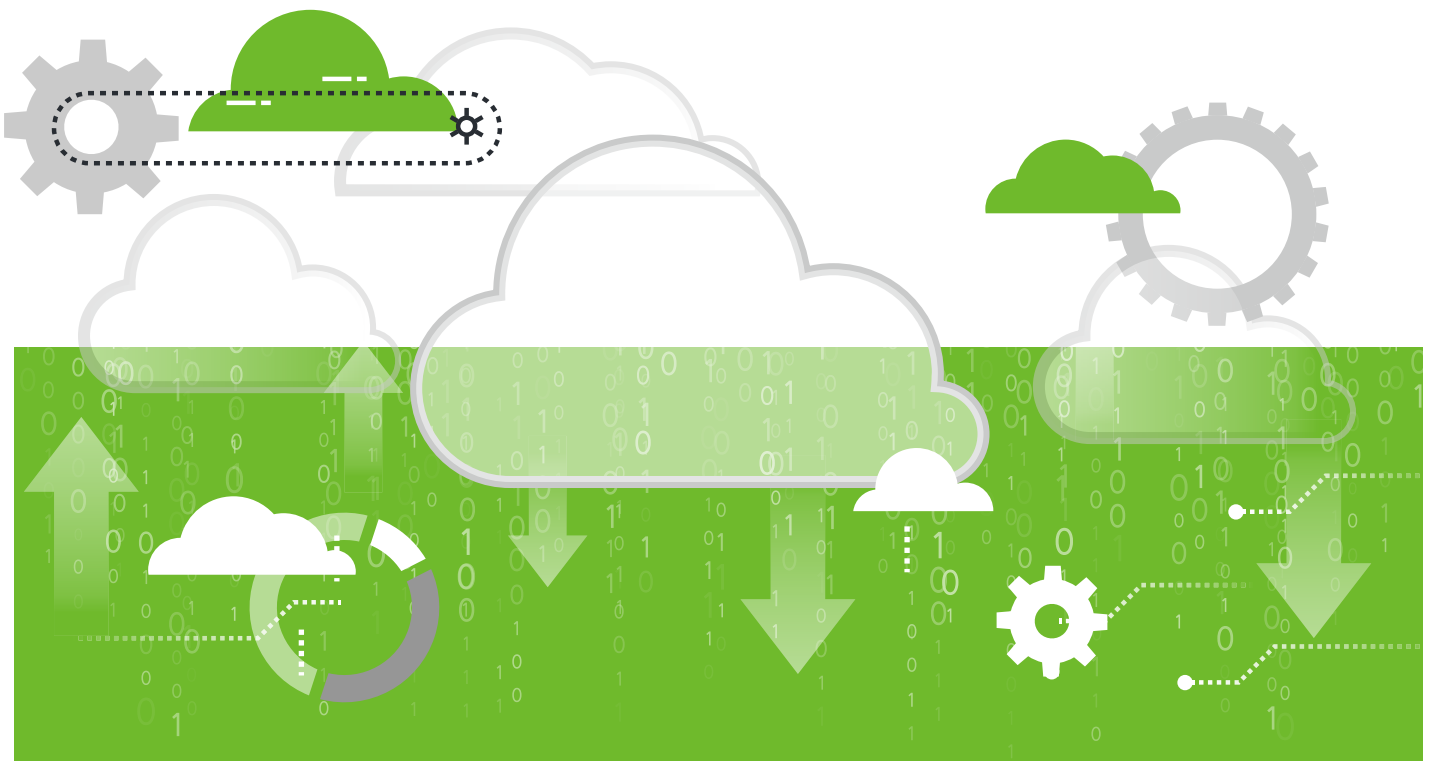
2020 | 巨人背后的专家

云原生安全技术报告

NSFOCUS 移动云



关于绿盟科技

绿盟科技集团股份有限公司 (以下简称绿盟科技), 成立于 2000 年 4 月 , 总部位于北京。公司于 2014 年 1 月 29 日起在深圳证券交易所创业板上市 , 证券代码 : 300369。绿盟科技在国内设有 40 多个分支机构 , 为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户 , 提供全线网络安全产品、全方位安全解决方案和体系化安全运营服务。公司在美国硅谷、日本东京、英国伦敦、新加坡设立海外子公司 , 深入开展全球业务 , 打造全球网络安全行业的中国品牌。

