk assessments and validating security policies is crucial to ensuring overall system security.

The problems are categorised based on some of the most relevant topics to consider when designing a microservices system's security.

Edge security

Ensuring user authentication and access control is crucial to securing a microservice architecture. Authentication is the process of confirming that someone is who they say they are, which helps in preventing spoofing attacks. Multifactor authentication is highly recommended to authenticate human users. Including an additional authentication factor alongside the standard username and password can reduce the risk of an account breach by an impressive 99.99%. Authorisation verifies whether an authenticated user or service has the necessary permissions to perform a specific action. In a microservices architecture, authorisation can be implemented at two levels - at the edge using an API Gateway or at the service level. It is also possible to carry out authorisation at both levels. For more straightforward solutions, authorisation is performed only at the edge level using an API Gateway. However, it is recommended to be implemented at both levels to ensure maximum security and granular access control. OAuth 2.0[12] can be used as it´s an industry-standard protocol.

Service-to-service communication

Once the client requests have passed through the authentication and authorisation mechanisms and reached the microservices, the next step is to ensure the security of communications between them. Unlike traditional architectures, where data circulates within a process, microservices have more communications made at the network level, making them more vulnerable to man-in-the-middle attacks. Therefore, microservices security design must be done with the understanding that the network is a hostile place. Following the zero-trust network principle, all requests must be authenticated and authorised before processing. Two standard methods to secure communications between microservices are [7] JSON (JavaScript Object Notation), Web Tokens (JWT[13]) and mTLS (mutual Transport Layer Security).

JWT is an open standard that defines a compact and self-contained way to transmit information between parties as a JSON object [8]. This information can be signed and/or encrypted using a secret (symmetric cryptography) or a public/private key pair (asymmetric cryptography). In a typical microservices environment, these JWTs are issued by a single trusted microservice, STS (Security Token Service). They are regularly used when there is a need to pass context about the user to the microservices.

mTLS is an extension of the TLS protocol that allows mutual authentication. In a connection between client and server, TLS uses asymmetric or public-key cryptography to ensure data confidentiality, integrity, and server authentication. Additionally, mTLS ensures client authentication. mTLS is a widely used mechanism for service-to-service authentication in microservices architecture. Each microservice in the network has a public/private key pair generated by a trusted Certificate Authority, which uses that key pair to authenticate itself against other microservices.

Although the recommended methods are different, they should be used together to provide a second layer of security [7].

Logging

Keeping a detailed log of all system events is crucial as it helps identify and troubleshoot any issues that might have affected the performance or security of the system. For microservice architectures, it is recommended to take the following measures:

- Microservices should write log messages to a local log file instead of sending them directly to the central logging system via network communication. It mitigates the risk of data loss due to logging service failure or attack [9].
- A logging agent should collect log data on the microservice (read local log file) and send it to the central logging system. This agent should be a dedicated component decoupled from the microservice [9].
- Microservices and logging agents should sanitise logs containing sensitive data, such as passwords and API keys. These messages should not be sent to the central logging system to adhere to the data minimisation principle [9].

- The logging agent must publish log messages in a structured format such as JSON, CSV (Comma-Separated Values) or Syslog. Standardised logging allows easy parsing and analysis, simplifying issue identification and troubleshooting [9].
- The logs should be secure with access controls and encryption to prevent unauthorised access and data exposure [9].

Container security

Microservices architecture demands isolation between services to prevent any interference of one service with another. The most popular method [4] is using containers to achieve this. Containers are "lightweight" virtual machines containing all the dependencies an application requires. Unlike VMs, which emulate the entire hardware and software, including the hypervisor and kernel, containers are much lighter. They emulate only the OS (and share the same kernel), which allows them to run either on a VM or directly on "bare-metal" servers. Although they do not have a strong isolation level like VMs, containers are more lightweight, enabling more efficient scaling and quicker spin-up times. Therefore, containers are commonly used for the packaging and deployment of microservices and for that reason, it is essential to consider the security of containers when designing microservices.

