如何设计安全的无服务器架构

CSA GCR cloud security alliance
GREATER CHINA REGION ®

云安全联盟无服务器计算研究的官方网址：
https://cloudsecurityalliance.org/research/working-groups/serverless/

致谢

本文档 《如何设计安全的无服务器架构》(How to Design a Secure Serverless Architecture)由CSA的专家编写，CSA大中华区秘书处组织翻译并审校。

中文版翻译专家组（排名不分先后）

组长：何国锋

翻译组：陈 皓 高亚楠 胡向亮 蒋蓉生 李帅宇 李腾飞 李 岩 刘 芳
茆正华 吴 贺 夏永涛 谢 佳 徐文想 杨文宏 岳 婧 张明敏
周溥璇 左奕航 徐帅健妮

审校组：何国锋 解 佳 姚 凯 郭鹏程

感谢以下单位对本文档的支持与贡献：

北京启明星辰信息安全技术有限公司 北京山石网科信息技术有限公司
北京北森云计算股份有限公司 成都云山雾隐科技有限公司
贵州白山云科技股份有限公司 上海安几科技有限公司
上海派拉软件股份有限公司 浙江大华技术股份有限公司
中孚信息股份有限公司 中国电信研究院安全技术研究所
中国工商银行股份有限公司

英文版原创专家组

团队领导者/作者: Aradhna Chetal Vishwas Manral Marina Bregkou Ricardo Ferreira David Hadas Vani Murthy Elisabeth Vasquez John Wrobel

主要贡献者： 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全球员工:Marina Bregkou Claire Lehnert (Design) Stephen Smith (Design) AnnMarie Ulskey (Cover, Design)

审稿： Anil Karmel Alex Rebo

在此感谢以上专家。如译文有不妥当之处，敬请读者联系CSA GCR秘书处给与雅正!

联系邮箱：research@c-csa.cn； 云安全联盟CSA公众号。

序 言

在当今快节奏的商业环境下，开发人员被要求更迅速地构建和部署应用程序以跟上业务发展需求，同时云原生理念也被普遍接受，基于云原生的无服务器架构因此日渐受到业界青睐。但是像其它云计算解决方案一样，无服务器架构也带来了各种各样的安全风险。

本文全面概括了无服务器平台的各种威胁，重点关注无服务器平台应用程序所有者面临的风险，聚焦于最佳实践，并提供了适当的安全控制建议。文中提出的基于无服务器的安全计算执行模型，对指导应用程序所有者如何采用无服务器架构具有极好的参考价值。

李雨航 Yale Li

CSA 大中华区主席兼研究院院长

目录

执行摘要

无服务器平台使开发人员能够更快地开发和部署应用，用一种简单的方式将应用迁移到云原生服务上，而无需再管理容器集群或虚拟机等基础设施。随着企业努力更快地将技术价值推向市场，无服务器平台正逐渐获得开发人员的青睐。

像任何解决方案一样，无服务器带来了各种各样的网络风险。本文探讨了无服务器应用的安全性，聚焦于最佳实践和建议。本文全面概括了不同威胁，重点关注无服务器平台面临的应用程序所有者风险，并提供了适当的安全控制建议。

从部署的角度来看，采用无服务器架构的组织可以专注于产品核心功能，而不必考虑负载平衡、监控、可用性、冗余和安全方面的工作来管理和控制平台或计算资源。无服务器方案本质上是可扩展的，并为“即用即付”范式提供了大量的最优计算资源。

此外，从软件开发的角度来看，采用无服务器架构的组织可以获得一种全新的部署模型，在该模型下，组织不再需要管理和控制底层操作系统、应用服务器或软件运行时环境。因此，这样的组织可以在更短的时间内部署服务并交付给市场，并降低总体运营成本。

本文的建议和最佳实践是通过在信息安全、云运营、应用程序容器和微服务方面具有广泛知识和实践经验的多元化团队间相互协作制定的。这些信息适用于在无服务器环境中可能负有一定责任的受众。

1. 介绍

1.1 目的和范围

本文的目的是提供实现安全无服务器方案的最佳实践和建议。

无服务器服务涉及两种角色：

- 服务/平台提供者——无服务器平台的提供者，可在此基础上构建无服务器应用；
- 应用程序所有者——无服务器方案的用户，他们的应用运行在无服务器平台上。

基于无服务器方案，服务提供者提供弹性自动分配的计算资源，以满足不同可执行体的需求，不需要用户控制每个可执行体所需的资源。本文的范围限于一个部门基于其他部门作为平台提供者提供的无服务器解决方案来运行他们的工作负载。无服务器的实现既包括基于容器镜像的无服务器（也称为容器即服务 Container-as- service），也包括基于函数的无服务器（也称为函数即服务 FaaS）。

在本文中，我们将更多地关注应用程序所有者，并考虑在使用无服务器服务时对应用的威胁。然后，我们对安全最佳实践提出了具体的建议，并列出了应用程序所有者应采用的建议控制措施。

由于其他材料中有很多关于保护不同云服务的细节，本文将只关注由于迁移到无服务器服务而引发的变化，避免详细阐述通用的云安全。

本文的主要目标是提出并推进基于无服务器的安全云计算执行模型。其目的还在于指导应用程序所有者如何采用无服务器架构。

1.2 受众

本文主要适用于应用开发人员、应用架构师、安全专业人员、首席信息安全官（CISO）、风险管理专业人员、系统与安全管理员、安全项目经理、信息系统安全官以及任何对无服务器计算安全感兴趣的人。

本文假设读者有一定的编码实践、安全和网络专业知识，以及应用容器、微服务、函数和敏捷应用开发知识。由于无服务器领域技术在不断变化，鼓励读者利用其他资源（包括本文中列出的资源）获取最新和详细信息。

鼓励读者遵循与安全软件设计、开发和部署相关的行业标准实践

2. 什么是无服务器

1）无服务器定义

无服务器计算是一种云计算运行模型，在该模型中，云提供者负责为应用程序所有者的工作负载分配所需的计算和基础设施资源。应用程序所有者在任何时间都不再需要确定和控制分配多少计算资源（以及体量）来服务于他们的工作负载。它可以凭借大量的计算资源为工作负载提供按需服务。无服务器计算基于“即用即付”范式提出，在这种范式下，支付通常和实际的物理资源（如 CPU）使用量有关。

使用无服务器的应用程序所有者使用服务提供者提供的能执行（调用、触发）的“可调用单元”，以及一组可调用单元执行（调用、触发）需要的“事件”。应用程序所有者还可以提供支持的代码在可调用单元运行。支持代码可以作为库提供，又称为“层”，可以在运行函数的上下文中运行（作为同一进程的一部分）；也可以作为“扩展”或可以在同一环境中运行的支撑线程/进程（与可调用单元不同的进程）。“层”和“扩展”共享分配给函数的相同资源，并在同样的上下文中运行。

注意，“无服务器”这个名称只适用于使用该服务的应用程序所有者所经历的行为。在底层，执行代码的 «服务器»仍然存在，但从应用程序所有者那里抽象出来。

2) 可调用单元

不同的无服务器解决方案为可调用单元提供了一系列选项。一个常见的服务选项是使应用程序所有者能够在服务提供者支持的运行态（JavaScript、Python、Java 等）中提供函数代码。这种服务在业界被称为“函数即服务”或 FaaS。在 FaaS 下，应用程序所有者提供的函数代码通常嵌入到由服务提供者掌管的容器镜像中。FaaS 的扩展包括给服务提供者镜像安装额外的库，以及在镜像启动和函数执行前向容器注入数据。

第二个常见选项是让应用程序所有者提供其控件的容器镜像作为可调用单元。FaaS 的这种扩展与非无服务器服务有很大的区别，后者的应用程序所有者提供了容器镜像，并在托管或非托管容器服务上执行。下表强调了这一区别。在基于镜像的无服务器模式下，服务提供者负责实现正确数量的可调用单元实例，以响应任何给定时间的事件。

无服务器 (本文涉及)		微服务 非无服务器 (本文未涉及)		
名称	基于函数的无服务器	基于容器的无服务器	托管容器服务	Kubernetes 服务
由应用程序所有者交付可调用单元	函数是否存在依赖	容器镜像		
依赖	特定编程语言	因所有二进制文件和依赖项都已打包，可独立于代码语言运行应用。		
控制可执行体的可扩展性、负载均衡、冗余度和监控实例	服务提供者		应用程序所有者	
控制实例可用性和冗余度	服务提供者控制实例的可用性和冗余度		应用程序所有者	
底层服务器的生命周期和扩展性	服务提供者控制实例的可用性和冗余度		应用程序所有者	
执行时间	通常很短（秒级或更短）。一般限制在几分钟。		通常是长期和无限的。	
状态	无状态和瞬时的——所有状态主要在可调用单元外维护		通常情况下微服务是无状态的，但可以在挂载的卷中维护状态	
扩展计算	服务提供者负责		应用程序所有者负责	
支付模式	即用即付		为已分配资源付费	
运行时责任	服务提供者	应用程序所有者		
例子	AWS Lambda Azure Function Google Cloud Functions IBM Cloud Functions	AWS Fargate Google Cloud Run IBM Code Engine	AWS ECS Red Hat Openshift (基于 AWS/Azure/..)	AWS EKS Google GKE Azure AKS IBM IKS

在基于函数的无服务器（也称为"无服务器函数"）下，镜像由服务提供者拥有和控制，这将保证镜像安全的责任留给了服务提供者。因此，服务提供者必须适当控制，以减轻镜像给应用程序所有者工作负载造成的任何风险（请参见本文 5.3 中的"服务提供者控制的安全风险"）。

在基于容器镜像的无服务器（也称为"无服务器容器"）下，镜像由应用程序所有者拥有和控制，这将保证镜像安全的责任留给了应用程序所有者。因此，应用程序所有者需要适当控制，以减轻从镜像到其工作负载的风险（请参见本文 5.3 中的"应用程序所有者设置阶段威胁"）。

3）事件

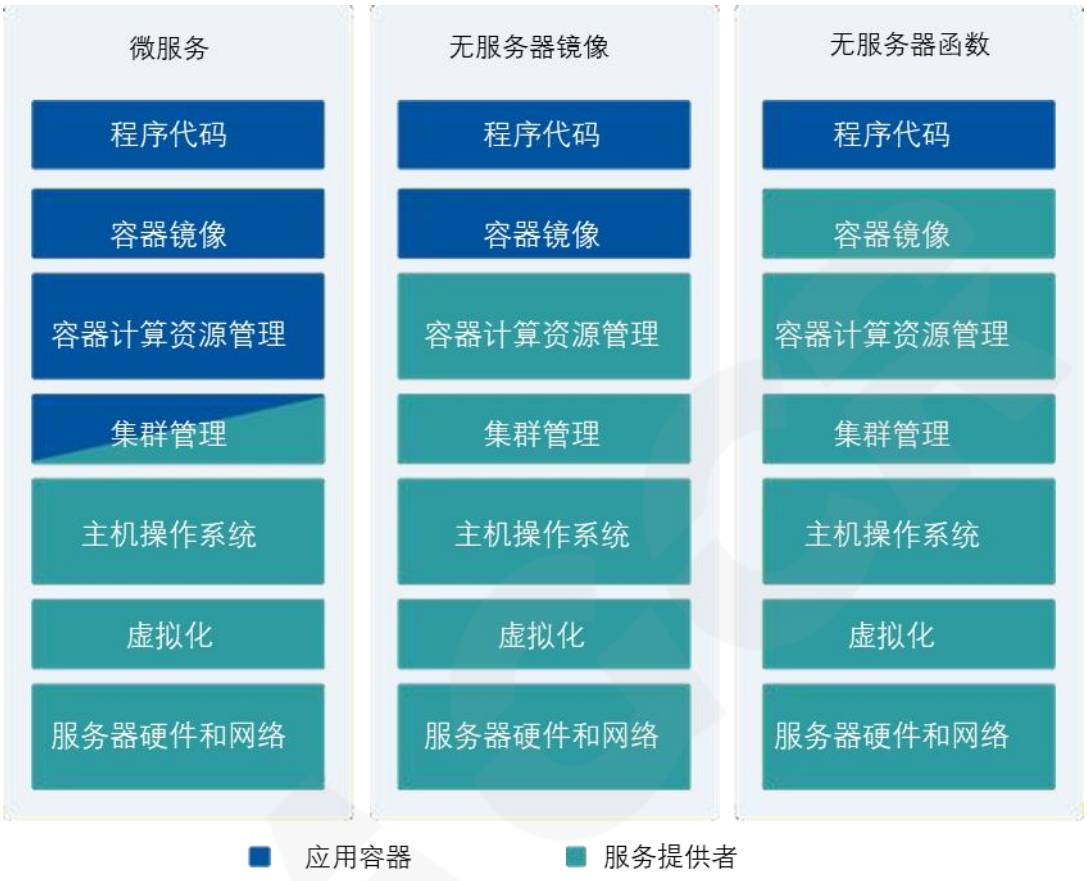
事件是在无服务器环境中可能导致特定函数被触发的条件——它可能包括新的附加数据、接收到的新数据包，或者只是不同周期阶段的过期终结。除了可调用单元之外，事件驱动型应用程序的所有者还可以向服务提供者者提供一组事件，从而定义何时需要执行可调用单元的实例来处理事件。服务提供者负责对事件排队，初始化充足的可调用单元的模型，并根据服务级别协议，在约定的时间内可调用单元处理完成每个事件。事件的实际处理时间被累计用于计费。

事件因事件的来源和事件的类型而有所不同。如定时器事件、web 请求触发的事件、由存储器或数据库中的数据的变化触发的事件、由云帐户的服务之间或用户与云帐户的服务之间的交互触发的事件、来自事件服务的事件、云帐户中的监控服务或日志服务的事件等。

4）无服务器安全概述

无服务器安全带来了一种新的安全范式：应用程序所有者除了保护数据之外，只负责应用的保护。所有管理服务器或其安全方面，包括启动、机器操作系统（OS）打补丁、更新和降级，都由无服务器平台提供者管理，从而使应用程序所有者能够专注于应用本身，而不用再费心于基础设施。但是，正如本文稍后在讨论应用程序所有者需要考虑的威胁时将详细介绍的那样，无服务器也引入了它自己的安全挑战。

为了更好地解释平台提供者和应用程序所有者之间对不同模型的共享责任，我们绘制了下面的图。



比较分担责任模型

无服务器责任模型详图:

基于函数的无服务器(FaaS)共享责任模型

基于镜像的无服务器共享责任模型

由于基于函数的无服务器可以特定于操作/运行时环境，因此平台提供者承担了维护和更新不同版本和编程语言的责任。

5) 混合无服务器架构（私有和公有）

业界存在许多无服务器架构，以下常见的基础设施示例有（部分）：

- 亚马逊：Lambda, Fargate, AWS Batch
- 谷歌：Cloud Functions, Knative, Cloud Run
- 微软：Azure Functions, Azure Container Instances
- Nimbella：OpenWhisk
- IBM：Kortine（代码引擎）、OpenWhisk（Cloud Functions）
- 红帽：Koitine（OpenShift 的一部分）

3. 为什么选择无服务器

与传统的基于云的或以服务器为中心的基础设施相比，无服务器计算有几个优势。[P SEP]

在无服务器计算中，应用程序所有者通常不需要关心托管其应用的基础设施，包括应用运行的操作系统的维护和补丁或基础设施扩容。这能降低大量的运营成本。[Manral，2021]

此外，应用代码是在动态平台上承载和运行的：特定代码可能运行在许多物理机器上，但应用程序所有者几乎无法得知代码具体在哪个物理机上运行。

无服务器平台通常具有动态缩放功能。

3.1 无服务器架构的优势和收益

下表介绍了无服务器架构的几大优势，以方便读者阅读。

无服务器架构可提供的优势	本文讨论的无服务器架构
部署速度	无服务器使应用程序所有者专注开发业务应用，而不用关心应用基础设施，使业务能够快速构建和部署应用。因此，它是一个很好的实验工具，可为市场更快地带来新价值。
成本	基础设施成本： • 通常按每个事件计费，这意味着当不使用基础设施时，不需要付费 • 非常划算的工作负载开销——即便不需要服务器时也不必维护它们，可以提供非常细粒度的资源使用控制。 运营成本： • 没有需要管理的基础设施可以减少运营成本和维护它所花费的时间。[Manral，2021]
应用程序拥有者的体验	容易部署： • 无服务器服务可以通过命令行工具，编程控制或简单的应用程序编程接口（API）以最少化的配置轻松部署。 易于监控： • 大多数云提供商都提供与其无服务器产品捆绑在一起的开箱即用的日志记录和监控解决方案。 该平台由 API 驱动，这对应用程序拥有者的生产力至关重要。 无服务器管理开销： 无服务器服务抽象了所有服务器管理任务，例如补丁、服务开通、容量管理、操作系统维护。
规模	按性质可扩展： • 无服务器根据使用情况在细粒度上自动扩展，只需配置基础设施而无需设置基础设施 • 无需配置扩展或缩减工作负载的策略。 • 在本地工作时，伸缩仅限在于可用的基础设施上。

3.2 无服务器的共享责任模型

在共享责任模型中，开发人员负责保护他们的代码和用于向云交付应用程序的工具。在这个共享的责任模型中，每一方都完全控制他们所拥有的那些资产、过程和功能。

服务	应用程序拥有者	无服务器平台提供商
平台补丁		基于镜像和功能的无服务器。
平台配置	与应用程序相关的应用程序和平台配置。	向应用程序拥有者开放最小配置。
镜像补丁	基于镜像的无服务器容器。	基于函数的无服务器
安全编码实践	基于镜像和函数的无服务器。	
供应链安全	基于应用和组件的供应链。	平台供应链.
网络安全监控		基于镜像和函数的无服务器。
应用程序安全监视	基于镜像和函数的无服务器	
CI / CD流水线配置	基于镜像和函数的无服务器	

3.3 什么时候适合无服务器

当存在相对较大的应用程序或应用程序集以及成熟的软件开发和运营 (DevOps) 团队、流程和产品来支持它们时，无服务器模型最合适。

在这种情况下，应用程序可以分解为称为微服务的较小组件（请参阅《Best Practices in Implementing a Secure Microservices Architecture》）。 每个都支持一个或多个团队，并在无服务器环境中运行。 这允许通过专注于特定功能来更有效地使用开发资源。 与单体应用