中国移动云能力中心

中国移动云能力中心是中国移动通信有限公司的全资子公司，公司定位为中国移动云设施构建者、云服务提供者、云生态汇聚者。公司以移动云运营为中心，产品和服务在政务、金融、制造、交通、医疗等行业得到广泛应用。

版权声明

目录 CONTENTS

前言

网络安全发展的方向有两种趋势，一是与新场景、新应用结合，典型的如云计算、人工智能、物联网等；二是安全技术自身的演化，例如欺骗技术、威胁狩猎、智能决策等，而这两类趋势往往可能会融合，如人工智能与安全技术之间，既利用 AI 技术赋能安全运营形成 AISecOps，又加速了智能化安全运营和智能化攻击技术的对抗。

当云计算发展将近二十年后，已然进入了新的阶段：云原生时代。以容器、编排和微服务为代表的云原生技术，正在影响各行各业的 IT 基础设施、平台和应用系统，也在渗透到如 IT/OT 融合的工业互联网、IT/CT 融合的 5G、边缘计算等新型基础设施中。理解云原生体系存在的新架构、新风险和新威胁，有助于我们构建面向新型 IT 基础设施的安全防护机制；利用云原生的先进技术，融合到当前的攻防体系中，也有助于我们提升整体安全防护的弹性、适应性、敏捷性。

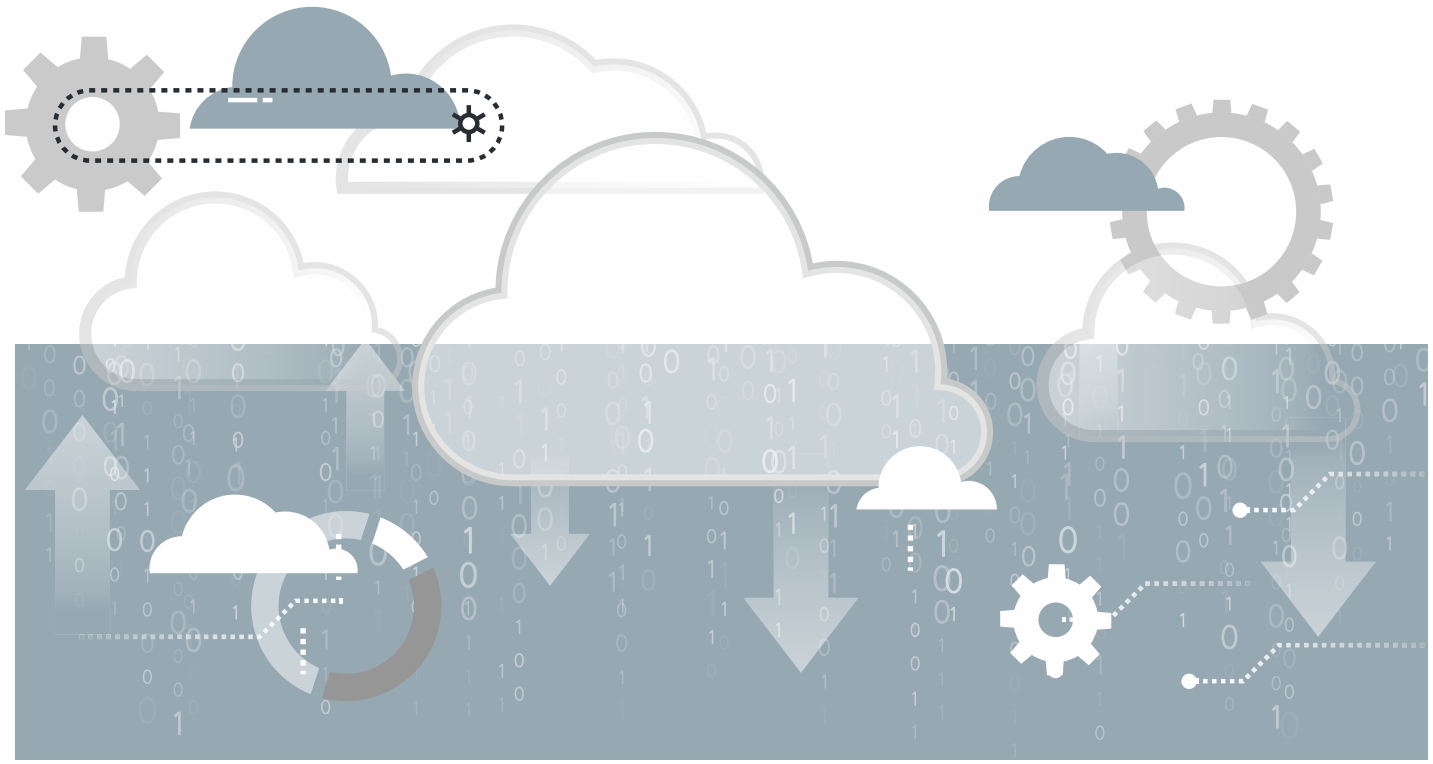
我们在 2018 年发布了《容器安全技术报告》，详细分析了容器技术所面临的风险和安全挑战，介绍了安全左移思路下的安全基线、镜像安全等内容，给初建容器安全的机构一些有益的建议。近年随着编排技术、无服务和服务网格等技术的快速发展，我们将重点放在了整个云原生生态体系的安全研究上。