- Run immutable containers: containers are mutable by default. As a result, if an attacker gains access to a container, they can easily install malicious software. Thus, it is recommended that the filesystem be configured to read-only and that shell access within the container be restricted [10]. In cases where there is a need to update the container, a new tag for the container image should be created to replace the older version.

- Minimise Image Footprint: the base image is crucial when creating a container image, as it holds the operating system and additional dependencies. However, standard images like Debian often come with features that are not required to run applications. It can increase the attack surface and make the image vulnerable to security threats. For this reason, using minimal images as a base is recommended. They boast only the necessary functionality required to run the application, thus minimising the attack surface.
- Limit Container privileges: if an attacker successfully gains access to a container with root privileges, they can install tools within it. These tools can then be used to identify any vulnerabilities in other services on the network or to gain unauthorised access to the underlying host. Running containers as non-root users are strongly recommended whenever possible to mitigate this risk. In cases where the container needs to perform some operation with privileged or root access, it is recommended to use capabilities with the principle of least privilege instead. Capabilities are a Linux feature that allows for the definition of all the privileged operations which the root user can perform. By default, Docker starts containers with a restricted set of Linux kernel capabilities. When using capabilities, it is possible to run privileged operations without full root access. In short, it is recommended to use capabilities instead of giving a container full root access and limit them to only those necessary for the container to perform its job [11].

Typically, it is expected to have hundreds of small, independent services, which can make its management quite complicated. Therefore, containers are often used alongside orchestrators to automate the scaling, deployment, and security processes. Hence, it is crucial to consider also the orchestration platform while securing the microservices deployment. Regarding container orchestration platforms, Kubernetes is the most commonly used.

Container orchestrator security

It is important to note that the security of the orchestrator[14] will depend on how it is deployed. For instance, if a managed orchestrator like GKE (Google Kubernetes Engine) is used, the cloud provider takes responsibility for the control plane's security. However, as an alternative, the client may take full responsibility for the orchestrator's security to meet specific requirements. In this case, here are some recommendations[15] to enhance the security of a Kubernetes orchestrator:

- Pod Security (Security Context) - pods are the minor deployable units in Kubernetes and can harbour one or more containers. Attackers often target pods after exploiting containers, so it is essential to harden them to minimise the impact of successful attacks. Kubernetes provides various tools to define and enforce security mechanisms at the pod level, strengthening the system against possible attacks. Security contexts are configurations at the pod level that enhance container security. The security context can apply specific container-level security recommendations, such as limiting container privileges and ensuring immutability, thereby making the system more secure.
- Network Policies - by default, pods in a Kubernetes cluster can communicate with each other, but this can pose a risk since a compromised pod can affect others. One way to address this is through network policies restricting pod-to-pod communication. Network policies specify how a pod can communicate with various network entities (pods, namespaces, or IPs). The principle of least privilege should be applied by restricting all traffic by default and creating specific policies to allow necessary traffic.
- Restrict Access to API Server - the API Server is a crucial part of Kubernetes, bridging the cluster and its users. It provides HTTP REST endpoints that allow users and cluster components to execute tasks within the cluster. These REST calls can be accessed externally and internally using command line tools like kubectl or kubeadm. Since the API Server has significant permissions within the cluster, it should never be exposed on the Internet. Some cloud providers offer security features such as private clusters that restrict access to the API Server. For on-premises Kubernetes clusters, it is essential to set up firewall rules preventing API server access.
- Restricting User Permissions - it is advisable to avoid using or sharing administrator accounts to carry out operations within the cluster. Instead, accounts with limited permissions should be created according to the least privilege principle to restrict an attacker's actions if they gain access to the credentials. Doing so can ensure greater security by reducing the potential damage in case of a breach.
- Disable Anonymous Requests - users can access the server without providing credentials if anonymous requests are enabled. This behaviour is not recommended as it can allow attackers to enter the cluster without authentication. In some managed clusters, changing this setting may not be possible, as users do not have control over the Control Plane. Thus, it is highly recommended that strict RBAC (Role-Based Access Control) rules be used to limit access to the API server.
- Restricting Permissions for a Service Account - pods use the service account's token to authenticate with the API server. In Kubernetes, it is standard to mount the token file at */var/run/secrets/kubernetes.io/serviceaccount/token*. However, this practice can become a security risk if a workload is compromised. Attackers can use service account tokens to compromise the cluster further. Therefore, it is recommended