程序相比，该模型还允许对每个微服务进行更敏捷的开发，因为应用程序的每个部分的功能都可以转移到生产中，而无需过多关注与其他应用程序功能的完全集成和回归测试。

对于相对较小的应用程序或团队，无服务器模型有时比拥有支持应用程序的传统基础架构（例如 IaaS 基础架构即服务或 PaaS 平台即服务）的成本效益要低。 较小的应用程序通常具有较低的复杂性，并且失去了将应用程序分解为微服务的好处。

在这种情况下，微服务可以与其他服务紧密耦合，可能失去了微服务的一些长处，例如可复用性。 支持微服务的资源不足可能会导致团队需要停止一个微服务以支持另一个微服务。

同样重要并需要注意的是，几乎在所有情况下，无服务器架构可简化部署过程。 部署包括简单地上传一个容器镜像或一组代码，而不像传统的应用程序部署那样关注资源供应和网络架构。 所以企业在决定组织在决定使用无服务器体系结构时，需要执行业务影响分析和成本/效益分析，以选择技术效率最高、成本效益最高、适合其业务需求的解决方案。

4. 使用用例和举例

无服务器计算的关键作用是为云程序员提供必要的工具，这些工具可以通过消除对基础架构的考虑并支持直接使用编程语言来降低云计算架构的复杂性。然而，许多问题需要提前解决，包括负载平衡、请求路由以有效利用资源、系统升级（例如安全补丁）、迁移到新的可用实例以及冗余副本的地理分布以便在发生灾难时保护服务。

无服务器架构的一个使用用例是，在没有操作和扩展基础设施方面广泛技术专业知识的情况下，拥有超大弹性扩容的基础设施。无需维护硬件、操作系统或支持软件，即可构建、部署、管理和扩展无服务器应用程序。这使应用程序拥有者能够专注于业务逻辑，而不是非核心的能力。应用程序拥有者的成本是高度精细的，费用随着使用而线性增长，从而产生一致的经济可行性。总成本取决于运营规模。客户/用户必须根据其需求决定平台规模。

无服务器架构还有一些业务使用用例是:(注意:这并不是一个全面的使用用例的存储库，而只是提供一些示例来，为安全专业人员提供一些参考)。

- Web应用程序:用户需要访问现有的服务并查看或进行较小更新的地方。在该活动完成之后，该功能可能会被删除——基本上是传统的请求和响应工作负载。可以使用无服务器功能，但仍然需要解决错误的身份验证和不可抵赖性等安全威胁。
- 数据处理活动是事件驱动的，在数据处理请求完成后，服务可能会被删除。例如，启动触发器以通过经纪账户拉取一天内为客户购买的所有账户借方或股票的报告。
- 除了身份验证和授权之外，数据完整性、机密性和安全性问题也是无服务器功能需要关注的。
- 批量处理使用用例，通过设置一系列触发器，并构建一个工作流来提取、操作和处理数据。例如，拉出一份未通过碳排放测试的汽车列表，并向车主发送一封电子邮件，其中包含州要求和标准以及满足这些要求的截止日期。 安全问题主要是低权限访问、数据机密性和用户隐私。
- 事件采集和集成:从分析应用程序中收集所有事件，并将它们提供给数据库，用于索引和归档事件，并使用特定触发器启动报告功能或发布到仪表板/浏览器界面。在这种情况下，必须从安全角度解决访问管理和不可否认性问题，并确保使用检测所需的元数据生成日志。
- 无服务器已经在行业中用于安全检测和自动响应，主要用于警告错误配置和采取后续行动。
- 无服务器可用于图片识别和处理。例如谷歌Vision和亚马逊 Rekognition服务，然后根据标识对这些图片进行索引。
- 或者，无服务器也用于构建应用，客户上传信用卡信息，然后提取属性来处理交易。
- 对于这些使用用例，数据安全性、隐私、身份和访问管理问题都需要解决。
- 在一些使用用例中，可以生成触发器，将事件流水线传递给SaaS提供者进行处理。
- CI-CD流水线需要快速迭代软件开发的能力。无服务器可以自动化许多这些过程。代码检查可以触发构建和自动重新部署，或者变更请求可以触发运行自动化测试，以确保在人工审查之前对代码进行了良好的测试。在任何需要自动化的地方，都有可能使用无服务器应用程序简化或加速自动化，并可以轻松地从工作流程中消除手动任务。
- 在物联网传感器输入消息的工业环境中，需要基于可设置的触发器处理这些消息，然后使用无服务器功能响应消息并进行扩展。在某些情况下，不安全功能的影响

可能非常严重。因此，除了故障之外，身份验证、数据保护和检测是这些场景中需要解决的重要控制。

- 如果有一个应用程序需要基于业务逻辑执行特定的步骤:可以使用无服务器功能来实现执行一系列步骤的微服务工作负载的编排。从编排的角度来看，使用无服务器存在几个安全问题；从可审核性的角度来看，触发事件传递的数据/元数据、故障/可用性、以及身份验证和授权之外的不可否认性问题。
- 假期期间的客户服务使用用例，响应客户的查询，即聊天机器人，无服务器提供自动伸缩以满足高峰需求并在聊天结束后删除其功能。
- 从安全角度来看，用户身份验证和数据保护/隐私仍然是需要解决的问题。
- 行业中无服务器的其他流行使用用例是基础设施自动化任务，例如在定时备份等。

从安全团队的角度来看，企业面临的一个重要的问题是，他们可能不知道在他们的平台中所有的无服务器功能都在哪里使用了——考虑到其中一些功能可能是临时的，这就使它变得很困难。以下章节提供了构建控件的详细建议，这些控件将有助于企业中无服务器的可见性和资产跟踪。

如上所述，安全团队可以使用无服务器函数来触发和使用其他无服务器函数以自动化安全任务。检测和响应的使用用例在业界已经很普遍，但安全团队还可以使用无服务器的其他任务，比如CI-CD 流水线中自动扫描发现漏洞时强制中断构建。

无服务器函数可以按需触发对基础架构的扫描，并报告结果或自动执行。

无服务器函数也可用于加强其他安全控制。例如，每当用户上传数据元素时，函数都会检查数据是否已被标记或标签。如果没有，它会向用户生成警报或电子邮件，告知上传的文件没有被标记。然后将其进行隔离，直到标记，以确保适当的DLP或其他合规政策可以执行。

无服务器函数也可用于安全策略的执行，例如在验证对服务的访问时，函数检查可以确保是否落实身份验证。或者还会需要升级身份验证，因此失败的身份验证保证级别有可能触发另一个功能请求升级身份验证。同理，可能还有其它函数用于执行安全政策。

类似地，无服务器在安全领域中的应用也在逐步被探索，并在使用无服务器功能去简化和自动化安全任务这一项上还具有无限潜力。同时，安全团队与开发人员也必须了解对无服务器技术至关重要的威胁概况和安全控制，这些将在后面章节中详细讨论。

5. 无服务器安全威胁模型

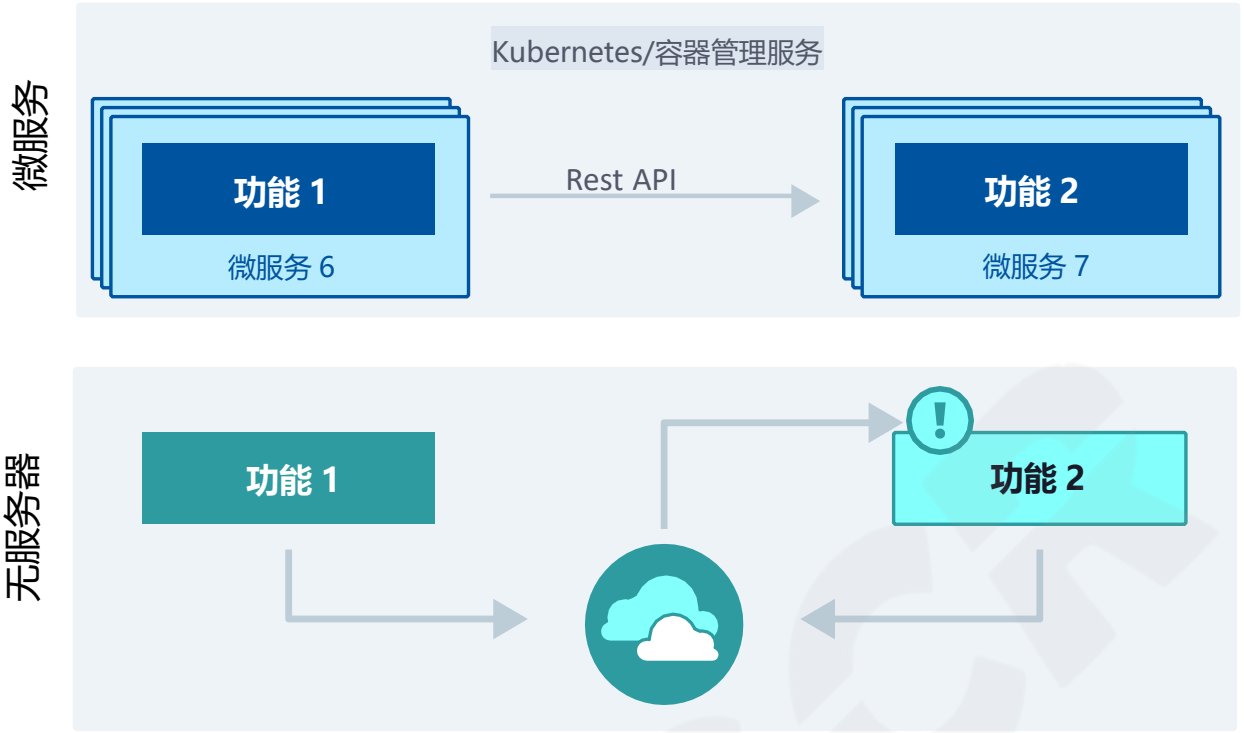
在使用无服务器技术时，应用程序的开发者需要关注多种威胁。本节将讨论无服务器服务的使用是如何以及为何改变了应用程序服务的威胁形势。详细介绍无服务器独特威胁模型及表现方式；以及针对无服务器的特殊安全注意事项及工具。无服务器缓解措施、设计架构及减少安全控制在第6章中介绍。

5.1 无服务器 – 一场全新的安全博弈？

云服务提供者引入无服务器服务，给应用/服务开发商和拥有者带来了全新的安全挑战。无服务器作为事件驱动架构，将工作负载分解成多个无缝隔离的执行环境。每个执行环境都承载着一个特定任务并负责处理一个单独事件，在时间与空间中各自运行，具有自己的依赖项、代码、镜像、权限要求、配置和生命周期。和传统安全威胁模型以及非无服务器的微服务环境中的云威胁模型相比有显著变化。使用无服务器的应用程序拥有者需要重新评估不断进化的威胁形势，并重新考虑所需的恰当安全控制。

无服务器也许包括一些跨越信任边界的流量，例如从公共位置获得数据，将数据输入客户经营场所，调用其它位置的函数等。现有的网络边界正在逐渐消散。这种碎片化也会导致需要大幅增加收集、处理和分析以检测攻击的安全原始数据量。

无服务器是将工作负载深入云中的又一步骤，同时将更多过去由工作负载开发人员拥有和控制的功能转移到云服务提供者（CSP）手中。例如，在以前的传统代码中作为函数调用实现并可能成为微服务中的Rest API，现在转变成了由云服务提供者（CSP）来提交、排队、处理的事件。云服务提供者（CSP）会跟进直至实际数据被功能函数处理完成 --一直到请求者要求处理这些数据之后的某个时候。应用程序拥有者将多项安全控制权放权给服务提供者 -- 这个过程再次要求重构服务消费者的权限和义务，以此来保证基于无服务器的微服务/应用的安全性以及明确由谁来做。



无服务器 — 一场全新的安全博弈

5.2 无服务器威胁形势

5.2.1 主要威胁领域

使用无服务器时对应用程序拥有者工作负载的威胁可以分为以下几类：

1. 应用程序拥有者设置阶段威胁
2. 应用程序拥有者部署阶段威胁
3. 服务提供者的行为威胁

在下述图表中，我们总结了无服务器下应用程序拥有者的主要威胁领域。

威胁形势

5.2.2 应用程序拥有者设置阶段的威胁

我们将与应用程序所有者准备工作负载资产相关的威胁包命名为部署威胁，包括所有必要的代码、镜像、CI/CD工作、云资源的供应和配置等，统称为应用程序拥有者设置阶段的威胁。其中包括了无服务器特有的一组威胁和一组存在于系统中（如微服务）的威胁，但随着时间推移，系统中运行或启动不同的重复代码的预期数量会随着无服务器的使用而增加。

对于无服务器而言，应用程序拥有者设置阶段的威胁的独特性包括由访问权限或配置错误而导致的威胁：

- 广泛且通用的权限
- 广泛且通用的事件访问
- 在无服务器控制上拥有广泛的用户权限
- 弱配置

应用程序拥有者设置阶段的威胁随着向无服务器的转变而加剧，其中包括：

CI/CD流水线或依赖项中与交付流程相关的威胁：

- 可利用的存储库和基本映像注册表
- 针对/通过构建/部署工具的攻击
- 易受攻击的依赖关系
- 易受攻击的基本镜像

错误配置导致的与服务设置相关的威胁：

- 相关云服务的错误配置或漏洞

5.2.3 应用程序所有者部署阶段威胁

本文命名与应用程序所有者部署工作负载资产相关的威胁包，包括所有无服务器关联的云和云外资产、应用程序所有者部署阶段威胁。本文再次区分了无服务器特有的威胁和因使用无服务器而加剧的威胁：

无服务器特有的应用程序所有者部署阶段威胁包括：

可调用单元的设计和实现造成的运行时相关威胁：

- 数据注入
- 全局环境泄漏
- 错误和异常处理不当
- 认证中断或不安全

与无服务器按量计费相关的威胁：

- 财政和资源枯竭（如果已设定资源限制）
- 资源丰富和意外开支。

应用程序所有者部署阶段因使用无服务器而加剧的威胁包括：

- 密钥管理不安全
- 不安全的日志记录/监控
- 日志和元数据中的敏感数据
- 记录/监测不足且不安全

5.2.4 服务提供者的行为威胁

本文将与应用程序使用的服务提供者提供的服务相关的威胁集命名为服务提供者的行为威胁，包括服务提供者使用的整个堆栈，以及用于形成无服务器或相关服务的任何依赖项、个人和其他资产。这些威胁包括无服务器独有的，也包括与其他服务中相似的：

- 易受攻击/恶意的服务基础镜像
- 易受攻击的运行服务
- 可调用单元调用之间的漏洞
- 不同可调用单元之间的漏洞
- 无服务器服务正确性
- API/门户/控制台漏洞

5.3 威胁模型—25项无服务器应用程序所有者面临的威胁

下表列出了无服务器面临的各种威胁。

序号	威胁类别	威胁描述	缓解措施（安全控制）
应用程序所有者设置阶段的威胁（A）			
无服务器面临的特有威胁			
1	**泛化和通用权限** 未维持可调用单元的最小特权原则。	应用程序所有者可以定义无服务器可调用单元在运行时可拥有的特权。 过多权限可能被攻击者利用开展网络攻击。	见章节6, 6.3.5, 6.3.7.1, 6.3.7.2等。
2	**泛化和通用的事件访问** 未维护触发可调用单位的事件的最小特权原则。	应用程序所有者可定义有权限触发可调用单元事件的用户。泛化的访问控制极大的减化了攻击的执行难度，增加了事件驱动的无服务器架构中的攻击面。	见章节6, 6.3.5, 6.3.7.1, 6.3.7.2等。

3	**无服务器访问控制的泛化用户特限** 未维护团队的最小特权原则。	应用程序所有者可定义有权设置无服务器服务、镜像存储等的用户。 更泛化的访问增加了潜在的攻击路径，增加了来自内部人员的风险。	见章节6, 6.3.5, 6.3.7.1, 6.3.7.2等。
4	**不完善的配置** 无服务器配置不完善和配置迁移可能会使平台和所承载应用程序易遭受攻击。	许多用于托管无服务器应用程序的服务的配置不安全。特定的配置参数对应用程序的整体安全状况具有重要影响，应予以关注。例如，谁可以承担执行功能的角色，以及基于这个假定的角色，能实现什么？	见章节6, 6.3.7.2. 6.3.2 6.3.3 6.3.4等。
无服务器面临的通用严重威胁			
5	**相关云服务的错误配置或漏洞** 其他云服务与无服务器服务协同构建的工作负载可能配置错误或易遭受攻击。	通常，可调用单元的安全性取决于所使用的相关云服务的安全性。例如，可调用单元的安全性可能依赖于密码服务或身份和访问控制管理系统等的安全功能。可调用单元也可能依赖于供应链中第三方提供的服务。因此，对其他服务的依赖和这些服务中的错误配置作为无服务器应用程序的一部分，会影响无服务器功能的完整性。	见章节6, 6.3.7.1, 6.3.7.2, 6.3.3等。
6	**可利用的存储库和基本镜像注册表** 用于存储库依赖项和基础镜像的存储库和注册表中存在漏洞。	攻击者可能会尝试通过识别共享（公共或私有）代码存储库和镜像注册表中的漏洞来组合应用恶意代码。 由于独立可调用单元数量的潜在急剧增加，无服务架构面临的威胁也会增加。	见章节6, 6.3.3等。
7	**攻击对抗/通过构建/部署工具** 用于构建和部署可调用单元和事件的 CI/CD 自动化工具的漏洞和错误配置。	作为 CI/CD 实践的一部分，自动化工具通常用于构建、部署可调用单元格（包括将触发它的事件）。这种自动化功能需要通过具有提升权限的工具来存储可调用单元和设置无服务器云服务。攻击者可能会使用提升后的权限将恶意代码合并到目标应用程序中，或者作为一种导致无服务器应用程序更新发生拒绝服务的方法。	见章节6, 6.3.7.1等。