鉴于此，我们发布了《云原生安全技术报告》，本报告较为全面地分析了云原生落地所面临的安全风险和威胁，进而提出了云原生的防护思路和安全体系。首先，详细介绍了多种实现云原生环境的可观察、可追踪的技术手段；然后介绍了在事前应实现零信任网络访问控制的微隔离技术，以及在事中面向容器环境的异常检测技术和面向云原生应用和服务的应用业务安全技术。此外，报告也介绍了云原生新形态的服务网格和无服务安全防护技术。最后，报告就 5G 核心网和边缘计算等场景探讨了云原生安全的应用。

本报告在编写过程中参考了大量资料，吸取了多方的宝贵意见和建议，在此深表感谢。报告的编写和发布得到了相关单位的大力支持，我们在此表示衷心的感谢！欢迎广大读者批评、指正。

1

云原生安全概述



随着云计算技术的成熟和云计算系统的广泛部署，上云成为了企业部署新业务的首选。同时，各大云计算服务商的厮杀日益激烈，新的概念也层出不穷。近年来，云原生计算（Cloud Native Computing）越来越多地出现在人们的视野中，那么云原生计算与传统云计算相比有什么不同，能利用云原生计算解决什么问题，本章我们将介绍云原生计算的概念和应用场景。

1.1. 云原生简介

近年来，云计算的模式逐渐被业界认可和接受。在国内，大多数利用开源或商业的 IaaS 系统构建云计算平台，这种方式所带来的好处是整体资源利用更加合理，集约式的运营降低成本，提升整体水平。但总体而言，这样的上云实践只是“形”上的改变，还远没有到“神”上的变化。

在上云实践中，传统应用有升级缓慢、架构臃肿、无法快速迭代等问题，于是云原生的概念应运而生。云原生充分利用云计算弹性、敏捷、资源池和服务化等特性，解决业务在开发、集成、分发和运行等整个生命周期中遇到的问题。真实业务中出现的问题，才是真正的“问题”。所以，我们认为云计算下半场的开场哨已经吹响，谁赢得云原生的赛道，谁才真正赢得了云计算。

更为正式地，云原生计算基金会（Cloud Native Computing Foundation，CNCF）对云原生的见解[1]是“云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式 API。这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。”