that service account tokens should only be mounted in pods if there is a specific need for the workload in the pod to interact with the API server. For these cases, RBAC should be implemented to restrict pod privileges within the cluster.

- Etcd Encryption - Etcd is a database that stores all information about the state of the cluster, including sensitive data, such as secrets. The location where etcd is running varies depending on the cluster's topology. By default, the data in etcd is stored in plain text, which an attacker can easily read if they gain access to where etcd is located. To ensure the security of the data stored in etcd, it is highly recommended to be encrypted using KMS (Key Management Service) providers, like Google KMS or HashiCorp Vault. It provides an added layer of protection against attackers even if they gain access to etcd or the underlying VM (Virtual Machine).
- Secrets Management - secrets are essential to Kubernetes as they store sensitive data like passwords, authentication tokens, SSH keys, and more. Using Secrets prevents the need for confidential data in the application code. Note that Kubernetes secrets are not encrypted by default. They are only base64 encoded, which means anyone with access to the API or etcd can read and modify their content. In the context of a pod, secrets can be accessed through volumes or environment variables. Once the secret is mounted or exposed as an environment variable, the application running inside the container can read its contents. However, attackers can use the secrets to compromise other system parts if they gain access to the container. To use Secrets more securely, it is recommended to configure their encryption at rest through etcd encryption as mentioned above, restrict the users and containers that have access to secrets using RBAC rules based on the principle of least privilege, or use an external secret provider to keep confidential data out of the cluster, where pods can then be configured to access that information[16] [17]. In addition, it's advisable to mount Secrets as volumes rather than environment variables because a process, user, or malicious code can easily access environment variables from the PID (Process ID) namespace[18] [12]. Even if there is no compromise, applications can dump

their environment when they crash, thus exposing secrets to anyone with access to the logs. Mounting Secrets as volumes allows for updating secret values without restarting the pod. Ideally, it would be best to mount the volume as a temporary filesystem so that the files are not written to disk but are held in memory. It ensures that the secrets are never stored in plain text at rest [12].

Service mesh

Container orchestrations, like Kubernetes, help manage microservice deployment but need some quality-of-service features [7]. By using a service mesh in combination with an orchestrator, it is possible to overcome these complex challenges.

A service mesh[19] is a dedicated infrastructure layer that manages communication and coordination between microservices; it allows to transparently add capabilities to the services like observability, traffic management, and security without changing the microservice code and avoiding implementing them for each service. Typical service mesh architectures consist of 2 components: the control plane and the data plane. The control plane manages tasks such as creating instances, monitoring, and implementing network management and security policies. The data plane comprises service instances, sidecars, and their interactions. Sidecars are proxy instances attached to containers and/or microservices. As these proxies intercept the requests between services, they act as policy enforcement points (PEP), enforcing access control policies based on the Policy Decision Points (PDP) decisions. The PDPs can be distributed and integrated at the service level (Embedded PDP Model) or centralised (Centralised PDP model). The control plane acts as a policy administration point (PAP), configuring the traffic routing proxies and gathering telemetry if needed.

In addition to these specialised proxies for each service, it is also possible to use proxies deployed to manage the traffic entering and leaving the system. An ingress controller is a proxy for incoming traffic, acting as an API gateway. In contrast, an egress controller can regulate outgoing traffic from the system to the outside world. In regular cloud-based systems, these activities are performed by Firewalls and NATs (Network Address Translators), respectively.