8	**脆弱性依赖** （易受攻击的依赖项） 任何第三方库中的漏洞或恶意代码都可能导致可调用单元发生供应链攻击。	应用程序可调用单元通常使用多个第三方库依赖。此类库可能包括现有或新发现的漏洞。此外，恶意攻击者可能会在此类库中嵌入恶意软件——这适用于所有应用程序和服务，包括无服务器微服务。	见章节 6, 6.1, 6.2, 6.3.7.1, 6.3.7.2等。
9	**易受攻击的基础镜像** 基于镜像的无服务器服务形成的基础镜像中的漏洞。	应用程序所有者在基于镜像的无服务器下构建的基础镜像易遭受预安装依赖项中多种现有或新发现的漏洞的影响，并且还可能包括预安装的恶意软件。	见章节6, 6.3.3等。
应用程序所有者部署阶段威胁 (B)			
1	**数据注入** 无服务器可调用单元在激活期间从各种事件接收输入 - 每个此类事件都代表数据注入相关的潜在威胁。	当不受信任的输入在经过适当审查和验证前直接传递给解释器执行，就会出现注入漏洞。 此类漏洞通常被攻击利用。 大多数无服务器架构提供了大量事件源，可作为数据注入攻击的潜在载体。数据注入事件还会破坏业务函数的执行排序并导致拒绝服务。	见章节6, 6.3.7.2等。
2	**全局信息泄漏** 无服务器全局信息可能引发信息泄露，例如，在可调用单元的调用时会进行令牌维护（例如，无需每次调用都重新验证身份），全局信息可能会在请求时泄漏敏感数据。	由于可调用单元的不同调用通常用于为其他工作负载的用户的数据使用提供服务，因此可调用单元与其他请求之间发生数据泄漏是一种威胁。 敏感数据可能会留在容器中，并可能在函数的后续调用期间暴露。恶意数据可能被故意留下来攻击函数的未来调用。	见章节6, 6.3.7.2等。
3	**错误和异常处理不当** 与标准应用程序的调试功能相比，基于无服务器应用程序的云初始调试选项有限（且更复杂）。	错误处理不当会产生漏洞并允许恶意操作，例如缓冲区溢出攻击和拒绝服务攻击。详细的错误消息可能会导致信息意外泄露给攻击者-- 这适用于所有应用程序、无服务器、元数据和资源。	见章节6, 6.3.7.2等。

4	**损坏或不安全的身份验证** 对事件源的身份和/或发起此类事件的用户/进程的身份验证不正确。	通常，可调用单元需要确认所发送事件背后的实体的身份。攻击者将试图利用所使用的身份验证机制中的任何漏洞。	见章节6, 6.3.7.2, 6.3.5等。
5	**财务及资源枯竭** 攻击者利用无服务器机制恶意占用资源	攻击者可能会利用无服务器的“即用即付”服务机制，通过创建许多调用应用程序可调用单元的虚假事件和/或通过启动导致告警的事件（例如，通过暴露代码或依赖项中的一些其他弱点）来迫使应用程序所有者长时间支付大量的计划外费用。	见章节6, 6.3.6等。
6	**资源冗余** 无服务器机制被攻击者利用作为无限计算资源池	攻击者可能会利用无服务器可作为无限资源池的机制。并鉴于这些漏洞去利用可调用单元可用的大量计算资源并利用它谋取利益，例如，利用相关资源进行挖矿或对某些第三方发起攻击。	见章节6, 6.3.7.1, 6.3.7.2, 6.3.3等。
无服务器面临的通用严重威胁			
7	**不充分且不安全的审计/监测** 对安全事件的态势感知不足，无法监测安全漏洞。	审计不足会妨碍组织对攻击/违规行为迅速做出响应的能力，并使取证分析变得困难或不可能。	见章节6, 6.3.7.1, 6.3.7.2, 6.3.3等。

8	**密钥的不安全管理** 可调用单元使用的密钥泄漏，可能导致对部分系统的恶意访问或权限提升。	通常，可调用单元被触发的时候需要访问特定的云或外部资源。要做到这一点，可调用单位可能需要获取密钥。攻击者可以利用未安全存储的密钥或使用标准凭据集等情况。像任何其他使用无服务器功能的应用程序一样，凭据和密钥的泄漏可能导致虚拟身份和数据泄漏。	见章节6，6.3.7.1,6.3.7.2，等（见#3.过度的数据暴露）

9	**不安全的日志记录/监视** 向攻击者公开日志数据或允许攻击者删除日志。	不安全的日志记录可能使攻击者能够隐藏其攻击行为而难以追踪，或删除其行为的痕迹来阻止发现和取证。同时，功能所有者可能检测不到任何安全问题，也可能不能响应这些问题，从而影响无服务器应用程序的整体完整性。	见章节6，6.3.7.1，6.3.7.2，6.3.3,等
10	**敏感日志记录/监视** 日志记录涉及安全和隐私的敏感信息	可调用单元和事件可能通过日志记录和监视系统暴露敏感数据，包括密钥、个人识别信息、用户数据等。	见章节6，6.3.7.1，6.3.7.2，6.3.3,等
服务提供者行为的威胁（C）			
无服务器的独特威胁			
1	**有漏洞的/恶意的服务基础镜像** 在基于功能的无服务器下，服务提供者选择的基础镜像可能包含漏洞/恶意软件。	服务提供者使用的镜像包括多个第三方依赖项。此类镜像易受现有或新发现漏洞的影响，也可能包括预装的恶意软件。考虑到服务提供者管理平台的底层基础，这可能会影响无服务器应用程序的安全态势。	见章节6， 6.3.3
2	**有漏洞的服务运行环境** 在基于功能的无服务器下，运行环境是由服务提供者的代码配置和扩充的，这可能会导致运行环境存在漏洞。	在基于功能的无服务器下，基础镜像通常会增加额外的代码，以形成服务并监视服务提供者。部署时，运行容器的配置由服务提供者设置—这可能会导致潜在的漏洞，如开放端口或集成到运行环境中的其他管理设施。因此，在完整上下文中评估无服务器应用程序的安全控制措施是恰当的。	见章节6， 6.3.6（Kubernetes的风险和控制:深入挖掘）6.3.7.1等
3	**可调用单元之间调用的泄漏** 在当前无服务器服务合约之外，可调用单元之间隔离失效的，存在的无服务器服务脆弱性。	由于可调用单元的不同调用通常服务于多个工作负载用户拥有的数据，因此调用之间的数据泄漏是一个重大的威胁。例如，一个可调用单元被重用以服务于不同的最终用户或会话上下文，可能会泄漏先前留下的用户敏感数据。或者，先前故意留下的恶意数据/状态可能会伤害后续的用户。	见章节6， 6.3.7.2等。

4	**不同可调用单元之间的泄漏** 在同样的运行环境实例上按顺序执行的可调用单元之间的隔离失效，存在的无服务器服务脆弱性。 例如，函数1结束，而函数2在相同的没有进行适当清理的运行环境上的FaaS下加载。	由于其他可调用单元具有不同的云和数据访问权限，维护不同的身份和密钥，具有各种可利用的漏洞，因此，不同可调用单元之间的泄漏是一个重大威胁。例如，如果重新使用无服务器执行环境来服务一个可调用单元，然后服务另一个（来自相同/不同应用程序所有者）可调用单元，则可能会留下敏感数据，或者故意留下恶意数据/状态来伤害后续用户，利用后续的权限、身份以及密钥或利用后续可调用单位的漏洞。	见章节6， 6.3.7.1等。
5	**无服务器服务的正确性** 在无服务器下，工作负载是由服务提供者整合应用程序所有者的代码片段组合而成的。因此，无服务器核心服务的正确性直接影响工作负载的正确性。	与非无服务器的微服务云服务不同，工作负载的安全性取决于服务提供者的事件系统、镜像管理，和对可调用单元运行实例的访问控制的正确性。当这些系统在某些情况下出现故障，将为攻击者打开了一扇门，这就是威胁。例如，发送错误的事件、启动不正确的功能或组件、提供错误的权限等。因此，除了身份验证和授权之外，在执行前对功能进行编排和对关键事件进行验证也是至关重要的。	见章节6， 6.3.2，6.3.4，6.3.7.1等。
6	**API/门户/控制台漏洞** 有漏洞的API允许用户使用web API、通过CLI或web界面，远程配置和管理无服务器平台。	就管理门户或控制台的而言，可能会导致在多个级别上对平台进行非授权访问，从而带来能修改（并削弱）配置和对环境进行侦察活动的能力。其他应用程序可以通过API调用无服务器应用程序。作为无服务器功能的一部分，通过调用可能增加额外的资源和服务；因此，安全的API，它们的输入输出和上下文对于无服务器功能的整体安全性至关重要。	见章节6， 6.3.6，6.3.1，6.3.7.1等。

5.4 无服务器威胁的独特性

（本节进一步解释了无服务器应用程序环境中的上述威胁）

在本节中，我们将讨论无服务器与其他IT环境相比的一些方面，以及无服务器如何影响前面几节中确定的威胁。我们这样做是为了阐述和强调与无服务器环境相关的特定威胁的重要性。当我们将这些威胁在无服务器环境中的表现形式与其他IT环境中进行对比时，无服务器安全的独特性开始显现。

本节中提到的所有威胁都用粗体标注，并在前面的章节中进行了描述。

5.4.1 应用程序所有者设置阶段的威胁（参考5.3(一)）

在本小节中，我们将详细介绍无服务器环境和其他环境在设置阶段的威胁差异，并识别安全官员和从业人员需要特别注意的方面。

1）碎片化方面

在拥有大量可配置实体和参数的环境中，维护最小特权原则是十分困难的。当使用无服务器服务时，工作负载被分割成小的可调用单元，每个单元都有一组控制其安全性的参数。其中包括用户的凭据和访问权限，允许修改和设置可调用单元的系统、可调用单元在执行时使用的凭据和访问权限，以及与可调用单元相关的日志记录、监视、使用和其他通知的控制手段。当应用程序所有者试图保护自己免受“泛化和通用的权限”、“泛化和通用的事件访问”以及“无服务器访问控制的泛化的用户特限”的威胁时，这给他们带来了新的挑战，

此外，创建和维护多个片段的安全配置也是一项挑战。碎片的数量使处理上述识别出来的威胁变得复杂，例如“弱配置”和“云服务相关的错误配置或漏洞”。

即使应用程序所有者确保了所有组件的安全配置，并在初始部署到生产环境期间使用一系列受到严格限制的权限，由于新组件/服务、依赖关系和更广泛访问的需求等原因，配置和权限集合可能会随着应用程序的成熟发展发生转移。

同时，无服务器提供了一个更加坚固的、结构良好的部署环境。服务提供者首先处理入站流量，然后再将其传播到可调用单元。一般来说，无服务器环境的灵活性不如微服务或虚拟机环境，并且减少了由于“弱配置”带来的威胁。减少威胁的示例包括CSP确保在可调用单元声明的事件之外，没有入站流量到达可调用单元，或者始终使用最新推荐版本的TLS。因此，CSP坚固的环境可以减少攻击面和错误配置的机会。

随着代码段数量的增加，供应链漏洞带来的挑战也在增加。如果没有适当的处理，每个代码段可能会使用一组不同的依赖项。每一个依赖项可能使用不同的代码库和构建/部署工具。当使用基于镜像的无服务器时，每个镜像都可能依赖于一个不同的基础镜像。结果，随着各种依赖项数量的增加，潜在漏洞的数量也在增加，工作负载的攻击面也增加了。这些威胁即上述的：“易受攻击的依赖项”、“易受攻击的基础镜像”、“可被利用的代码库和基础镜像注册“、“针对/通过构建/部署工具进行攻击”。

5.4.2 应用程序所有者在部署阶段所面临的威胁（参考 5.3 (二)）

在本小节中，我们将重点介绍无服务器在部署阶段面临的威胁。

1）数据注入

无服务器会从可调用单元能够处理的各种来源中，提取大量事件加以利用，其中每一种被执行的事件，无论其来源，都是一种潜在的数据注入威胁。

尽管服务提供者可以通过控制特定来源的特定事件（甚至给出所指定的基本事件格式）来提供特定的保护，但这种保护并不能阻止攻击者利用注入缺陷。攻击者可以控制的任何信息（比如，利用漏洞、错误配置、获取的虚假凭据）并将其作为事件的一部分传递给可调用单元，这也应该被视为一种威胁。

举个例子来说，被传递到可调用单元的事件可能来自于“对象被加载到对象存储”的通知。此事件可能包含有关加载对象的信息，这些信息可能受攻击者所控制。类似地，可调用单元的事件还可能来自受攻击者控制的 Web 请求。假设可调用单元直接使用事件携带的信息（没有经过适当的筛选）作为数据库访问请求的一部分。那么在这种情况下，攻击者将会使用精心构造的信息来修改数据库或读取机密信息。

2）全局上下文

在无服务器中，多租户应该是 “按设计”处理的。 可调用单元的每次调用都会处理一个特定的事件，因此可以被认为是特定的租户所为（暂且忽略事件服务范围很广且与特定租户无关的情况）。只要信息在可调用单元的调用之间没有泄漏，这种“按设计”的多租户支持就一直存在。由于云服务提供者可能使用同一个无服务器实例来依次处理多个事件，因此应用程序所有者的程序员需要确保来自一个事件的一次调用信息不会泄漏到后续事件的调用中。

当一个可调用单元/函数实例化时，它可能需要初始化以进行身份验证，获取密钥/令牌，同时建立连接和上下文用于支持服务。对于处理一连串事件的调用队列，此类初始化只需执行一次。可调用单元需要一个全局上下文来维护（缓存）令牌、密钥、开放的连接等。这样的全局上下文对可调用单元来说才是高效的；否则，它在每次事件发生时都需要进行身份验证并与其他系统连接，即使这些事件有序且高速到达,这对于可调用单元的调用函数来说将是一个巨大的开销，而通常这些调用应使用有限且短暂的时间。

引入全局上下文会带来上述的“全局上下文泄漏”威胁，即来自一次调用的信息可能会泄漏到后续调用中。

3）计算资源分配的失控（“即用即付”的危害）

与虚拟机和微服务等其他方式不同，在部署期间，无服务器不会给应用程序所有者使用的计算资源设置上限。应用程序所有者无法控制在给定时间内使用的资源数量，同时需要通过设置预算、预测数量和向服务提供者付费的方式，来重新控制资源。这种特性引发的一个安全问题是，应用程序所有者会在没有适当监控和处理的情况下，面临前文提到的“财务和资源枯竭”威胁。

举一个例子，攻击者可以通过持续性地给应用程序所有者的工作负载施压，给应用程序所有者造成诱发性拒绝服务。即使假设服务提供者为了处理合法和非法事件而提供了足够的资源，但非法事件的持续高负载仍将给应用程序所有者带来预期外的支出，这可能会迫使应用程序所有者关闭或限制服务，这一做法违背了其业务目的和需求。例如，诱发性拒绝服务可能会作为分布式攻击的一部分出现，攻击者通过直接地增加应用程序所有者的云花销或间接地关闭他的工作负载，迫使其承受经济损失。

通过直接经济损失造成诱发性拒绝服务

诱发性拒绝服务也使得“资源滥用”威胁成为现实，攻击者凭借这一威胁，操纵可调用单元来进一步扩大攻击的影响范围。再举一个例子，假设攻击者可以操纵可调用单元来发送流量到所控制的目的端，这些无限资源可以作为针对第三方的拒绝服务攻击的一部分。需要注意的是，无服务器不会限制可调用单元发起的出站流量。阻止此类攻击需要应用程

序所有者进行及时响应，仅通过支付账单是无法阻止的。一旦遭受此类攻击，应用程序所有者需要立即阻止攻击，最好能够在实体受到攻击之前定位到应用程序所有者的可调用单元。可能需要关闭服务来制止可调用单元现有的攻击行为，从而阻止诱发性拒绝服务。

通过损害名誉造成诱发性拒绝服务

第三个例子，假设攻击者可以利用可调用单元执行他们的代码（例如通过供应链攻击或执行EXP或利用漏洞），此时可调用单元可以提供方便且丰富的挖矿资源。如果攻击者把加上攻击负载后的整体工作负载控制在合理的范围内，这种威胁将很难被检测到。攻击者还可以通过执行代码将可调用单元变为通用僵尸网络的傀儡机，攻击事件可触发直至超出可调用单元的时间限制。丰富的资源让僵尸网络的流量隐藏其中，加大检测的难度。在僵尸网络被终止之前，可调用单元可以通过触发可调用单元的一个或多个其他实例来进一步持久化僵尸网络。因为在无服务器架构中出站流量是可以被利用的，所以这类僵尸网络能够采取客户端-服务器的命令和控制架构。

这样的僵尸网络还可以通过重新利用和可调用单元相关联的事件，来使用修改后的点到点命令和控制。注意，一旦可调用单元的代码被攻击者控制，现有的事件可能成为傀儡机之间进行必要通信的通道。

因此，“资源滥用”是无服务器应用程序所有者面临的一个显著威胁，需要适当处理以避免被攻击者利用。

1）碎片化导致的复杂性

无服务器架构促进了碎片化的系统设计。此类架构下的应用可能包含数十个（甚至数百个）不同的无服务器函数，每个函数都有特定的用途。这些函数组合起来形成整体的系

统逻辑。一些无服务器函数通过公开 的web APIs 对外开放，其他函数可能作为进程或其他函数之间的“内部粘合剂”。某些函数可能会使用不同源类型的事件，例如云存储事件、NoSQL 数据库事件、IoT 设备遥测信号以及SMS短信通知。这些因素造成了极大的复杂性，也因此成为了潜在的安全威胁。这些事件是此类身份验证相关漏洞的潜在载体，也就是前文提到的“不可靠的身份验证”和“不安全的密钥管理”。

另一个与系统复杂性相关的例子是，应用程序所有者面临着失去任何态势感知能力的威胁。在复杂和分散的工作负载中，服务提供者控制一部分控制措施和数据路径（例如事件系统），应用程序所有者也控制一部分的控制措施和数据路径（事件处理）。这可能会导致完全失去态势感知的能力，即前文提到的**“日志/监控不充分且不安全”**。

2）代码健壮性和正确性

在微服务和虚拟机场景下，开发人员可以在本地完全复制执行环境，而无服务器与此不同。由于基于无服务器的工作负载的部分数据和控制路径由服务提供者提供，因此需要在云上进行比其他计算选项更多的开发和测试。与标准应用程序的调试功能相比，基于无服务器应用程序的云原生调试选项是有限的（而且更复杂）。当无服务器函数使用基于云的服务（在本地调试代码时不可用）时，这种情况尤其如此。此类威胁即前文提到的“错误和异常处理不当”，可能会通过不必要的详细错误消息或隐藏漏洞造成信息泄露。有一些错误消息模板的最佳实践和标准可以参考；对无服务器部署进行安全角度的测试是有必要的，以验证错误消息是通用的，不会泄露任何数据或元数据。