在云原生应用和服务平台构建过程中，容器技术凭借其轻隔离、易分发等特性和活跃强大的社区支持，成为了云原生等应用场景下的重要支撑技术。无服务、服务网格等服务新型部署形态也在改变云端应用的设计、开发、部署和运行，从而重构云上业务模式。

1.2. 云原生安全简介

与云计算安全相似，云原生安全也包含两层含义：“面向云原生环境的安全”和“具有云原生特征的安全”。

[1] https://www.cncf.io/about/faq

1.2.1. 面向云原生环境的安全

面向云原生环境的安全，其目标是防护云原生环境中的基础设施、编排系统和微服务的安全。

这类安全机制，不一定具备云原生的特性（比如容器化、可编排），它们可以是传统模式部署的，甚至是硬件设备，但其作用是保护日益普及的云原生环境。

一个例子是容器云（CaaS）的抗拒绝服务，可用分布式拒绝服务缓解机制（DDoS Mitigation），而考虑到性能限制，一般此类缓解机制都是硬件形态，但这种传统安全机制正是保障了是面向云原生系统的可用性。

此外，主机安全配置、仓库和镜像安全、行为检测和边界安全等都是面向云原生环境的安全机制。

1.2.2. 具有云原生特性的安全

具有云原生特征的安全，是指具有云原生的弹性敏捷、轻量级、可编排等特性的各类安全机制。

云原生是一种理念上的创新，通过容器化、资源编排和微服务重构了传统的开发运营体系，加速业务上线和变更的速度，因而，云原生系统的种种优良特性同样会给安全厂商带来很大的启发，重构他们的安全产品、平台，改变其交付、更新模式。

仍以 DDoS 为例，在数据中心的安全体系中，抗拒绝服务是一个典型的安全应用，以硬件清洗设备为主。但其缺点是当 DDoS 的攻击流量超过了清洗设备的清洗能力时，无法快速部署额外的硬件清洗设备（传统硬件安全设备的下单、生产、运输、交付和上线往往以周计），因而这种安全机制无法应对突发的大规模拒绝服务攻击。而如果采用云原生的机制，安全厂商就可以通过容器镜像的方式交付容器化的虚拟清洗设备，而当出现突发恶意流量时，可通过编排系统在空闲的服务器中动态横向扩展启动足够多的清洗设备，从而可应对处理能力不够的场景。这时，DDoS 清洗机制是云原生的，但其防护的业务系统有可能是传统的。

1.2.3. 云原生安全：融合的安全

虽然我们形式上将云原生安全分为了两类安全技术路线，但事实上，如果要做好云原生安全，则必然需要使用“具有云原生特性的安全”技术去实现“面向云原生环境的安全”，因而两者是互相融合的。

让我们继续以 DDoS 威胁为例，一方面可用性是整个云原生系统中重要的安全属性，无论是宿主机、容器，还是微服务、无服务业务系统，都需要保证其可用性；另一方面，在云原生系统中要实现可用性，