There are multiple service mesh solutions, but Istio[20] is the most used according to a 2020 survey by CNCF[21] (Cloud Native Computing Foundation). So, the following recommendations will be based on Istio.

- Authorization Policies - like network policies in Kubernetes, Istio provides a way to manage communication within the system. However, unlike Kubernetes, Istio controls communication at the application level through authorisation policies. Although either method can be used, it's considered best practice to combine both. This approach follows the defense-in-depth strategy, strengthening the system's security [13].
- Service-to-Service Communication – regarding communication between services, Istio supports the two abovementioned methods, namely mTLS and JWT. As seen previously, the use of mTLS brings several security benefits. On one hand, it allows the communication between services to be encrypted. On the other, it guarantees the authenticity of both services. It makes it harder for the attackers to read the exchanged data or to impersonate one of the services (man-in-the-middle attacks). To implement mTLS in the cluster, creating and managing public certificates and private keys for each service is mandatory, which is an excruciating process. However, many service meshes take care of this, managing the keys and generating and rotating certificates automatically. Istio already encrypts service-to-service communications by default, but it uses the permissive mode, which means that it accepts both mTLS and plaintext traffic in cases where it is impossible to implement a sidecar in a given service (e.g. legacy). To ensure that the communication is done using only mTLS, it is recommended that a strict mode be used [13]. Istio also supports JWT authentication, which is validated at the proxy level. It uses a RequestAuthentication CRD (Custom Resource Definition) to define access control policies based on JWT. It is important to note that Istio rejects requests that contain invalid JWTs but does not reject requests without any JWTs. Therefore, it's recommended to use RequestAuthentication CRD policies along with an AuthorizationPolicy CRD that rejects requests without request principals [7].
- Distribute tracing - one of the key features provided by the service mesh is distribution tracking. As the number of services in the system increases, tracking an order on the network becomes more complex. Distributed tracing can analyse how different requests move through the system. It helps to detect possible attacks and other factors that could affect availability. Istio can be combined with a tracing framework like Jaeger to implement this. However, to make this work, applications should be instrumented to forward the headers for communication packets they receive [14].

Finally, despite its advantages, Service Mesh also has some downsides. Since each microservice requires its proxy, this can increase the number of runtime instances and the application's attack surface [14].

Conclusion

This paper gives an in-depth explanation of cloud computing, microservices architecture, and its essential characteristics. It discusses various security recommendations that can be applied to microservices architectures hosted in the cloud. It is crucial to note that there is no one-size-fits-all security solution. The recommendations presented in this document are generic and should be used as guidance. Each use case should be further detailed and analysed to gather security and privacy requirements. Only then will the most appropriate solution be selected.

Endnotes

1 https://www.techtarget.com/searchapparchitecture/definition/microservices

2 https://www.techtarget.com/searchcloudcomputing/definition/cloud-native-application

3 https://www.techtarget.com/searchitoperations/definition/serverless-computing

4 https://www.techtarget.com/searchitoperations/definition/Google-Kubernetes

5 https://www.sdxcentral.com/articles/news/google-friends-launch-cloud-native-computing-foundation/2015/07/

6 https://www.techtarget.com/searchcloudcomputing/definition/cloud-computing

7 https://csrc.nist.gov/projects/cloud-computing

8 What is a Private Cloud? - Private Cloud Explained - AWS (amazon.com)

9 On-Premise Vs Private Cloud: Choosing The Right Infrastructure (stonefly.com)

10 https://www.leanix.net/en/wiki/apm/iaas-vs-paas-vs-saas

11 https://martinfowler.com/articles/microservices.html

12 https://oauth.net/2/

13 RFC 7519 - JSON Web Token (JWT) (ietf.org)

14 https://devopedia.org/container-orchestration

15 https://kubernetes.io/docs/concepts/security/security-checklist/

16 https://kubernetes.io/docs/concepts/configuration/secret/

17 https://kubernetes.io/docs/concepts/security/secrets-good-practices/

18 https://www.tenable.com/audits/items/CIS_Kubernetes_v1.6.1_Level_2_Master.audit:98de3da69271994afb6211cf86ae4c6b

19 https://www.techtarget.com/searchitoperations/definition/service-mesh

20 https://istio.io/

21 https://www.cncf.io/ wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf

References

[1] C. Dotson, Practical Cloud Security: A Guide for Secure Design and Deployment, O'Reilly Media, Incorporated, 2019.