5.4.3 服务提供者的部署威胁(参考: 5.3 (三))

在最后一节中，我们将考虑来自服务提供者的威胁，并解释那些与无服务器相关方面的问题。

1）隔离

与任何服务一样，服务提供者最关心的安全问题是在云租户之间建立牢不可破的隔离。当涉及到为云用户提供隔离时，无服务器带来了许多新的挑战。其中事件系统和计算系统都是恶意租户的攻击面。这些系统中的潜在漏洞，和与核心无服务器服务相关的许多其他潜在危害一起，被统称为前文定义的**“云服务漏洞”**。

与虚拟机或微服务不同，使用无服务器时服务提供者需要建立新的隔离。由于无服务器的可调用单元频繁地启动和终止（并且由于云用户仅按实际消耗的计算时间付费），因

此服务者正在重用为多个事件服务的可调度单元来大幅减少因实例上下调度而带来的开销。由于不同的事件可以与应用程序所有者的工作负载的不同租户相关联，因此应用程序所有者需要且期望在可调用单元的调用之间进行隔离。这给无服务器带来了一个特殊的威胁，即前面章节定义的**“可调用单元调用之间的泄漏”**。

第二个被定义为**“不同可调用单元之间的泄漏”**的威胁顾及了在同一运行时环境中可能执行的可调用单元之间的隔离问题。根据服务提供者的不同，当FaaS下的一个函数执行完毕后，后续的函数也可能会在同一个运行环境中实现——这为攻击者引入了一个相当大的潜在攻击面：攻击者可以收集先前可调用单元留下的残余数据，并修改运行时环境以潜在地利用后续可调用单元中的漏洞。

2）共同责任制

使用任何来自服务提供者提供的服务都会共同承担起对工作负载安全性的责任。然而，无服务器将共同责任制发挥到了极致。在虚拟机和微服务中，应用程序所有者和服务提供者之间的责任划分在容器或虚拟机的边界上得到了很好的定义，工作负载的执行完全在应用程序所有者的控制之下（不像数据存储、网络等，责任在服务提供者一方）。然而说到无服务器，这个边界就变得十分模糊。

在基于函数的无服务器下，应用程序所有者的代码在服务提供者控制的运行时环境内运行。其安全性可以由代码的设计、运行时的安全性以及两者之间的交互来确定。应用程序所有者的代码可能依赖于一组预先安装的库，并在特定镜像上进行测试。一旦CSP更新了该镜像，代码和新镜像之间的交互就可能会引入可调用单元的漏洞。

此外，在无服务器工作负载下，功能由应用程序所有者代码片段组成，而这些代码片段由服务提供者处理的事件连接在一起。同理，其安全性不仅取决于代码的设计或事件系统的安全性，还取决于两者之间的交互。例如，当应用程序所有者代码以某种方式调用事件系统时。

这些示例旨在说明，与虚拟机和微服务相比，无服务器环境下的共同责任制更为复杂。该威胁即前面章节所定义的“无服务器服务正确性”。

6. 安全设计、控制以及最佳实践

微服务是支持业务领域的独立发布和可部署的服务。微服务可以封装功能，并允许其他服务通过API访问该功能。例如，一个服务可能用于执行会计处理，另一个服务处理账户管理事务，还有一个服务用于报表，但它们合在一起就能构成一个完整的银行账户管理系统。

面向服务的架构是灵活的，只要是可以独立部署的，就不会规定应该如何绘制服务边界。技术不可知是微服务的关键优势之一。微服务的一些主要特点是:

- 独立部署能力
- 围绕业务领域建模
- 拥有自己的状态
- 规模和灵活性

基于领域驱动架构(Martin Fowler)，关于事件驱动微服务(同步和异步事件驱动的微服务)有很多争论。同步服务存在依赖扩展、点对点耦合、API依赖、故障管理、数据访问依赖和管理挑战等缺点。事件驱动的异步微服务因灵活性、技术独立性、业务需求灵活性、松散耦合、持续交付模型等变得流行起来。

虽然行业仍在发展，但也将会出现单一服务和基于功能的事件驱动微服务而组成的混合架构。

事件驱动的微服务的一些架构最佳实践，本质上是使用FaaS服务，描述如下:

接口- API和协议

- 自动重试
- 并发
- 规模需求
- 信息和有效载荷
- 用户和服务/会话标识
- 如何处理故障、超时、重试以及速率限制

无服务器环境带来了许多体系结构变化，而这些变化迫使安全控制的发展。以下是一些例子：

- **事件**
  - 在无服务器模式下，可调用单元通过事件进行通信，这些事件在管理域之间传递。因此，一个可调用单元可能会跨越不受信任的边界向其他可调用单元发送数据。
  - 由于工作负载的碎片化、可调用单元的短周期以及事件驱动体系结构的性质（例如，每次文件更改时，可能会使用多个函数），使得可调用单元被触发的数量增加。
  - 使用链式函数，其中一些函数从多个可信或不可信的来源提取数据。
- **网络**
  - 网络建设由服务提供者控制，服务提供者是目前唯一能够访问网络日志的实体。
  - 网络阻塞点（例如，无服务器平台性能问题）由服务提供者控制。
  - 在新的边界中，任何网络安全控制都需要重新设计，或用零信任模型和身份控制取代。
  - 在传输网络中安全主要由服务提供者负责。使用者必须配置应用程序传输安全性。
- **生命周期**
  - 生命周期截然不同;函数在被销毁之前可能有毫秒的持续时间(有限的生命周期)，这可能会影响安全监控系统。
  - 因为可调用单元快速的打开或关闭，短暂的使用周期可能会影响IAM和秘密管理系统。
  - 要控制和处理的功能/镜像的数量会影响确保完整性所需的控制。
  - 攻击面（详见第5章）
  - 由于无服务器提供的基础架构抽象，传统攻击面正在减少。
  - 云服务实例的数量显著增加，同时也带来了错误配置的机会，需适当的设置IAM角色的复杂性，以及为应用程序开发生命周期中的所有工作负载设置适当的安全流程。
- **监控**
  - 传统上在应用程序（如Nginx/Apache）内部耦合的日志记录和监控等方面现在需要从用户那里抽象出来。这需要重新设计和/或对开发人员提出新的要求，以便在代码中添加适当的日志记录和监视事件。

- 安全控制
- 防火墙、IDS/IPS 和 SIEM 等传统安全控制无法有效应对这种增加连接和集成的新范式。因此，它需要一个更分布式的安全体系结构来保护无服务器组件。
- 无服务器安全性的新方法正在出现且应用程序所有者需要加以重视。我们将在本文的“未来无服务器安全愿景”一章中对这个主题进行更多的阐述。

下面是无服务器体系结构的示意图，其中去周边化非常明显：

无服务器安全视图

上图描述了一组服务（松散耦合），这些服务可能组成一个应用程序，而不必具有传统的安全信任边界，如静态防火墙等。

无服务器范式使用的功能在与第三方服务（例如托管）正确耦合时，允许组织运行端到端的应用程序，而无需关心传统基础设施管理和传统安全的其他方面。

适当的无服务器的安全模式仍在被提出，这要求应用程序所有者采取更周到的安全方法。这些模式需要一段时间才能成熟，本文和该领域的其他论文都是如此。

6.1 无服务器设计注意事项

应用程序架构师必须认识到无服务器技术中存在的固有缺陷和他们在设计中可能引入的缺陷。了解这些缺陷将使他们更好地掌握需要实施的安全控制（这些将在下一章中解释）。

即使从安全角度来看，无服务器体系结构也有许多优点，这些优点可以作为安全微服务设计考虑事项的一部分加以权衡。这些优点包括:

- 无状态且短暂的：短暂的无服务器功能在内存中处理未加密的数据，且处理时间短。无服务器功能不会写入本地磁盘。因此，需要保持状态的功能需要依赖于外部存储，从而降低了被攻击利用的可能性，因为这些攻击都是针对持久目标进行设计的。
- 每个无服务器功能只需要一个数据子集来执行其微服务。因此，只要该功能具有仅访问其所需数据的正确权限，那么成功利用该功能就应该更加关注它可能泄露的数据。
- 无服务器应用运行在由云服务提供者（CSP）管理的容器内或自管理的容器内。 因此，它们像运行在不可变容器镜像上的容器一样，具有一些固有的安全优势。不需要持久服务器的容器可以轻松地持续地给容器镜像和计算实例打补丁。减少在易受攻击或未打补丁的底层基础设施上运行的担忧

架构师和开发人员可能需要考虑的其他因素包括：

供应商锁定：无服务器功能可能正在使用与服务提供者环境不兼容的其他云服务，因为其可能集成了跨CSP环境中的各种服务和依赖项。例如，后端数据库可能位于 Azure的RDS或MS SQL中。

部署工具限制和执行环境限制：根据无服务器应用的需要，了解所需的工具和执行环境并根据这些需要选择提供者是很有必要的。探索可能影响服务整体功能的潜在限制也很重要。

6.1.1 无服务器平台设计对无服务器微服务安全的影响。

如前所述，功能/应用的所有者对管理和控制应用负有更大的责任。他们不太关心平台的基础设施和安全性，这属于服务提供者的责任。随着这种责任的转移，功能/应用的所有者固有地失去了对基础架构的一些可见性和责任。现在，从功能所有者方面来看，出现了如下问题：

- 功能都在哪里运行？
- 功能的实际网络暴露情况如何？

- 在空闲容器中执行的功能是否已经被先前的执行初始化为“热启动”或需要实例化新的容器进行“冷启动”？
- 以前的数据在高速缓存中是否仍旧可用？

让我们从了解无服务器微服务设计过程中所需的一些固有设计特征开始。无服务器功能本身都具有面向公众的出口访问权限。

在无服务器世界中，与云客户相比，CSP 对安全负有更多责任，而云客户 (CSC) 的责任有限。例如，CSC负责其应用内的安全性，负责无服务器组件和服务的安全配置。CSP 默认提供的安全性可能无法满足或解决所有客户的所有安全要求（例如，由于合规性）。CSP 提供的API网关可以在公共互联网上的任何地方访问，访问权限通过API密钥进行控制。客户将负责根据其业务需求设置任何辅助控制（基于传输的访问控制）。

要增强安全性，可设计以下控制措施，以限制数据暴露在互联网上。

- 应用网络策略来限制可从该FaaS访问的端点，这个可通过由各个CSP提供的虚拟私有云 (VPC) 网络策略配置来实现。
- 应用服务和资源策略限制那些可以访问你的数据存储/服务的端点，这样就可以减少数据的渗漏路径。

示例包括 AWS VPC 端点、Azure VNet 服务端点和 Google VPC 服务控制。

按照设计，大型应用程序在分解为微服务时，很难完全了解无服务器微服务部署的架构。

无服务器应用建立在微服务架构之上，允许构建许多以逻辑为重点的功能，这些功能可以并发运行且容易扩展。

然而，随着可用功能的数量不断增加，无论是对于单个应用还是用于更大功能的微服务的分布式API，在创建该功能是如何执行的完全可见性的过程中都会出现挑战。 例如，一些有助于考虑的问题是：

- 是否所有这些功能都在VPC内执行，尤其是在关键数据要求最小化公开披露的情况下？
- 为每个功能创建的角色，是否都调整为仅允许最小权限？
- 开发者是否重用了预先存在的角色，所以需要读取数据的功能只使用与需要处理和更新数据值的功能相同的角色？

- 假设一个现在重新设计的应用指定将功能的启动方式由HTTP事件触发改为由事件队列触发，HTTP事件触发器是否已作为部署选项的一部分被删除了？
- 是否所有能被HTTP事件触发的功能都部署在API网关后面？

关于无服务器环境的一些具体问题-

- 功能的日志和监控不足
- 不安全的无服务器部署配置
- 不安全的应用密钥存储
- 不安全的第三方依赖管理

1) 功能的日志和监控不足：用于详细了解事件驱动微服务/功能的执行环境的控制措施包括：

- 功能的日志记录不足：日志记录提供了实体在给定时间段内进行的活动的记录。它提供对系统/应用的洞察力，并提供在事件发生时进行调试、评估和重放的能力。在无服务器领域记录日志的最佳实践是将记录的值结构化。结构化日志更容易解析。另一个最佳实践是确定所需的日志级别或详细程度，以便日志在需要时能提供有价值的信息。应利用日志来监控异常模式或条件，以提醒和通知操作人员采取纠正措施。确保您使用的是可以集中的集成日志记录，这样就可以方便地对整体应用的性能和安全进行监控。平台提供者的日志记录将有助于收集有关功能执行的数量、持续时间和内存使用情况的统计信息。通过在无服务器功能中根据需要添加的日志语句，可以可视化应用的错误。例如，您的错误日志是否使您能够确定故障是发生在自己定义的进程中还是来自意外的数据输入，或者是第三方函数进程的结果？
- 函数监控不足：使用应用和安全监控工具来帮助了解函数执行的频率和逻辑执行路径。 应该监控和发现公开的所有API或端点，以及所有下游或依赖的API。应该监控并发现哪些事件和角色正在执行应用的函数以及任何意外的执行路径或方法。

2)不安全的无服务器部署配置：平台提供商的默认配置会影响特定微服务用例所需的安全级别：

无服务器为特定的需求或任务（网络策略、命令行界面）提供自定义和配置设置。

无服务器的安全性取决于与无服务器微服务集成的功能和多个上下游服务的配置。您必须了解这些PaaS服务的配置以及所有安全强化选项。 对关键设置的错误配置可能会导致巨大影响。

例如，如果使用PaaS数据服务（例如：Amazon S3、EMR 或 RedShift），那就仍然在使用所有的默认配置，并且该配置仍有可能将数据向公众公开。当您的函数只需要读取和处理数据时，您是否在使用默认平台提供商的IAM角色，这些角色允许您的函数读取和写入数据。

建议的最佳做法：

- 必须根据微服务用例来定义安全设置和策略，并依赖最小化的默认平台提供商配置。根据微服务和数据的安全要求和风险状况来确定如何安全地配置每个服务。
- 无服务器为特定的需求、任务（网络策略、命令行界面）提供定制和配置设置。 对关键设置的错误配置可能会对无服务器产生重大影响。因此，应对微服务进行安全测试，以确保您的配置满足您的所有安全要求。安全测试应该可以帮助您验证那些可以访问您的函数或数据的方法和角色。更重要的是，对应用的安全测试将有助于揭示您的应用程序逻辑或输入验证仍可被注入攻击所利用，以及可能存在的其他配置漏洞。

3)不安全的应用密钥存储

随着应用程序的规模和复杂性不断增长，存储和维护应用程序密钥（API 密钥、加密密钥等）的需求变得至关重要。

开箱即用的CSP产品不能满足所有租户的安全需求。租户应确定其安全级别（PII 数据、敏感数据、法规遵从性要求等）并在默认CSP产品之上升级安全性。租户可以考虑的一些问题是 - 您是否正在设计无服务器应用是否会访问和处理机密数据或受高度监管的数据？如果是的话，那么对数据的安全防护要求就会增加，您不仅要考虑外部攻击者导致的数据泄露，还要考虑内部人员恶意导致的数据泄露。

平台提供商默认在未加密的计算实例上执行无服务器应用。此外，无服务器应用很可能依赖于平台提供商提供的默认未加密的PaaS数据服务。开箱即用的CSP产品不能满足所有租户的安全需求。租户应确定其安全级别（PII 数据、敏感数据、法规遵从性要求等）并在默认 CSP 产品的基础上升级安全性。

建议的最佳做法：

- 确定您的平台提供商是否提供将无服务器代码部署在机密计算实例上执行的选项[IEEE Spectrum, 2020]。这可以帮助您决定是使用服务器功能还是其他计算选项来实施补偿性安全控制，以充分满足应用程序和数据的安全要求以及风险状况。
- 确定包含在您的无服务器应用中每个PaaS数据服务的默认或托管的加密选项，并保存好这些机密或高度监管的数据。确保您对PaaS数据服务的使用和配置将满足您的应用和数据的安全要求。如果不是，则实施应用层加密作为补偿控制。另外，机密计算是昂贵的，随着设计决策的不断演进，必须权衡解决方案的可行性以及机密计算附加的对服务执行性能的影响。

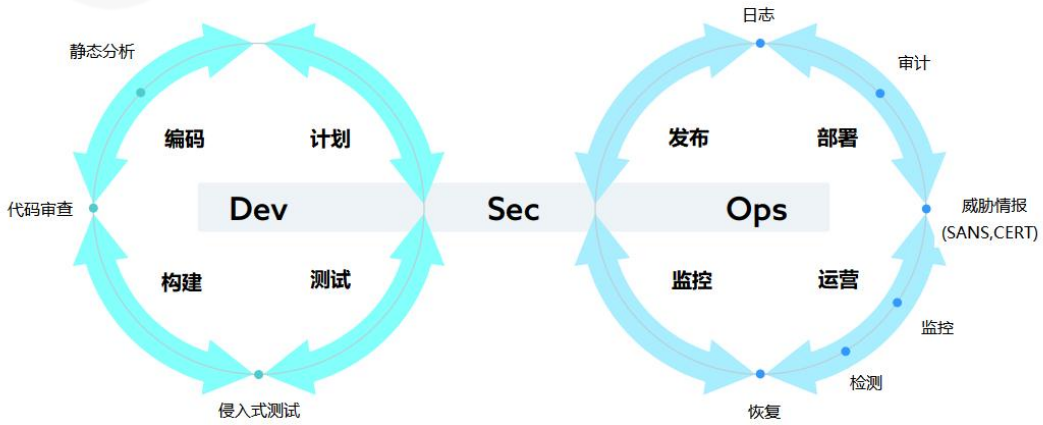
4)不安全的第三方依赖管理：无服务器应用程序依赖第三方库

有时，无服务器函数将依赖于第三方软件包、开源库，甚至通过API调用使用第三方远程Web服务。针对无服务器应用代码和部署的漏洞扫描对依赖的第三方库的可见性有限，尤其是在源代码存储库之外管理的第三方库中引入的漏洞。明智的做法是查看第三方依赖项，因为它们可能存在漏洞，进而使无服务器应用程序暴露在攻击之下。