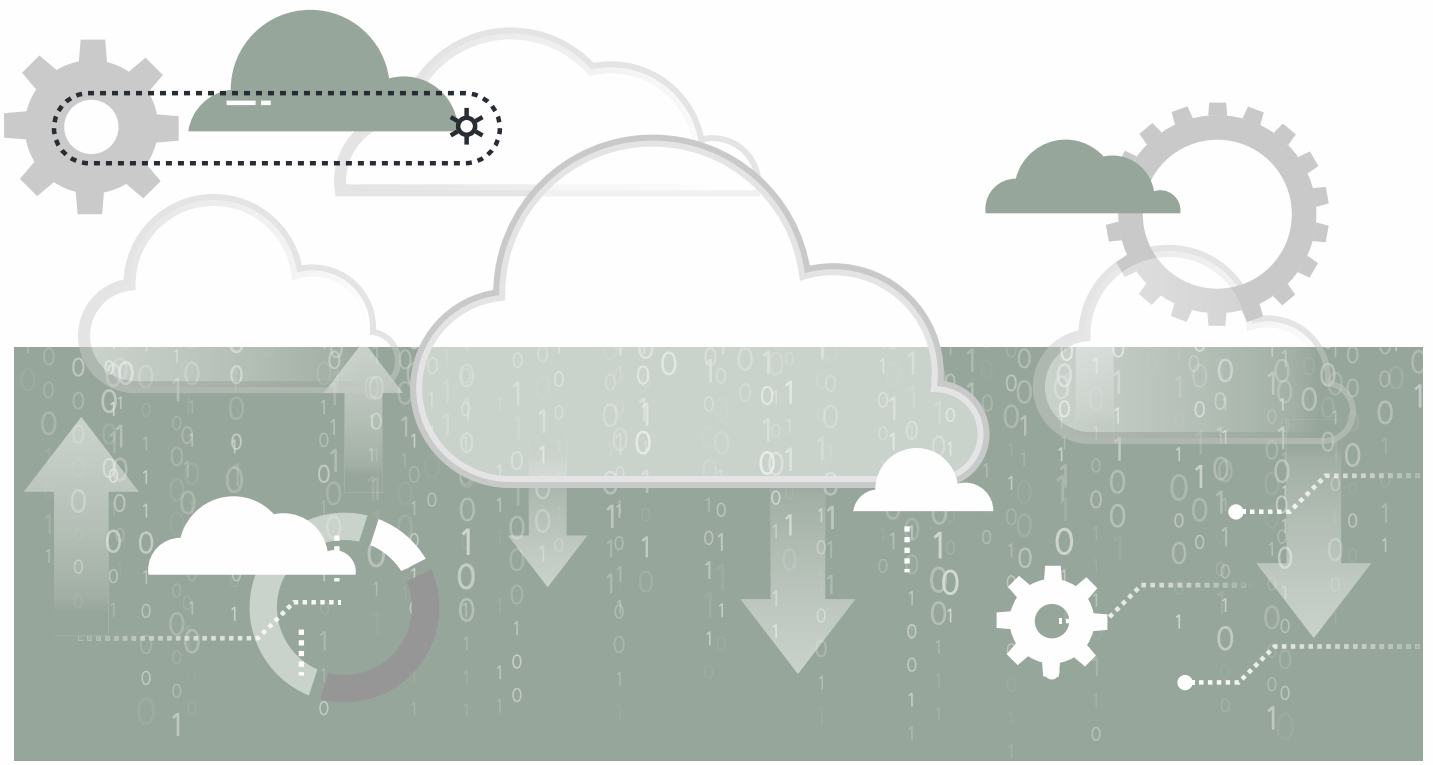
在物理边界侧要构建按需、弹性的 DDoS 缓解能力，在容器、微服务边界侧部署虚拟、弹性的 DDoS 机制，这就需要云原生的安全能力。两者相辅相成，缺一不可。

1.3. 小结

要实现云原生安全，就需要首先评估云原生系统中的安全风险和相关威胁，设计具有针对性的安全机制；在此过程中，云原生系统的固有业务特点，就要求安全厂商现有的安全能力云原生化。目前可预见的是，安全能力的云原生化，绝不是简单的将原有安全设备虚拟化或容器化，而是要按照部署要求进行容器化，按照性能要求进行轻量化，按照功能进行拆分、微服务化，最后实现弹性、敏捷、轻量级等特性。

2

安全问题成为影响云原生落地的重要因素



从各种各样的安全风险中可以一窥云原生技术的安全态势，云原生环境仍然存在许多安全问题亟待解决。在云原生技术的落地过程中，安全是必须要考虑的重要因素。

2.1. 服务暴露

服务暴露的问题，我们在 2018 年的报告 [2] 中就已经详细进行了分析，这里再简单做一下回顾。我们利用网络空间搜索引擎 Shodan 对 2020 年 10 月份某天暴露 2375 端口的 Docker Daemon 服务进行了分析，发现仍然有约 4800 个 Docker Daemon 对外暴露 2375 端口，其中部分是没有设置访问认证的，可以直接与之交互。