[2] A. S. a. K. Chatterjee, “Cloud security issues and challenges: A survey,” 2017.

[3] K. R. a. C. W. a. Q. Wang, ***Security Challenges for the Public Cloud***.

[4] N. G. S. L. A. L. M. M. M. F. M. R. & S. L. Dragoni, Microservices: yesterday, today, and tomorrow, 2017.

[5] S. Newman, Building Microservices, O'Reilly Media, 2021.

[6] C. K. Rudrabhatla, “Security Design Patterns in Distributed Microservice Architecture,” 2020.

[7] P. a. D. N. Siriwardena, Microservices Security in Action, Manning Publications, 2020.

[8] M. B. J. a. J. B. a. N. Sakimura, ***JSON Web Token*** (JWT), RFC Editor, 2015.

[9] OWAS, “Microservices Security Cheat Sheet,” [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Microservices_Security_Cheat_Sheet.html#logging. [Accessed 10 11 2023].

[10] B. Muschko, Certified Kubernetes Security Specialist (CKS) Study Guide, O'Reilly Media, Inc., 2023.

[11] S. S. a. I. A. a. T. Dimitriou, “Container security: Issues, challenges, and the road ahead,” Institute of Electrical and Electronics Engineers Inc, 2019.

[12] M. Luksa, Kubernetes in Action, Manning, 2018.

[13] Istio, “Istio Security Best Practices,” [Online]. Available: https://istio.io/latest/docs/ops/best-practices/security/#release-and-security-policy.

[14] R. C. a. Z. Butcher, ***Building secure microservices-based applications using service-mesh architecture***, National Institute of Standards and Technology, 2020.

Acronyms

AKS	Azure Kubterntes Service
API	Application Programming Interface
AWS	Amazon Web Services
Cloud DR	Cloud Disaster Recovery
CNCF	Cloud Native Computing Foundation
CRD	Custom Resource Definition
CSV	Comma-Separated Values
EKS	Elastic Kubernetes Service
ESB	Enterprise Service Bus
GCP	Google Cloud Platform
GKE	Google Cloud Engine
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IBM	Internacional Business Machines
JWT	JSON Web Token
KMS	Key Management System
mTLS	Mutual Transport Layer Security
NAT	Network Address Translation
PaaS	Platform as a Service
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PID	Process ID
RBAC	Role Based Access Control
REST	Representational State Transfer
SaaS	Software as a Service
SLA	Service-Level Agreement
SSH	Secure Shell
STS	Security Token Service
TLS	Transport Layer Security
UI	User Interface
VM	Virtual Machine

Authors

Miguel Almeida

Software Engineer

Altice Labs, Portugal

miguel-f-almeida@alticelabs.com

Ricardo Machado

Software Engineer

Altice Labs, Portugal

ricardo-j-machado@alticelabs.com

https://www.linkedin.com/in/ricardo-machado-b513879a

Mafalda Nunes

Engineer in security and privacy in the development of systems

Altice Labs, Portugal

mafalda-g-nunes@alticelabs.com

https://orcid.org/0009-0009-6085-0385

Telma Mota

Electrical and Computer Engineer

Altice Labs, Portugal

telma@alticelabs.com

Contacts

Address
Rua Eng. José Ferreira Pinto Basto
3810 - 106 Aveiro (PORTUGAL)

Phone
+351 234 403 200

Media
contact@alticelabs.com
www.alticelabs.com