最佳实践建议：

- 在开发过程中引入对任意第三方库的软件成分分析。通过这种方法，您不仅可以发现大量依赖项，还可以检测是否使用了包含已知漏洞的组件导致风险的引入。
- 在运行过程中使用安全监控系统识别易受攻击的第三方库，并识别项目中包含但未被使用的库。通过这种方法，您可以删除冗余库以降低包含漏洞的风险。

在本节中，我们将简要讨论用于保护无服务器的安全控制和最佳实践。



云服务遵循云服务提供者和云服务客户之间的共享责任模型——对于无服务器也是如此。随着无服务器被各种组织采用，充分理解平台服务提供者与客户在功能和基于镜像模型中的责任尤为重要。在无服务器架构中，云服务提供者负责云的安全（保护服务器和操作系统），租户则通过安全控制负责云内的安全，例如 IAAA（AuthN、AuthZ、审计日志）、SDLC（代码审查、静态分析、构建、测试、发布、部署）、数据保护（静态、动态）、策略执行（例如：代码审查必须由两个对等方进行，代码应在彻底的漏洞扫描后发布）等。

6.2 针对 FaaS 的控制

注意：所有适用于 FaaS 的安全控制也适用于基于容器镜像的无服务器，因为 FaaS 可以架构在基于容器镜像的无服务器之上，处在更加顶层的位置。（本节主要介绍当企业选择自建私有FaaS时，需要实施的安全控制）。

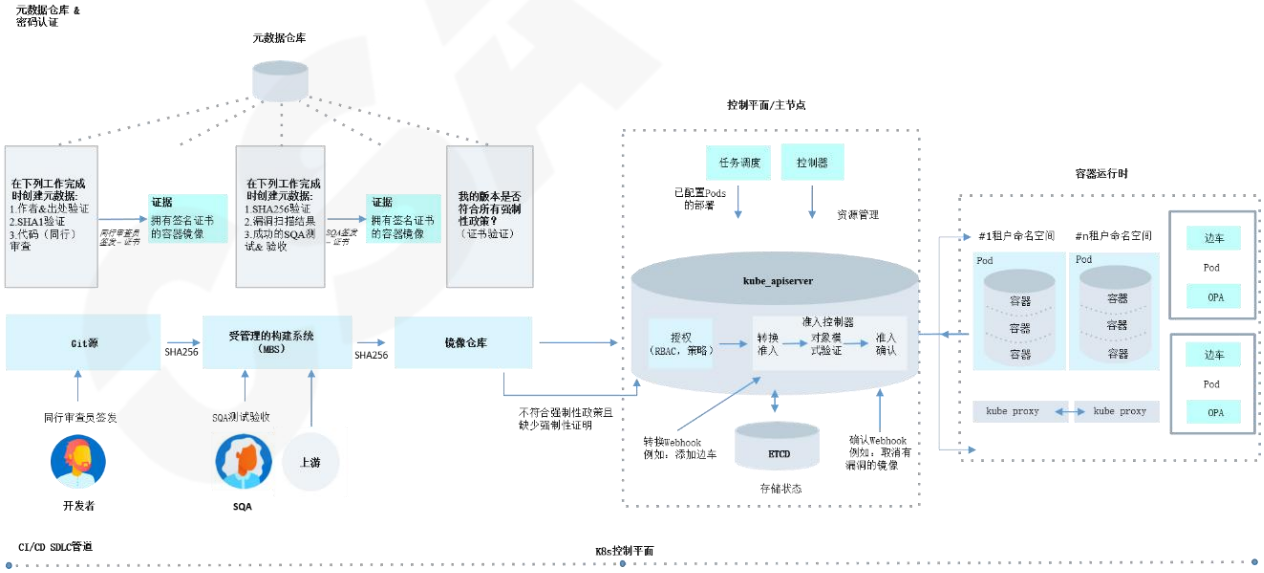
我们正在转向FaaS 模型，目前已经有几个开源的FaaS框架。应用程序团队的安全所有者现在可以将扩展（Lambda 扩展、Knative 等）和代理添加到运行函数的容器中 - 捕获业务逻辑级问题的最佳位置就是在函数内部。安全控制可以通过多种方式来审视：

- 平台配置审计（作为CI-CD流水线的一部分进行代码审计、SAST、DAST、策略实施）。
- 平台组件、漏洞（云提供商承担无服务器平台中的大部分安全责任，即主机操作系统、容器、编排服务、服务网格等）以及网络策略违规检测、平台层的威胁检测和管理。这意味着，如果企业决定自建私有FaaS，他们必须构建所有的安全控制层。
- 功能配置审计（同样可作为 CI-CD 流水线和运行控制的一部分进行代码审计、SAST、DAST、策略实施）。
- 功能组件和漏洞（大多数无服务器漏洞）与编程相关；注入、身份验证破解、访问角色的滥用、部署已知漏洞或不安全的秘密存储、不安全的部署设置、不正确的异常处理以及不充分的日志记录和监视，这些都可以在OWASP安全编码最佳实践中加以限制。
- 身份和访问管理（用户和应用程序身份管理、Federation、SAML 2.0、访问控制、多因素身份验证）。
- 功能工作负载安全（系统完整性监控、应用白名单、应用加固、反恶意软件、漏洞利用的预防、检测和响应）。

同样，如果我们需要定义FaaS的控制类别，我们可以将其划分为以下高级类别：

- 平台服务提供者 API 、管理控制与集成。
- CI-CD 流水线安全控制，包括组件、配置、漏洞扫描等。
- 身份和访问控制管理。
- 平台层检测和针对策略冲突、失败的函数/触发器的运行时检测控制，等等。

6.3 CI-CD 流水线、函数代码、代码扫描以及函数和容器的策略实施



所有的代码变更都需要在发布前经过同行代码审计或手动代码审计或自动化代码审计（静态分析）。 所有更改/更新都需要通过提交一个拉取申请，并且必须由除更改者以外的其他人进行代码审计。当更新通过审计后，将其合并到主分支中，进行测试，随后部署到生产中。

1)静态扫描

随着应用程序的开发和部署速度的提升，在部署之前实现自动扫描“功能或容器代码”变得尤为重要。 分析源码问题的静态分析工具集成到了CI/CD流水线中。拥有防止特定问题传播的门变得至关重要。

要使扫描成功，需要以下几点：

- 跨环境的深入而清晰的可视性—对要扫描的资产进行简单、方便和直观地发现和控制。
  - 这意味着进行代码扫描的能力，不是针对完整的代码库，而是在代码检入后立即对受影响和更改的代码进行扫描。
  - 能够跨代码版本查看所有代码相关库、二进制文件以及库中的更改。
- 扫描基于函数的无服务器和基于容器镜像的无服务器的部署清单，以查找配置中的问题。
- 结果准确性- 需要能够控制误报和漏报以提升准确性。
- 超时验证 – 当函数有时限，且超过该时限函数就会被停止时，将带来问题。静态扫描控制应识别并纠正代码中的问题，例如可能防止代码被突然抢占的部分。
- 易于与开发环境和IDE（例如 Visual Studio、Eclipse、IntelliJ）集成。开发人员使用这些工具来开发函数，而易于集成可以帮助开发人员尽早发现代码中问题并进行修复。
- 支持早期扫描以及定期或在发现新问题时按需重新扫描。这一点至关重要，因为代码组件中的新漏洞会经常被发现。
- 开源问题 – 通过代码扫描列出开源组件和账户中所有开源库的清单。在需要时，这将有助于解决问题。
- 对编程语言的广泛支持——需要同时支持编译型和解释型语言。功能的开发者使用不同的语言，广泛的语言支持对于静态扫描是必不可少的。
- 访问源代码和二进制文件 - 与 git、svn、GitHub、bitbucket 等源代码工具集成
- SDLC 和 CI/CD 流水线与 Jenkins、TeamCity、Bamboo Maven、Ant 等工具的简单集成。这是检查软件动态绑定的最佳位置。
- 与Jira 和ServiceNow 等错误跟踪系统轻松集成。这样可以在扫描代码时轻松集成结果。
- 能够控制工具执行的检查种类； BufferOverflow、SQL 注入缺陷、不安全的反序列化等。

2)配置模板扫描

部署功能不是自动完成的，而是通过基础设施即代码 (IaC) 模板完成的。可以扫描这些模板，如 CloudFormation、Terraform，以确保功能被安全部署，而不会出现导致安全性受损的配置问题。

模板还可以查看函数的输入和输出、函数的拼接方式、端口打开情况，并将其与已知的不良行为进行对比并帮助修复问题。当输入对互联网开放时，需要对代码进行严格检查，并着重提示风险。

3)策略实施

代码模板扫描对于无服务器的安全性至关重要。代码扫描及其相关策略可以在不同阶段实施，并分为两个工作流程：

- 如果开发人员能够轻松地使用代码扫描工具，那么CI/CD流水线中的强制策略将是十分有效的。这有助于开发人员尽快发现和解决问题；然而，在许多情况下，安全团队并不知道CI/CD流水线，也无法对其进行控制——这是好的第一步，但它不能确保所有部署的功能都被扫描。
- 部署控制器—包括CloudFormation控制器、Knative控制器、Terraform控制器。所有被Push的代码都要通过这些控制器。安全团队可以在这里执行对所有函数的扫描策略，该策略与CI/CD流水线运行的位置是独立。

策略可以是强制模式(例如，禁止任何用户访问某个功能)，在这种模式下，它们将阻止所有不符合策略的内容的部署。监视模式是发出警报但依旧可以执行功能的模式。这可能会引发事故，并导致像Jira等CI/CD工具出现问题。

对于不同的环境，如开发、测试、预生产和生产环境，需要不同的策略。与其他环境相比，生产环境可能有更严格的控制，并与其他环境有明确的隔离。

4)对可能访问这些功能的服务和功能应用身份和访问管理控制

由于安全团队对无服务器功能的控制有限，无服务器安全中最关键的事情之一就是理解和管理IAM权限。IAM由以下两部分组成：

- 用于创建、部署和删除功能的用户或服务/角色的IAM
- 已部署的功能/容器服务本身的IAM角色/权限

在动态和快速变化的环境中创建、删除和部署新功能导致IAM的权限管理非常困难，应该执行以下操作:

- 首先，确保IAM角色和权限的可见性(由于变化的速度很快)以保证FaaS的安全性，即每项功能的权限都和部署该功能所需的权限相同。你无法控制你看不到的东西。各项功能均需使用最小权限模型来保证安全性。
- 其次，创建IAM权限保护，即环境中的功能的权限集。部署的任何功能都应该遵循该权限进行设置。任何IAM权限高于此权限的函数都应该被阻止，并通过异常路径执行。

正如上面在策略实施一节中所定义的，可以通过在部署业务逻辑时将控制放在部署流水线中或在部署功能的控制器中来实现功能。

5)网关和接口控制

代码中的外部接口(APIs)增加了攻击面。外部对FaaS和基于容器镜像的无服务器环境的所有访问都通过网关进行。这使得网关对环境的安全性至关重要。

为了保护外部访问，网关应该支持对“OWASP十大安全漏洞”风险的控制和修复。详细信息请参见6.3.7.2 API安全（OWASP十大安全漏洞)。

6)数据保护和与加密/KMS服务的集成，以保护静止和传输中的数据。

根据数据保护法规，特别是通用数据保护条例(GDPR)，FaaS可以利用加密和KMS来保护静止和传输中的数据。在共享责任模型中，数据及其存储是客户的责任。

7)功能编排控制和验证

“步骤函数”有助于协调函数链。安全控件作为必要的一环，确保了如果某个步骤失败，整个链也会失败。

8)检测与响应

- 故障检测
- 违反策略检测
- 威胁/危害检测
- 自动响应
- 运行时检测和策略实施

CSP在无服务器故障检测方面的功能差异很大，一些由第三方供应商提供。好的做法是评估CSP的原生能力，再看还有什么附加价值是第三方能带来的。

6.4 基于容器镜像的无服务器增量/附加控制

云原生计算是一种高度复杂且不断发展的结构。如果没有核心组件来实现计算资源的利用，组织就无法确保工作负载的安全。容器允许开发人员在共享主机上部署多租户应用程序，并根据需要运转和关闭单个容器，从而使开发人员能够更容易地最大限度地利用服务器。所以，为了能够支持这些需求，开发人员需要合适的环境来运行容器。

由于容器提供基于软件的虚拟化，使用特定于容器的操作系统非常重要，该操作系统是只读操作系统，禁用了其他服务，有助于减少攻击面。它还提供了隔离和资源限制，使开发人员能够在共享的主机内核上运行沙箱应用程序。

在基于容器镜像的无服务器平台中，租户可以选择使用自己的操作系统镜像或基于容器镜像的私有无服务器平台。与之相关的是，租户需要确保操作系统安全，遵循安全配置，使用极简操作系统配置，及使用Seccomp来保护系统调用免受滥用 。

Seccomp，即安全计算模式，自2.6.12版本以来就是Linux内核的一个特性。它具有用于沙箱化进程的特权权限，从而限制进程从用户空间到内核的调用。业务流程引擎允许用户自动将加载到节点上的seccomp配置文件应用到各自的Pods（指一组或多个容器）和容器中。同样，在Windows上，Win32k进程黑白名单、Windows文件系统过滤器等功能对沙箱系统调用、Windows进程内核隔离过滤都是非常重要的。

6.4.1 管理基于容器镜像的无服务器服务的API访问

Kubernetes是一个用于部署容器化应用程序的开源编排器。由于Kubernetes完全是由API驱动的，控制和限制可以访问集群的对象及其可以执行的操作是第一道防线。一个Kubernetes集群提供了一个编排API，允许用简单的声明式语法定义和部署应用程序。Kubernetes API提出了部署等概念，以便更容易地执行软件的零停机更新，服务负载平衡，以及在服务的多个副本之间分发流量。此外，Kubernetes还提供了用于命名和发现服务的工具，以便构建松散耦合的微服务体系结构。Kubernetes希望所有API通信都默认使用TLS加密。所有API客户端都必须经过身份验证，即使是节点、代理、调度器和存储插件等基础设施的一部分。这些客户端通常是服务帐户或使用x509客户端证书，它们在集群启动时

自动创建，或作为集群安装的一部分进行设置。一旦经过身份验证，每个API调用也将通过授权检验。

Kubernetes控制平面组件说明请参见Kubernetes.io网页。

6.4.2 基于容器镜像的无服务器配置和策略实施

1) Kubernetes控制平面：

控制平面的主要组件包括Kubernetes API服务器，该服务器提供了用于控制Kubernetes的REST API。对此API具有完全权限的用户在集群中的每台计算机上都具有相同的根访问权限。kubectl是此API的命令行/客户端，可用于向API服务器发出请求，以管理资源和工作负载。对Kubernetes API具有写访问权的用户可以作为根用户控制集群。所有用于API服务器监听请求的端口都必须关闭，并且应该对用户进行适当的用户身份验证和授权。详细信息请参见6.3.5节访问管理控制。通过使用基于角色的Kubernetes访问控制进行配置灵活的授权策略，达到管理Kubernetes资源权限的目的。

Kubernetes API端点应仅能在Kubernetes集群部署的网络中访问，而不是在互联网上公开。Kubernetes API服务器应配置为根据请求的数量进行扩展，以确保集群能够处理较大的流量峰值。

Kubelet作为每个集群节点上的代理，负责与运行时的容器进行交互，以启动pod并报告节点和pod的状态和指标。Kubelets在集群中也有一个API，其他组件可集成并提供启动和停止pod等指令。当未经授权的用户在某个节点上访问此API以在集群上执行代码时，该用户可能获得整个集群的控制权。因而，限制对Kubelet API的访问并通过在API上的——准入控制1中强制执行节点控制来限制Kubelet的权限非常重要。这使得Kubelet将只能修改与其关联的pod。Kubelet还需要客户端证书才能与API服务器通信。这些证书应定期轮换，并可设置为自动轮换。

Kubernetes将配置和状态信息存储在一个名为etcd的分布式键值库中。任何可以写入etcd的用户均可有效地控制Kubernetes集群。即便是阅读etcd的内容，也能轻易地为潜在的攻击者提供有用的信息。所有的Kubernetes机密信息在存储到ETCD之前都应该加密。为了安全起见，机密信息不应仅依赖于静止的ETCD加密。Kubernetes API应该部署在具备高可用性和高容错率环境中。高可用性部署需要满足信息安全中的可用性，确保在遭遇中断或攻击/事件期间仍然可以管理集群，并维持服务质量。ETCD节点应该被配置为仅接受2379

端口上API 服务器的流量，及2380端口上其他ETCD节点的流量。ETCD节点应部署在一个高可用性的负载均衡器后端，以确保节点间的流量均衡，使ETCD节点不会直接暴露给API服务器。ETCD节点的数量应该部署为一个奇数，以便它们可以在集群中做出正确的决策。这是高可用性所必需的。ETCD节点应部署到API 服务器和其他Kubernetes组件以外的单独实例中。为了防止出现问题，Kubernetes 审计、身份认证、API、控制管理器和调度日志应该被存储和监视。

常见问题是重复身份验证失败。使用静态密码的基本身份验证应被禁用。Kubernetes集群应该与企业身份和访问管理集成。 Kubernetes 主机不应该拥有公共IP 地址。Kubernetes主机应在经批准的操作系统上运行。[Kubernetes, 2021]

2) 数据平台

Pod 安全策略应在集群级别定义并由集群中的所有 Pod 使用。 默认情况下，系统不应允许 Pod 作为特权用户运行，也不应允许升级权限。Kubernetes 工作节点不应该被分配公共IP地址，并且应仅在负载均衡器或代理服务器之后提供流量。Pod 被动态授予基于角色的最小权限凭据。 这些凭据应由与后端提供程序集成的 Kubernetes Webhook生成。Pod 不应继承底层主机的凭据。 除非由服务帐户明确分配，否则默认情况下 Pod 不应拥有底层云提供商的特权。