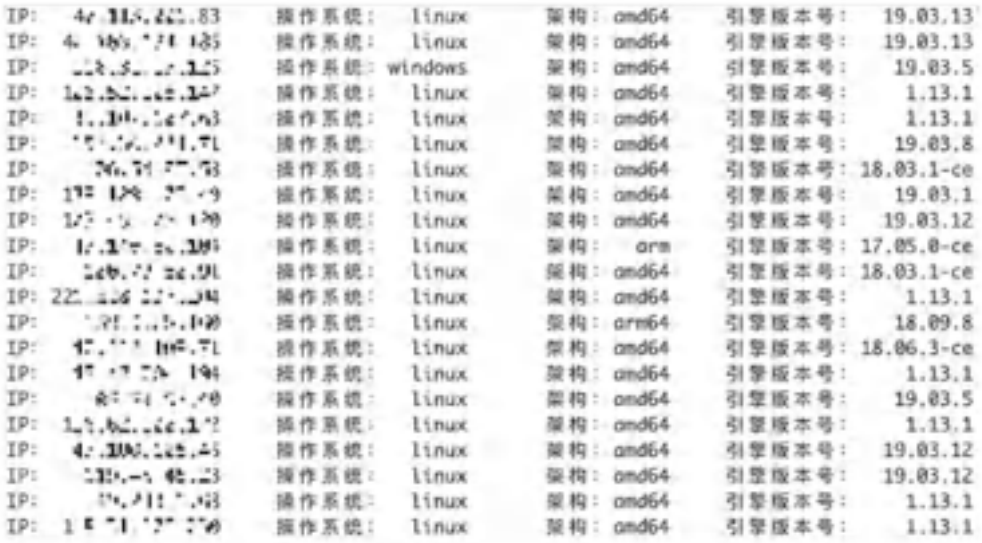


图 2.1　部分暴露 2375 端口且无访问认证的主机

另外，我们也对同时期 Kubernetes Dashboard 在互联网上的暴露情况进行了统计，发现仍有约 1200 个 Kubernetes Dashboard 服务暴露在互联网上。其中也仍存在不需要认证即可登录的情况。这种端口暴露是非常危险的。

[2] https://www.nsfocus.com.cn/html/2018/92_1112/70.html

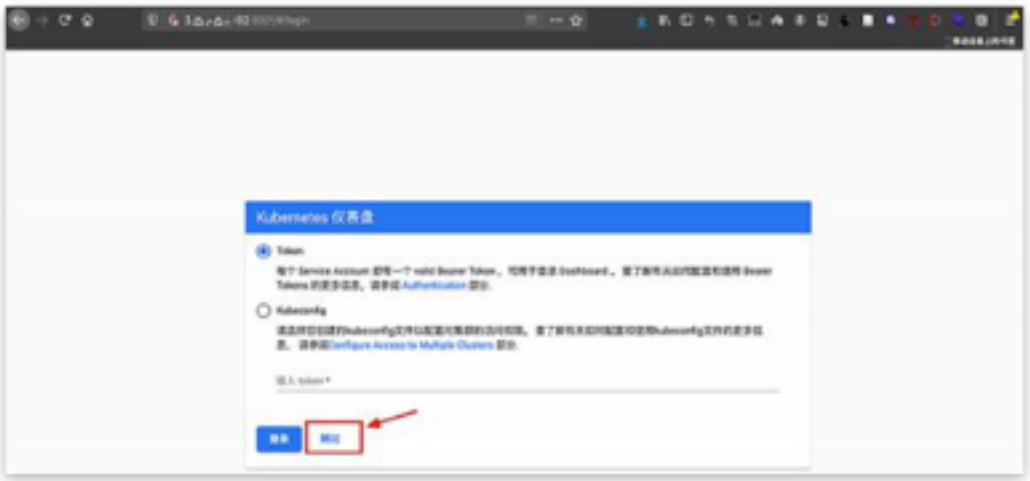


图 2.2　一个对外暴露端口且无访问认证的 Kubernetes Dashboard 服务

2.2.　数据泄露

数据泄露在云原生中仍然是重要的安全威胁。以亚马逊的对象存储服务 Amazon S3 为例，其在云原生中也得到了应用 [3]，然而，S3 数据泄露事件被屡屡曝光，不胜枚举 [4][5]。