默认情况下，所有 pod 都应被分配一个服务帐户或具有最小权限的默认服务帐户。

维护 Pod 和其中容器的安全是保护容器环境的一个关键因素。这些易受到攻击的单个计算单元可能被用于攻击 Kubernetes 集群。 对于纵深防御，维护最低组件单元级别的安全性可确保在单个应用程序组件上实施更细粒度的控制。 Kubernetes 和其他容器编排系统提供的功能可以使用户加固和保护 Pods，例如 Kubernetes 安全上下文和安全策略（如 Pod 安全策略）。

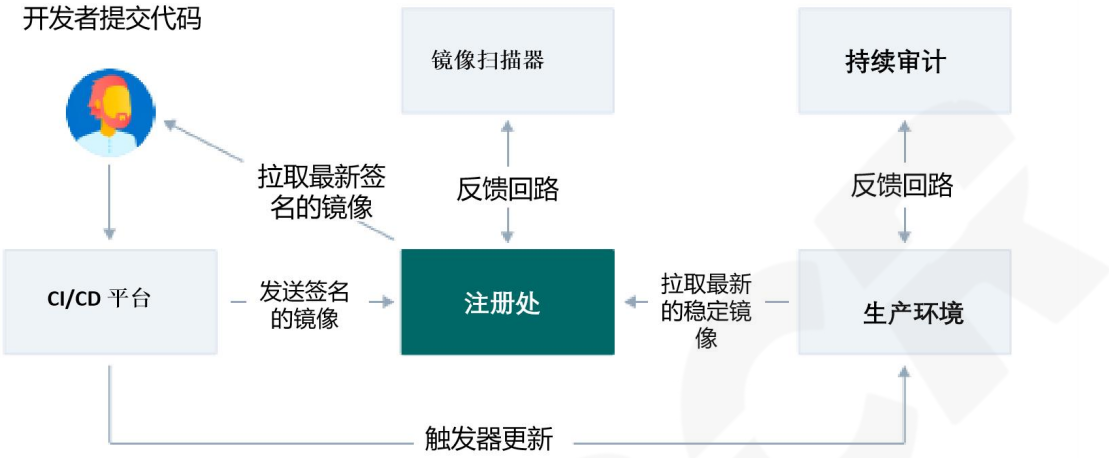
还有其他开源工具，例如OpenPolicyAgent (OPA)，它们也可以强制执行安全策略。

偏离这些政策的特定要求的情况依然可能存在。 因此，必须将安全策略应用于给定 pod 内的所有容器，以便明确单个容器的安全上下文。 因此，安全上下文的字段必须包含在容器清单中； 当存在重叠或冲突时，单个容器的限制将优先于 pod 指定的限制。

6.4.3 基础镜像管理与加固

1)基础镜像

使用容器进行开发工作流程



础镜像是所有镜像的基础，可以被认为等同于基础级别的操作系统。基础镜像可能归云开发团队所有，也可能归组织内的操作系统团队所有。 这些镜像是从各种官方存储库（Docker、Quay 等）中提取的，并存储在组织的镜像注册表/存储库中。这些镜像应作为已批准和授权的版本提供给工程团队，以供在组织内使用。 根据 NIST 800-190，组织应使用特定主机操作系统的极简容器，并尽可能禁用所有其他服务和功能。 组织应根据行业最佳实践强化这些镜像（通过修补和加固），以确保根据强化基准（互联网安全中心 - CIS 基准）仅提供满足业务需求所需的端口、协议和服务。

使用 CI/CD 流水线自动构建、标记和测试基础镜像。系统通过要求开发、测试和安全团队在将镜像部署到生产中之前对镜像进行签名来协助这一步骤。

一些基础镜像安全加固最佳实践：

删除或禁用默认帐户	默认帐户名和密码在攻击者社区中广为人知
配置操作系统用户身份验证	基本镜像应配置为对预期用户/服务进行身份验证，并应与加密一起使用。应尽可能使用 MFA。 企业应实施 TLS 以在不可信网络传输时保护敏感的身份验证信息。
限制访问权限	配置 RBAC 和 ABAC 访问控制并确保所有用户拥有执行其任务所必需的权限-不多不少。

不可否认性与完整性	使用安全存储在公证人处的数字签名来保护容器镜像（通过镜像清单）不被修改。
配置镜像注册处	基础镜像的上游版本应经过安全加固，然后提供经批准和授权的版本供租户工程团队在组织内使用。
解决镜像漏洞	每当发现新的安全漏洞时，都应尽快部署补丁和其他缓解控制措施（基于组织的业务/合规性需求），并立即测试漏洞以确认风险已得到解决。
保护软件供应链的安全	部署流水线的一个重要方面是确保部署遵循组织的批准流程。有一些常用方法如二进制授权、标记、认证等。 用相应的哈希值（commit hash）标记容器镜像可以轻松地将镜像追溯到历史中的特定点。 证明者可以通过在部署流水线的每个关键阶段在容器注册表中添加有关镜像的注释来声明镜像是否满足特定标准。 使用二进制授权时，可以配置一个策略，要求在可以跨集群部署镜像之前对镜像进行特定证明。 如果不满足定义的策略，则可以取消部署镜像。

6.4.4 Kubernetes 配置和服务网格策略实施

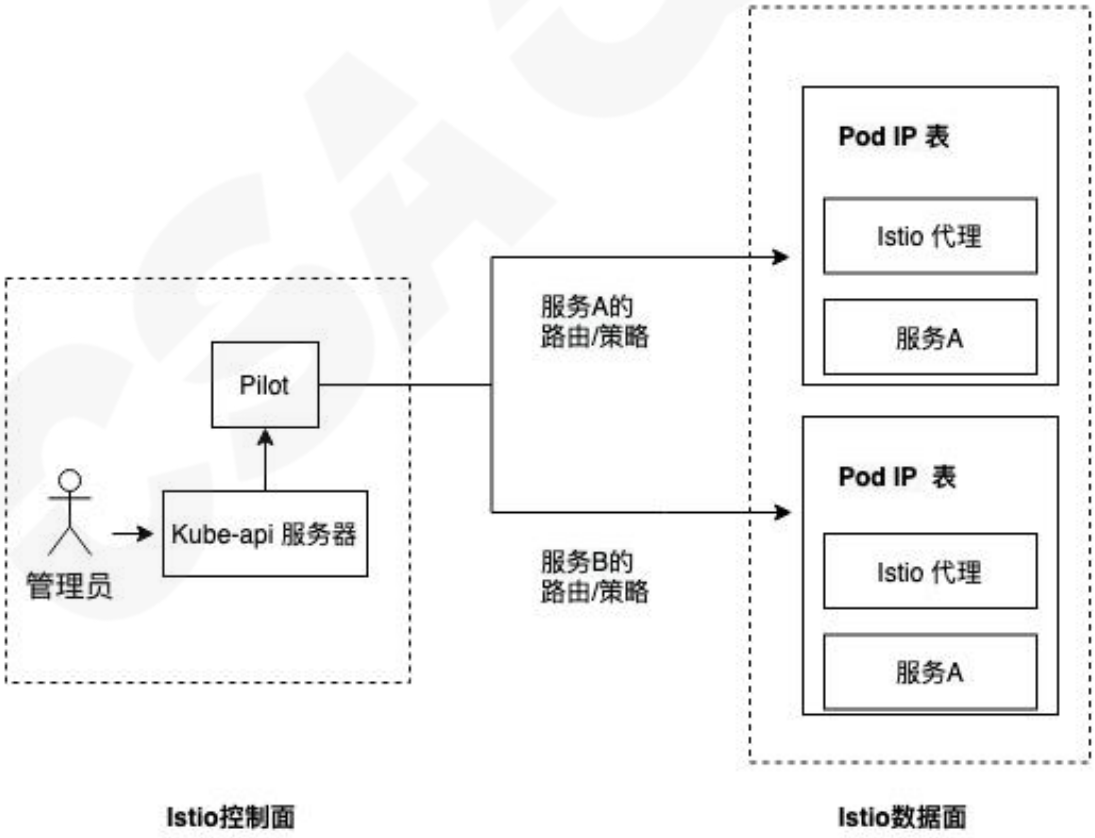
Kubernetes (K8s) 是一个开源编排平台，可自动部署、扩展和管理容器化应用程序。Kubernetes 控制平面将整个集群的状态和配置数据存储在 ectd (一个持久的分布式键值数据存储系统) 中。 每个节点都可以访问 ectd，并且通过它，节点可以学习如何维护它们正在运行的容器的配置。

K8s 控制平面通过 kube-Apiserver 与集群中的组件进行通信。 它确保 etcd 中的配置与集群中运行的容器的配置相匹配。 这是通过声明式 API 执行的。 但是，K8s 没有为配置语言/系统（JSON、YAML）指定标准（或强制要求）

Kubernetes 控制平面的核心是 API 服务器。 API 服务器公开了一个 HTTP API，允许最终用户、集群的不同部分和外部组件相互通信。Kubernetes API 允许您查询和操作 Kubernetes 中 API 对象的状态（例如Pod、命名空间、配置地图和事件）。 大多数操作都可以通过 kubectl 命令行界面执行，该界面通过 REST 调用使用 API。网络策略是 Kubernetes 的一项特性，用于配置 Pods 组如何相互通信及如何与其他网络端点通信。网络策略在 OSI 模型的第 3 层（网络层）和第 4 层（传输层）上运行。Kubernetes 网络策略指定了如何允许 Kubernetes 工作负载组（以下称为 pods）相互通信以及如何与其他网络端点通信。 网络策略资源使用标签来选择 Pods 并定义规则，指定允许哪些流量进入所选的 Pods。

Pod 的 IP 地址不是持久的，并且会随其扩展或缩小、应用程序崩溃或节点重新启动而出现和消失。 Kubernetes 中内置了服务来解决这个问题。Kubernetes Service管理一组 Pods 的状态，允许您跟踪一组随时间动态变化的 Pod IP 地址。服务 充当 Pods 的抽象，并将单个虚拟 IP 地址分配给一组 Pod IP 地址。寻址到服务的虚拟 IP 的任何流量都将路由到与虚拟 IP 关联的一组 Pods。这允许与服务关联的一组 Pods 随时更改；客户端只需要知道服务不会更改的虚拟 IP。

Istio 服务网格是一个安全服务层，为支持的协议提供基于策略的路由和基于策略的授权。 Istio 策略在“服务”（OSI 第 7 层）上运行。服务网格将代理和节点代理容器注入到应用程序的部署规范中，共享 pod 的网络命名空间并透明地拦截流量以应用策略。节点代理将使用 Pod 的服务帐户令牌为代理获取平台证书，并使用代理的密钥发现服务 (SDS) 将证书/密钥提供给代理。对于启用了服务网格的 pods，进出 pod 的所有 TCP 流量将通过作为 pod 中的“sidecar容器运行的特使代理重定向，并且可以扩展以支持外部服务，如下图所示。



使用Istio执行Kubernetes策略

1）Istio内部服务之间相互调用:

这两个服务都位于单个集群中的服务网格内，并作为Kubernetes本机服务部署，Istio控制平面可以自动发现服务的定义和适配代理。

2）外部服务调用Istio:

这表示服务网格外部的客户端（例如curl）或者服务器连接网格内部的服务（如上图所示的服务A和服务B）。要保证通信正常工作，客户端需要使用客户端证书，服务B还必须与Istio 入口网关集成。

3）Istio调用外部服务:

在这个例子中，网格内部的客户端需要启动与外部服务的连接（在图中显示为服务A和服务B）。除了Istio Proxy/Envoy sidecar，应用程序pod还可以包含Github博客中建议的OPAsidecar。当Istio Proxy接收到用于微服务的API请求时，它与OPA进行策略校验以决定是否允许该请求。

6.4.5 访问管理控制

配置Apiserve启用不同形式的客户端身份验证的情况下，同时应使用安全的HTTPS端口（443）上侦听远程连接。并且启用一种或多种形式的授权。

1）认证/授权

控制平台的所有通信都应该使用mTLS(双向认证)。防火墙规则配置为允许外部HTTPS访问API。用户使用kubectl、客户端库或REST请求来访问API。普通用户和Kubernetes服务账户都可以被授权访问API。当请求到达API 服务器时，它会执行一系列检查从而决定是否为请求提供服务，如果服务器确定为请求提供服务，则决定是否通过定义的策略进行进一步验证。请求的授权通常由K8s RBAC授权模块实现。但请求认证也可以通过其他认证方式实现，例如：利用Webhook授权模块，通过OPA(Open Policy Agent)实现高级授权策略

2）访问控制器/Pod安全策略

API请求通过了身份验证和授权后，可以通过访问控制器对资源对象进行验证或更改，然后将其持久化到集群的状态数据库中。访问控制器是一段代码，在对象持久化之前，但在请求经过身份验证和授权之后，拦截对Kubernetes API服务器的请求。

下面提供了一些访问控制器：

- DenyEscalatingExec——如果需要允许pod使用增强特权运行(例如，使用主机的IPC/PID名称空间)，访问控制器将阻止用户在pod的特权容器中执行命令。
- PodSecurityPolicy ——为所有创建的pod应用提供各种安全机制的方法。PodSecurityPolicy对象定义了一组pod必须运行才能被系统接受的条件，以及相关字段的默认值。
- NodeRestriction ——控制kubelet对集群资源的访问控制器。
- ImagePolicyWebhook——允许外部“图像验证器”检查pod容器定义的图像是否存在漏洞。

3）开放策略代理网关（CRD）

Kubernetes允许通过访问控制器使用Webhook将策略决策与API服务的内部工作解耦，在资源创建、更新或删除时通过webhook执行策略。网关是一个验证Webhook，它强制执行由策略引擎开放策略代理执行的基于 CRD 的策略。通过网关的审计功能，管理员可以查看当前哪些资源违反了给定的策略。最后，网关的引擎允许管理员检测并拒绝对系统源代码不合规的提交，进一步加强规范。

4）Istio 服务网格——认证

在Istio中客户端和服务端代理之间通信应使用mTLS（双向认证）。服务端代理使用CA证书对证书进行验证，然后从客户端证书中提取 SPIFFE (Secure Production Identity Framework for Everyone) URI，并将其用作为身份凭证。在服务网格之外让客户端和服务器进行相互通信可以使用上述标准的身份验证机制。在集群、命名空间和服务级别进行配置mTLS(双向认证)的启用。身份验证策略用于在让客户端证书进行强制验证的场景中。Istio还支持其他客户端身份验证方法(例如JWT)。

5）Istio 服务网格——授权

Istio的授权功能为服务网格中启用的服务提供了命名空间级、服务级和方法级的访问控制，它的授权特性是：

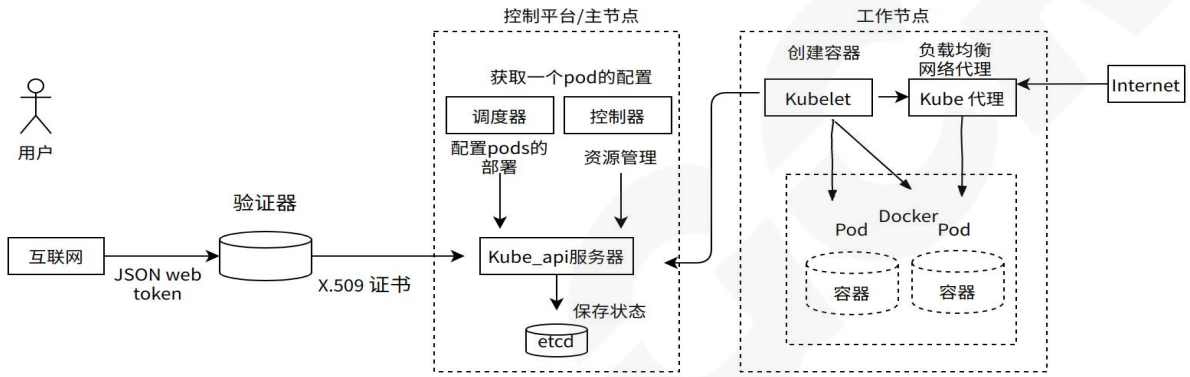
- 基于角色的定义，易于使用
- 高性能操作，因为授权是本机特权代理在本机强制执行的
- 支持HTTP、GRPC协议和TCP连接

命名空间管理员能在命名空间中置服务授权策略。这些策略使用k8s API将策略配置为k8s自定义资源定义（CRDs）。与其他k8s的CRD一样，授权策略是基于命名空间的，服务管理员必须对服务的k8s 命名空间中的策略CRD有写入权限，才能影响服务的授权策略。

当为服务启用授权时，只对策略允许服务的放开访问请求（及 在什么条件下谁可以做什么）。

6.4.6 Kubernetes 风险与控制

1）示例：API 服务器访问



API 服务器是集群内所有通信的中心。在假设某人或某物获得了对API的主动访问权的情况下，大部分集群的安全配置都应用在API 上。在这个例子中，其他人可能获得各种敏感信息，并获得对集群本身的控制。就安全性而言，谨慎管理对API 服务器的访问至关重要。

API 服务器控制所有K8s集群上工作负载的访问和这些负载使用，因此有大量的集群安全配置和应用在API 服务器上。如果不怀好意的人通过K8s API 服务器获得了对特权资源的非授权访问，他们就可以越过k8s RBAC保护策略，并访问任一集群资源上的信息（例如密钥），同时也可以访问集群节点上的任一文件。因此特权资源的谨慎管理对维持一个集群的完整性是十分重要的。部署在Kubernetes上的新应用最基本的一个需求就是通过不同URLs向公网暴露大量的web端点，并且通过SSL保护这些端点。对于端点安全的管理是十分重要的。

下面列举了在API 服务器不被保护的情况下存在的一些漏洞例子：

- 拒绝服务攻击(DOS)是指被攻击者的服务因大量虚假请求而超载。因此，该服务不再能够响应合法的请求。在极端情况下，这种攻击会耗尽整个机器的系统资源(CPU、内存、网络带宽、磁盘I/O等)，对服务造成更大的破坏。
- Netflix在他们的建议中宣布了8个影响HTTP/2的漏洞;CVE-2019-9512“Ping Flood”和CVE-2019-9514“Reset Flood”会影响Go net/http库。任何用Go编写的使用net/http包侦听HTTP/2请求的应用程序都容易受到DoS攻击，包括Kubernetes。
- CVE-2019-9512 “Ping Flood”：攻击者用连续的ping请求流攻击HTTP/2服务侦听器。为了响应每个请求，接收者开始一个接一个地排队响应。这个队列可能会增长，分配更多的内存和CPU，直到应用程序崩溃。
- CVE-2019-9514 “Reset Flood”: 这个攻击和第一个类似，除了它利用HTTP/2的RST_STREAM帧。RST_STREAM只是一种帧类型：它会取消一个应用到另一个应用的连接。因此，DoS攻击可以通过向服务器打开多个流并通过它们发送无效数据来实现 。当服务器收到无效的数据时，服务器发送RST_STREAM帧给攻击者来取消“无效”的连接。有了大量的RST_STREAM响应，它们开始排队。随着队列变得越来越庞大，越来越多的CPU和内存被分配给应用程序，直到应用程序崩溃。
- CVE-2018-1002105：kube-apiserver中对代理升级请求错误响应的错误处理。
- CVE-2016-7054（OpenSSL警告）[高严重性]：TLS连接使用*- chacha20 -POLY1305加密套件时很容易受到DoS攻击，因为它会破坏较大的有效负载。
- 根据我们2018-2019年全球应用和网络安全报告，HTTPS 洪水攻击或利用SSL/TLS的DDoS攻击是2018年最常见的应用层攻击形式。通过使用SSL/TLS，我们可以享受认证特性、完整性和机密性。然而，当涉及到目标服务器时，SSL/TLS连接需要分配大量的资源——确切地说，是请求主机需要资源的15倍。

2）示例：配置错误

Kubernetes 是一个复杂的系统，它提供多种配置选项。在使用这些配置时需要考虑其安全特性。错误的安全配置会暴露敏感的用户数据和系统详细信息，从而导致整个服务器受到损害。安全错误配置通常是不安全的默认配置、不完整或临时配置、开放的云存储、错误配置的 HTTP 标头、不必要的 HTTP 方法等的结果。攻击者将通过这些错误配置来利用这些未修补的缺陷、常见端点或未受保护的文件 和目录以获得未经授权的访问。

缓解策略：

- 建立可重复的强化和修补流程。
- 禁用不必要的功能。
- 限制管理访问。
- 定义并强制执行所有输出，包括错误。
- 确保主机安全且配置正确。根据 CIS 基准检查您的配置。
- 使用Autochecker 自动评估是否符合标准。
- 建立可重复的强化过程，从而快速轻松地部署一个适当的封闭环境。
- Kubernetes 将配置和状态信息存储在称为 etcd 的分布式键值存储中。任何可以对 etcd执行写操作 的用户都可以有效控制 Kubernetes 集群。
- 审查和更新整个API 堆栈的配置。审查应包括编排文件、API 组件和云服务。
- 通过自动化流程持续评估所有设置环境中配置（配置缺陷）的有效性。
- 配置自定义面板和警报，以便更早地检测和响应可疑行为。
- 持续监控基础设施、网络和 API 功能（可能通过自动化）。

3）示例：漏洞扫描

在整个生命周期中扫描镜像的漏洞至关重要。 权衡组织的风险承受能力与开发速度也很重要。 组织需要制定自己的策略和流程来处理镜像安全和漏洞管理。 部署带有漏洞的容器镜像可能会导致威胁向量利用这些漏洞进行攻击。

最佳实践：

首先使用以下指标定义作为不安全镜像的标准：

- 漏洞严重性
- 漏洞数量
- 这些漏洞是否有可用的补丁或修复程序
- 漏洞是否影响错误配置的部署

6.4.7 附加安全

6.4.7.1 Kubernetes 安全最佳实践

Kubernetes 是一个开源容器编排引擎，用于自动部署、扩展和管理容器化应用程序。Kubernetes K8s 集群由托管应用程序的工作节点/Pod 组成。 Kubernetes 控制平台管理集群中的 pod 网络。 我将从下面四个部分介绍 Kubernetes 和容器安全最佳实践：

第一部分

1）无处不在的 TLS

TLS应该应用在每个支持它的组件，以防止流量嗅探并验证连接双方的身份。

2）运行服务网格

服务网格是在高性能“sidecar”模式代理服务器 Envoy 之间建立的加密持久连接网络。Sidecar增加了流量管理、监控和策略（所有这些都无需更改微服务）。

3）使用网络策略

默认情况下，Kubernetes 网络允许所有 pod-to-pod 流量； 流量可以使用网络策略进行限制。

4）使用开放策略代理 (OPA)

使用开放策略代理，您可以在 Kubernetes 对象上实施自定义策略，而无需重新编译或重新配置 Kubernetes API 服务器。

5）日志和监控

您需要日志和监控来检测应用程序和基础架构级别的异常。 如果有任何攻击或异常——高使用率或潜在的危害——日志和监控将检测到它们。

应用程序性能监控将检测其他漏洞，例如 DDOS 攻击。

6）考虑使用堡垒机

这是网络上的专用计算机，专门设计和配置以抵御攻击。 对主节点的访问只能通过堡垒机来完成，堡垒机可以被加固并且可以监控对 Kubernetes 具有特权访问权限的用户。

7）私有网络

Kubernetes 主节点和工作节点都需要部署在私有子网中，以确保与企业网络的安全连接，防止从外网直接访问并减少整体攻击面。

8）使用 Linux 安全功能

Linux 内核有几个安全扩展 (SELinux)，可以配置它们以向应用程序提供最小权限。 Linux 内核有许多交叉的安全扩展（capabilities、SELinux、AppArmor、seccomp-bpf），可以配置这些扩展来为应用程序提供较少的权限。

9）集群节点镜像

当您在 Kubernetes 集群中构建时，您将使用 Linux 镜像。 它需要进行 CIS 基准测试，以确保所有 Linux 安全控制都到位。 不进行操作系统加固过程可能会导致基础设施出现软件漏洞。 考虑到整个软件供应链的安全性也很重要，应确保根据安全最佳实践遵循软件监管链。 二进制授权、标签、证明等是这些如何实现的示例。

10）隔离和保护 etcd 集群

etcd 存储有关状态和密钥的信息，这是一个关键的 Kubernetes 组件 - 它应该与集群的其余部分分开保护。

11）轮换加密密钥

一个安全最佳实践是定期轮换加密密钥和证书，以限制密钥泄露的“爆破半径”。

第二部分 容器安全

1）使用 Pod安全策略

策略是至关重要但经常被忽视的安全部分，它同时充当验证和更改控制器。 Pod安全策略是一个可选的准入控制器，默认通过 API 启用； 因此，策略可以在不启用 PSP 许可插件的情况下启用。

2）静态分析 YAML

如果 Pod安全策略拒绝访问 API 服务器，则应在开发工作流中规范使用静态分析组织的合规性要求或风险偏好。

3）以非 root 用户身份运行容器

以 root 身份运行的容器通常拥有远超其工作负载所需的权限，如果受到攻击，可能会扩大攻击者的攻击范围。

第三部分 自动化安全

1）镜像扫描

所有部署都应通过自动化 CI/CD 环境进行控制。 在高层次上，Kubernetes 中的所有内容都部署为容器镜像。 当有人创建一个应用程序时，它会成为一个容器镜像并被添加到容器注册表中。 容器镜像扫描器需要成为 CI/CD 流水线的一部分。 当有人创建容器镜像时，需要对其进行持续的漏洞扫描。 您可以通过 Kubernetes 中的准入控制器执行镜像白名单。 如果您的应用程序使用某些镜像，则这些镜像需要被批准。

2）密钥管理

您的集群还需要通过密钥管理系统进行集成。这确保应用程序Pod将在运行时根据附加到Pod的应用角色自动接收所需的密码和密钥。

3）代码分析

代码扫描和静态代码分析也是自动化安全的组成部分。在 Kubernetes 中处理任何应用程序代码时，您应该扫描源代码以确保它没有任何漏洞或任何硬编码异常。

4）第三方漏洞扫描

漏洞评估是组织的核心要求。这在使用可能易受攻击的上游组件时尤其必要。第三方漏洞扫描（BlackDuck、Tenable 等）如果持续进行，可以帮助组织领先于威胁代理。这些标准漏洞扫描工具会扫描开源库以查找在各种漏洞数据库（例如国家漏洞数据库）中发布的已知漏洞。

一旦发现漏洞，这些漏洞的补救措施将取决于这些漏洞可能如何影响组织的运营、业务、监管要求或声誉。示例：PCI DSS 3.2.1 要求应在 30 天内修复高/关键类型的已知漏洞。

5）DevSecOps (CI/CD)

安全应该内置到整个 DevSecOps 流程中。提供 DevSecOps 的敏捷流程也必须是安全的，安全用户故事必须在待办事项中。在整个软件生命周期中嵌入安全性以更早地识别漏洞，执行更快的修复，从而降低成本。

持续监控以确保持续发现和验证设备、工具、帐户等。

第四部分：身份和访问管理

1）以最小权限启用 RBAC

基于角色的访问控制 (RBAC) 为用户访问资源（例如访问命名空间）提供细粒度的策略管理。 跨组织集中身份验证和授权（又名单点登录）助力用户的入职、离职和一致性权限。

2）对 API 服务器使用第三方身份验证

Kubernetes - 控制访问：为了保护和管理对 Kubernetes API 的访问，管理员需要创建详细的身份验证和授权策略并实施先进的、功能齐全的筛选技术。

6.4.7.2 API 安全性（OWASP 前 10 名）

API 对于大规模自动化容器管理至关重要。 API 用于：

- 验证和配置pod、服务和复制控制器的数据。
- 对传入请求执行验证并调用其他组件上的触发器。

API 威胁被认为是最常见的攻击媒介。 威胁代理利用 API 中的这些漏洞来破坏应用程序。 请参阅 OWASP Top 10 威胁和缓解措施

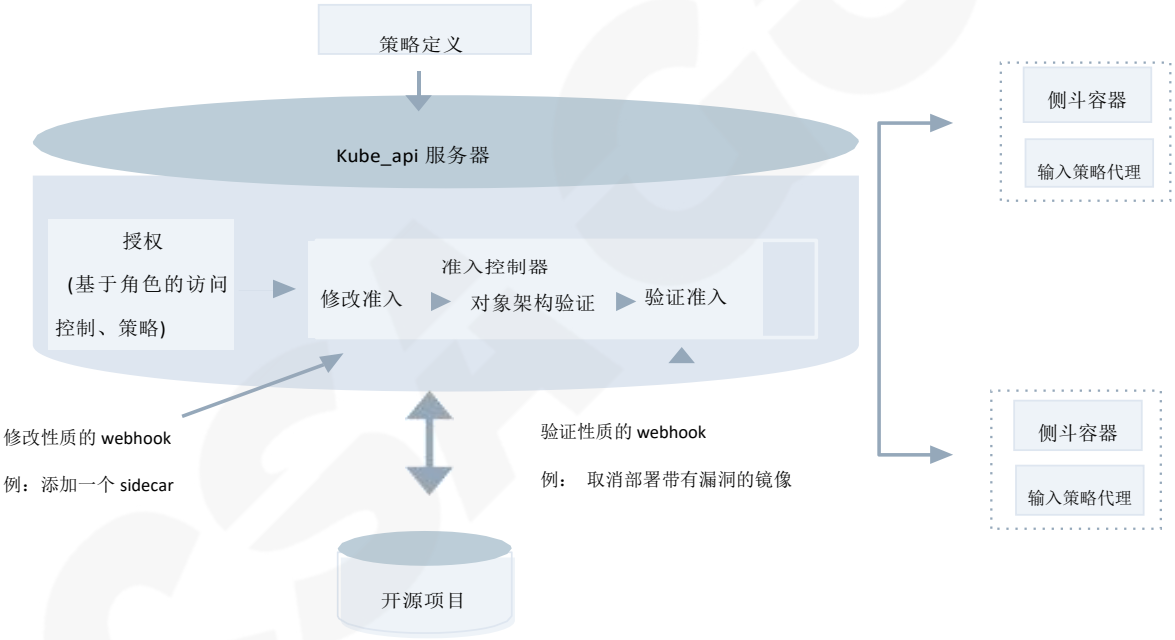
6.4.7.3 Kubernetes 策略

Kubernetes允许通过准入控制器Webhook将策略决策与API服务器的内部工作分离。准入Webhook是HTTP回调，用于接收准入请求并对其进行处理。有两种类型的准入Webhook，验证准入Webhook和变异准入Webhook。变异准入Webhook首先被调用，它可以修改发送到API服务器的对象以强制执行自定义默认值。在所有对象修改完成之后，以及在传入对象被API服务器验证之后，将调用验证准入Webhook，并并拒绝请求以强制执行自定义策略。

开放策略代理（OPA）是一个开源的通用策略引擎，它统一了整个堆栈的策略实施。它提供了一种高级声明性语言，允许您将策略指定为代码，并使用简单的API从软件中卸载策略决策。OPA使您能够在微服务、Kubernetes、CI/CD流水线、API网关等方面实施策略。网关是一个验证（变异）Webhook，它强制执行由开放策略代理执行的基于CRD的策略。网关的审计功能允许管理员查看当前违反任何给定策略的资源。

验证Webhook（链接）的示例：

- 禁止运行特权容器
- 禁止共享使用主机名称空间
- 限制对主机网络和端口的所有使用
- 限制对主机文件系统的任何使用
- 将Linux功能限制为默认设置
- 限制使用已定义的卷类型
- 权限升级到root
- 限制容器的用户和组ID
- 限制分配拥有pod卷的FSGroup
- 策略定义需要seccomp配置文件



Kubernetes策略最佳实践：

- 所有网络连接都应通过政策强制执行。
- 建立远程端点的身份始终基于多个标准，包括身份的强加密证明。
- IP地址和端口等网络级标识符本身是不够的，因为它们可能被恶意网络欺骗。
- 受损的工作负载不能绕过策略执行。
- 使用加密防止向窥探网络流量的实体披露数据。
- 首先应用“默认拒绝所有”网络策略。只允许其他网络策略明确列入白名单的连接。
- 网络策略具有命名空间，并为每个命名空间创建。

- 为了接收来自外部来源的流量，指定应用在pod上的标签，以允许从互联网访问，并创建针对这些标签的网络策略。
- 对于更严格的策略集，理想情况下，您希望指定更细粒度的CIDR块，并明确列出允许的端口和协议。

6.5 合规和治理

当我们利用云计算模型时，组织不仅能够减少硬件、设施、公用设施和数据中心的投资，而且从理论上讲，将此风险转移到CSP应该可以降低总体风险。有一个广泛的假设，即云提供商正在对平台安全和管理这些风险领域进行持续投资，但我们如何知道这一点？对云提供商的性能和服务质量进行持续和定期评估是组织安全保证计划的重要组成部分。组织的采购团队应定期分析和评估云提供商的状态，以确保履行合同义务。

安全团队的重点应该是识别与开发团队部署的无服务器应用程序相关的风险主题。可以对与无服务器计算相关的风险或风险主题进行分析，以更好地了解总体风险。然后，我们可以形成风险声明，用于确定特定风险可能造成的伤害，并开始评估如何应对风险。任何剩余风险都可以记录在组织的GRC系统中。

如果不了解所有资产，就很难保护和治理环境。

6.5.1 无服务器资产管理

资产管理系统需要一种机制来处理云计算的短暂性。云计算的短暂性给资产管理和事故响应带来了挑战。最好的方法之一是将自动资产跟踪集成到CI/CD流水线中。这样，无论何时部署或更改新的无服务器功能，都可以记录、更新相应的更改，并相应地标记资产。实时更新监控、警报和审记操作以及更改。然后，这些信息可以与系统集成，以自动响应并采取行动。

通过这种方式，可以缩小可视性差距，避免在实际事件中出现混乱或错误响应。或者，组织也可以采用轮询方法来收集资产清单。在无服务器中，为组织激活的API每天被轮询一定次数。通过采用两次连续轮询运行的增量，在组织的资产库存中更新信息。

当我们利用云计算模型时，组织可以减少对硬件、设施、公用设施和数据中心的投资，理论上，将此风险转移到CSP应该可以降低总体风险。有一个广泛的假设，即云提供商正在对平台安全和管理这些风险领域进行持续投资，但我们如何知道这一点？对云提供商的

性能和服务质量进行持续和定期评估是组织安全保证计划的重要组成部分。组织的采购团队应定期分析和评估云提供商的状态，以确保履行合同义务。

安全团队的重点应该是识别与开发团队部署的无服务器应用程序相关的风险主题。分析与无服务器计算相关的风险或风险主题可以为了更好地理解总体风险，我们可以检查并使用风险报表，确定特定风险可能造成的危害，并开始评估如何应对风险。任何剩余风险都可以记录在组织的GRC系统中。

由于无服务器计算的性质，开发人员不再需要担心基础设施、网络或主机安全。然而，新的攻击载体已经出现在完整的列表中；请参阅云安全联盟的“无服务器应用程序的12大风险”。

6.5.2 无服务器治理

从治理的角度来看，应该在本文的某个地方提到函数/容器清单，因为它是用户过去不必做的独特的跟踪的事情，需要新的工具/想法。

治理要素：

- 资产清单-利用您的CI/CD流水线、CMDB和CSP维护的资产元数据历史记录的组合，开发您需要保护的资产的完整视图
- 共同责任-客户或应用程序所有者和CSP-利益相关者和监管机构授权
- RACI-责任、问责、咨询、知情
- 指标——我们如何衡量
- 自动化
- 绩效-效率和效果
- 服务协议-SLA-TOR

1）开发无状态的单用途功能：

由于函数是无状态的，并且只能在有限的时间内保持，因此建议为函数编写单用途代码。这限制了对成本有直接影响的函数的执行时间。此外，单用途代码更易于测试、部署和发布，从而提高了企业的灵活性。最后，尽管无状态性可能被视为一种限制，但它提供了对平台的无限可扩展性，以处理越来越多的请求，否则这是不可能的。

2）设计基于推送、事件驱动的模式：

设计基于推送和事件驱动的体系结构模式，其中事件链在没有任何用户输入的情况下传播，从而为体系结构提供可伸缩性。

3）在整个技术堆栈中纳入适当的安全机制：

必须在API网关层和FaaS层纳入适当的安全机制。这些安全机制包括访问控制、身份验证、身份和访问管理、加密和建立信任关系等。

4）确定性能瓶颈：

持续测量性能瓶颈，以确定哪些功能正在减缓特定服务的速度，对于保障最佳客户体验至关重要。

5）创建更厚实、更强大的前端：

在前端执行更复杂的功能，特别是通过富客户端应用程序框架，通过最小化地函数调用和执行时间，有助于降低成本。一种方法是在不影响安全性的情况下将后端逻辑与前端完全解耦。这还允许从前端访问更多服务，从而获得更好的应用程序性能和更丰富的用户体验。

6）利用第三方服务：

无服务器是一个新兴领域，用于各种服务（如日志记录、监控等）的现有企业工具可能不兼容。选择正确的第三方工具来执行手头的任务将是企业确保充分利用无服务器优势的关键。

6.5.3 合规

执行连续的即时监控，以检测并记录是否发生任何漏洞漂移或安全性或合规性漂移。

不注重合规性的组织可能会受到法律问题的威胁。合规流程通常非常详细，需要大量的图表和文档，并向审计员和监管机构解释组织系统的合规性。由于无服务器应用程序无法将物理计算机识别为安全控制的证据，这使合规性变得复杂。

例如，SOX（萨班斯-奥克斯利法案）合规性：

强制的软件工程实践与DevOps政策涵盖了SOX合规性的很大一部分。然而，有一个策略来处理个人身份信息，以实现完全合规，并向监管机构证明这一点，这对于企业来说是

有相关性的。例如，对于AWS Lambda函数，这意味着SOX的重要实践将出现在AWS配置和开发路径策略，以及对资源的控制之中。

同样，对于GDPR（《通用数据保护条例》）来说，该法规的重要元素需要适当的安全实践、数据访问限制、变更控制管理、数据保留和销毁政策——所有这些都是上市公司所期望的，通常作为一个组织努力实现SOX合规的一部分来实施。

7. 无服务器架构安全的未来愿景

正如我们所见，无服务器架构带来了诸多益处与挑战。在本节中，我们将重点关注在未来几年对安全至关重要的领域，如运营、应用安全和加密/隐私。无服务器架构工具依然是个挑战，随着技术的成熟，将与云服务提供者形成捆绑关系。云服务提供者的部署与配置在过去的几年中已经有所改善。然而，由于一些云服务提供者比其他提供商更加成熟，这种体验依旧非常分散。

全面采用无服务器架构技术需要消费者具备在云服务提供者之上创建多云策略或抽象级策略时具有相同的成熟度。云服务提供者还需要将持续交付方案引入到无服务器架构的环境中，以检查和减轻安全缺陷，并提供部署层面的最佳操作实践，如 AB测试、蓝/绿部署和内置在平台内的金丝雀版本。由于无服务器架构增加了基数，因此它需要类似于容器的结构。目前，反思这些系统的工具仍然是一个待解决的问题。

除了实践上的挑战外，还有从安全角度看应用程序的质量及其依赖性。随着无服务器架构技术的普及，关注应用程序运行环境、沙盒、以及通过人工智能/机器学习方法来识别与降低风险的主题将在未来几年不断增长。开发人员通常没有接受过安全方面的培训。因此，新颖的编程语言、更高鲁棒性的软件开发生命周期集成和正式的方法将使开发人员能够在适当的护栏下保持敏捷。

最后，无服务器架构需要在加密和隐私方面突破；尽管大多数云服务提供者采用了针对特定工作负载的安全“飞地”，但它也需要扩展到无服务器架构产品。由于无服务器架构的瞬态和无边界特性，隐私层面和对加密数据进行操作的能力需要更好的隐私框架，在某些情况下，还需要对加密数据进行处理的能力，主流云服务提供者已经发布了几个框架（同态加密库SEAL）。

随着无服务器架构的普及，应用实践将要求并推动云服务提供者在不同环境中更加成熟和一致。应用程序安全生命周期将更加精简，并与云服务提供者集成。DevOps工具和供应链集成都将是主要的关注领域，特别是最近发生了利用这些工具的黑客攻击。最后，尽管是更深远的目标，主流云服务提供者已经发布了隐私领域的研发成果，强调了路线图与他们的关注点。

7.1 无服务器架构的未来之路

7.1.1 向基于容器镜像的无服务器架构发展

如今，所有企业都在拥抱数字化转型，以实现持续增长并获得竞争优势。

容器现在被广泛部署，主要使用Kubernetes编排系统。诸如Kubernetes的云原生技术为大规模管理应用程序和高创新速度提供了必要的自动化、可视化与控制。

对于一个不经意的观察者来说，以Kubernetes为首的自动化似乎最大限度地减少了部署流程中的操作任务，以及在部署和操作过程中。更深入的研究揭示了为什么并非如此。虽然应用程序可以使用Kubernetes配置灵活地部署，但理解副本、扩缩容、污点和容忍并不是许多开发人员最关心的问题。这就给开发人员带来了不必要的复杂性，这也解释了为什么容器的未来是无服务器架构。

基于容器镜像的无服务器架构模型为服务管理者考虑了许多功能的管理，如扩缩容、副本、管理控制平面等，这使得开发人员能够专注于他们的任务。

随着发展，我们将看到越来越多的基于容器镜像的无服务器架构模型为开发者的工作提供更大的灵活性，即使我们正朝着减少开发支出的方向发展。

7.1.2 虚拟化的变革

安全是容器的一大问题。虚拟机在主机上提供了非常强大的隔离功能。然而，容器使用Linux 命名空间来分离、限制与隔离一个容器与另一个容器之间的资源。这种隔离可以被破坏，因为容器无法像虚拟机一样充当安全边界。需要在快速部署的能力、简单的组件评估和强大的安全性之间妥协，这导致了许多不同的操作系统模型：

1）Unikernels

Unikernels是非通用的，由库操作系统构成，并使用单一的地址空间机器。该操作系统大大减少了攻击面和资源的占用。

2）Kata containers

Kata是一个安全的容器运行环境，它的轻量级虚拟机特质和性能与容器一样，但使用了硬件虚拟化技术作为第二层防御，提供了更强大的工作负载隔离。

3）gVisor

gVisor为沙盒容器提供了一个虚拟化的环境，以提高安全性和隔离性。通常由主机内核实现的系统接口被移到一个独立的、基于每个沙盒应用的应用程序内核中，以最大限度地减少容器逃脱的风险。gVisor不会造成大量的固定损耗，在资源利用方面仍然保留了类似进程的模式。

7.1.3 FaaS演变

随着FaaS（功能即服务）应用的增长，也需要为企业的FaaS应用部署提供更大的灵活性、透明度、更强大的安全性和更多的控制。FaaS建立在基于容器镜像的无服务器架构平台之上。

因此，它允许FaaS向不同方面发展，例如：

1）Knative

随着Kubernetes成为抽象层，开源平台Knative为Kubernetes增加了部署、运行和管理无服务器架构、云原生应用程序的组件。这为基础设施管理员提供了在现有基础设施上构建的能力，同时使平台变得更加灵活和透明。这也使得客户可以同样在私有云上运行函数。

2）OpenWhisk

随着FaaS的发展，支持有状态的FaaS成为需求。OpenWhisk提供了一个分布式的无服务器架构平台，可以在任何规模下执行函数以响应事件。OpenWhisk使用Docker容器管理基础设施、服务器和扩缩容，因此你可以专注于构建了不起的高效应用。OpenWhisk允许有状态的函数。这也使得客户可以同样在私有云上运行函数。

3）OpenFaaS

与Knative一样，OpenFaaS是一个利用Docker和Kubernetes构建无服务器架构函数的框架，对数据记录的支持是一流的。任何流程都可以被打包成一个函数，使你能够在不重复编码的情况下，处理一系列的网络事件。

与OpenWhisk相似，这也将允许在私有云上运行函数。

4）Closed FaaS

允许在运行这些函数的容器中运行监控和安全逻辑，以使企业能够控制他们的FaaS部署（即使是封闭式FaaS），这是一种日渐增长的趋势。

7.2 无服务器架构安全

“我们对如何以及何时使用无服务器架构的理解仍处于起步阶段。我们开始看到推荐实践的模式出现，而这类知识只会越来越多。”——Martin Fowler

以上是实用的锦囊和挑战的陈述，因为无服务器架构仍然处于早期的阶段与认知。实践和架构模式将在未来几年不断增长，试图学习最好的方法来更好地开发、部署、监控、交付性能与安全性更佳的应用程序。

1）无服务器架构是面向开发者的，代码优先。目前，在支持跨云计算提供商的语言方面存在着变化与挑战。我们通过使用框架来寻求解决方案。如果没有无服务器架构框架的帮助，配置和部署无服务器架构应用程序所需的一些任务是非常耗时的。开发人员没有必要手动处理这些任务，所以利用框架的优势是十分有意义的。框架将持续发展。加强对语言、持续开发和集成的支持，为部署和实际部署（本身可能需要多个步骤）过程中代码的绑定和打包，设置环境变量、管理密钥、配置终端以将你的无服务器架构服务作为公共API公开，采取并正确设置功能所需的权限，等等。

目前，无服务器架构框架的生态是相当丰富和多样的。一些框架专注于特定的编程语言和/或云服务提供者，一些则是运行环境、云不可知的。最好的办法是转向支持更多的云原生性，支持更多的开发者语言，并成为云不可知的。

2）标准在不断发展，并为无服务器架构环境中的安全控制措施整合做出了努力。

3）日志与调试，开发人员需要对运行代码的平台进行控制，需要由后台进程或守护进程产生的可信数据记录。代码可以通过感知化来实时向第三方服务发送数据记录，但这意味着延迟会影响整体执行。具体的信息对于登录无服务器架构应用程序至关重要，这样

在对安全漏洞采取行动的时候就可以使用这些信息。拥有关键的日志将有助于，例如，了解攻击者利用了哪些安全漏洞，以及如何修复这些漏洞，或建立一个IP地址黑名单，或识别被破坏的客户账户。

为了正确实施无服务器架构应用程序的日志与调试，我们必须重新思考如何对待这些活动。获取所需参数的信息，如调用/事件输入、响应有效载荷、认证请求及其完整的可用性、开放集成，这些都需要不断发展。

4）向有状态和追踪方向发展。无服务器架构函数是转瞬即逝的，并且几乎总是与外部服务互动，主要是因为它们本质上无法保存数据。

无论是数据库、对象存储、数据湖还是其他，函数都需要外部存储来完成依赖于数据有状态性的任务。与函数交互的其他服务可能是消息队列、数据流处理、认证机制、机器学习模型等。

在跟踪无服务器函数的整个生命周期时，对于性能改进、安全监视、调试等运行时执行都是必不可少的。所有这些外部的互动都对此带来了困难。在选择用于检测的跟踪系统时，要考虑的一个关键问题是它的可伸缩性和可用性。根据定义，无服务器平台可以快速扩展并提供高可用性，因此跟踪系统必须能够应对无服务器功能的弹性需求。出于这个原因，专门为检测无服务器函数而定制的解决方案

5）性能:尽管无服务器函数在可伸缩性方面提供了几乎无限的弹性，但这并不意味着它们应该在不受监视的情况下运行。设置性能预期的阈值是非常重要的，这样就可以确定什么时候需要注意一些事情。

一些关键的衡量标准是:

- 调用计数
- 运行时崩溃、应用程序失败、冷启动、重试的计数
- 内存利用率
- 执行时间

当函数没有足够的容器来满足给定时间点的请求数量时，就会出现冷启动。这迫使底层的无服务器平台在请求者等待响应时启动一个新容器——这可能需要几百毫秒到几秒的时间。在许多情况下，这是不可取的。如果我们的应用程序是这种情况，那么我们需要

检测和监控堆栈中的冷启动。云服务通常不会直接提供这些信息，但监控服务需要不断改进。

6）IAM 无服务器功能可以与域中的服务集成，也可以跨信任域集成。这使得基于容器镜像的无服务器和FaaS的信任管理和访问管理变得复杂。随着无服务器使用的增长，具有跨平台和域可用的统一服务信任和用户标识至关重要。业界正在开发SPIFFE和SPIFFE运行时环境(SPIRE)，用于跨平台身份验证和访问管理。SPIFFE还不是一个正式的标准，但正在被批准的过程中。SPIFFE/SPIRE还有助于提高可观察性、监视性和最终的服务水平目标(SLO)。通过规范化跨不同系统的软件标识(不一定是容器化的或云本地的)，并提供标识发布和使用的审计跟踪，SPIFFE/ SPIRE可以极大地提高事件发生之前、期间和之后的态势感知能力。更成熟的团队甚至可能发现，它提高了他们在问题影响服务可用性之前预测问题的能力。

7）供应链 开发人员可能在其函数中使用各种库。使用库可以通过利用和重用现有功能而不是编写库来节省开发人员的时间。这些库可以由第三方开发，也可以是开源的。这通常是无法保证该库是高效的，没有漏洞。该库的开发人员可能使用了其他库。库的数量可能在其供应链的几个层次深处。

当一段代码在其执行的逻辑中使用库时，该库在技术上是一个依赖项。代码层作为代码中的依赖项引入。当依赖项使用其他依赖关系时，这些层会变得更深。漏洞可能存在于任何层或分支，它可能影响无服务器功能，这取决于攻击的严重性和难度。因此，理解依赖树并使其尽可能短而窄是至关重要的。定期检查依赖项的漏洞是维护应用程序安全状态的一种好做法。此外，在Node.JS和最近的Solarigate漏洞等一系列妥协之后，围绕云本地技术和功能的供应链安全，有很多工作要做。这些最佳实践将提供建立供应链软件依赖关系的信任机制，并保护无服务器函数免受第三方库和开源软件依赖关系中漏洞的影响。

与其他技术相比，无服务器仍然处于早期阶段，关于更好地开发、部署和监控应用程序的最佳方法的知识在未来几年将继续增长。(Martin Fowler)

7.3 无服务器在数据隐私方面的进展

由于无服务器将成为组织如何构建、消费和与云原生能力集成的最大驱动因素(由于云提供商希望通过此方式增加粘性，并一直在增加他们的BaaS产品)，因此需要一种机制来保护数据隐私。

在过去的一年里，我们已经看到了来自最重要的供应商的机密计算2的增长，例如（AWS Nitro、Azure机密计算、GCP机密计算）。趋势是，其中一些能力也将被移植到无服务器(Gartner, 2021年)。

随着越来越多的工作负载转向无服务器，数据、状态数据和元数据的加密和处理成为一种需求。我们看到了一些可能会让它成为CSP的无服务器产品的示例:

自我保护: 函数将实时评估和适应每个资源的安全和微分割，甚至使用预测分析(AI/ML)来评估安全态势，并定义新的警报机制。

预定义的兼容函数: 开发人员可以根据所建立的屏障和需要使用的数据类型来利用和使用的一组函数。这可能是设置护栏的自然扩展，护栏是一种针对无服务器的目录。这样做的好处是可以在不降低敏捷性的情况下创建护栏。

FaaS的安全飞地: 随着一些受监管行业希望从云环境获得的审查和保证，组织需要利用机制来遵守不断增加的法规。

我们已经看到，可信执行环境被广泛用于维护数据、代码和计算的机密性和完整性。这方面的一个例子是与全球相关的《欧洲通用数据保护条例》(EU GDPR)。这些机制将受益于在Faas功能中提供可用的云环境，因为它们开始在更传统的IaaS环境中使用。

该技术将允许代码、数据和处理更好地与其他客户和CSP隔离。当它包含个人或敏感数据时尤其重要。

已经有一些库(如IEEE Xplore3, Graphene4)提供飞地作为一种服务，并包括进入FaaS的自然扩展

8. 总结

几乎所有行业的IT组织都感受到了更快交付价值、在竞争中领先并快速为客户提供新体验的压力。无服务器平台允许应用程序团队交付价值，而无需管理应用程序所运行的基础设施。随着这一运动的发展，我们将看到无服务器平台的激增，更多高价值的应用程序被放在这些平台上。从这里开始，对无服务器平台的安全担忧将会增加。

作为CSA无服务器工作组，我们已经尽可能收集了无服务器安全性的所有方面，包括基于容器镜像的无服务器平台和FaaS平台。当然无服务器架构也在不断发展中，当我们在无服务器平台上看到更重大的变化时，我们将提供该文档的新修订版本。

9. 参考文献

[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*. *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.* OWASP API https://owasp.org/www-project-api-security/
[PCI, 2018]	PCI. Security Standards Council. (2018). DSS 3.2.1 Standard. Requirementsand 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 Boundariesin SOA. https://rvsoapbox.blogspot.com/2011/07/service-boundaries-in-soa.html

附录1:首字母缩略词

本文选用的缩略语定义如下。

ABAC	Attribute-based access control 基于属性的访问控制
API	Application Program Interface 应用程序接口
BaaS	Backend as a Service 后端即服务
CICD	Continuous integration, continuous delivery 持续集成，持续交付
CISO	Chief Information Security Officer 首席信息安全官
CPU	Central processing unit 中央处理单元
CRUD	Create, Read, Update, Delete 创建、读取、更新、删除
CSP	Cloud Service Provider 云服务提供者
DevOps	A portmanteau of “development” and “operations.” “开发”和“运营”的合成词。
DLP	Data Loss Prevention 数据丢失防护
FaaS	Function as a Service 功能即服务
GDPR	General Data Protection Regulation 通用数据保护条例
HIPAA	The Health Insurance Portability and Accountability Act of 1996 1996年健康保险可携带性和责任法案
IaaS	Infrastructure as a Service 基础设施即服务
IT	Information Technology 信息技术
KMS	Key Management System 密钥管理系统
NIST	National Institute of Standards and Technology 国家标准与技术研究所
OPA	OPA (Open Policy Agent)是一个开源的通用策略引擎，它统一了整个堆栈的策略执行。OPA提供了一种高级声明性语言，允许您将策略指定为代码和简单的APIs，从而将策略决策从软件中卸载出来。
OS	Operating System 操作系统
PaaS	Platform as a Service 平台即服务
PCI	Payment Card Industry 支付卡行业
SLO	Service Level Objectives 服务级别目标
SPIFFE	Secure Production Identity Framework for Everyone 每个人的安全生产身份认证框架
SPIRE	SPIFFE Runtime Environment SPIFFE运行时环境

STS	Security Token Service 安全令牌服务
VNet	Virtual Network 虚拟网路
VPC	Virtual Private Cloud 虚拟私有云

附录2:术语表

服务边界	服务边界是由服务提供的功能的声明性描述定义的。在其边界内的服务拥有、封装和保护其私有数据，并且只选择在边界外公开某些(业务)功能。