在上述众多数据泄露事件中，绝大多数是 S3 存储桶的公开访问，有的事件涉及 S3 被设置为允许任何 AWS 登录用户访问，甚至一个医疗数据泄露事件的相关存储桶竟然被设置为了任何人均可读写，这是不可想象的。

2.3.　恶意挖矿攻击

特斯拉 Kubernetes 挖矿事件

2018 年 2 月 20 日，云安全公司 RedLock 发文披露 [6]，特斯拉的 Kubernetes 集群曾在数月前被入侵，黑客在其中部署了挖矿程序。据报道，入侵的直接原因是，其 Kubernetes 集群的 Dashboard 处于未授

[3] *https://aws.amazon.com/cn/products/storage/object-storage-for-cloud-native-applications*
[4] *https://businessinsights.bitdefender.com/worst-amazon-breaches*
[5] *https://github.com/nagwww/s3-leaks*
[6] *https://web.archive.org/web/20180222103919/https://blog.redlock.io/cryptojacking-tesla*

权即可访问状态，且暴露在互联网上。

对于一个 Kubernetes 集群来说，控制 Kubernetes Dashboard 意味着能够直接向集群下达指令，严重情况下攻击者甚至能够逃逸出容器，进而轻易控制集群所有宿主机节点，其风险等同于甚至超过域沦陷，却往往得不到相应的重视。

微软监测到大规模 Kubernetes 挖矿事件

2020 年 4 月 8 日，微软 Azure 安全中心发布博文称，检测到大规模 Kubernetes 挖矿事件 [7]。2020 年 6 月 10 日，微软 Azure 安全中心再次发布文章，警告滥用 Kubeflow 的大规模挖矿事件 [8]。两起事件的曝光日期如此接近，说明以 Kubernetes 为代表的云原生系统很可能正在成为不法分子赖以牟利的新渠道。

针对 Kubeflow 的攻击事件，影响了数十个 Kubernetes 集群，通常用于机器学习的 Kubernetes 集群各节点的算力会比普通服务器要强大许多，这些算力同样能够被攻击者所用，为挖矿带来显著增益。

事实上，容器环境下的恶意挖矿并不新鲜，挖矿活动一般发生在脆弱容器中。例如，黑客利用容器内部运行有漏洞 Web 的应用程序，获得容器控制权，进而部署挖矿程序。

Graboid 蠕虫挖矿传播事件

2019 年 10 月 15 日，Unit 42 安全团队发现了一款新型挖矿蠕虫 [9]，命名为 Graboid（大地虫），该蠕虫的挖矿产出是门罗币（Monero），被发现时，它已经传播到了超过 2000 台不安全的 Docker 宿主机上。这是安全研究者第一次发现借助 Docker 容器进行挖矿和传播的蠕虫。

本次事件中，攻击者首先通过不安全的 Docker 守护进程暴露出来的远程端口获得对目标主机的控制权，然后下达指令从 Docker Hub 上拉取（pull）并运行实现上传的恶意镜像。由于传统终端防护机制（AV、EDR）并不能很好地审计容器内部的行为和数据，故很难检测出这类恶意行为。

我们必须认真对待云原生环境下的蠕虫传播事件。相比传统内网环境，云原生集群环境下各节点之间具有更强的关联性。例如，在 Kubernetes 集群内，一旦蠕虫获得了 API Server 的控制权限，就能够

[7] https://azure.microsoft.com/en-us/blog/detect-largescale-cryptocurrency-mining-attack-against-kubernetes-clusters

[8] https://www.microsoft.com/security/blog/2020/06/10/misconfigured-kubeflow-workloads-are-a-security-risk

[9] https://unit42.paloaltonetworks.com/graboid-first-ever-cryptojacking-worm-found-in-images-on-docker-hub

在非常短的时间内感染整个集群，并能利用编排技术持久化恶意代码，后果不堪设想。

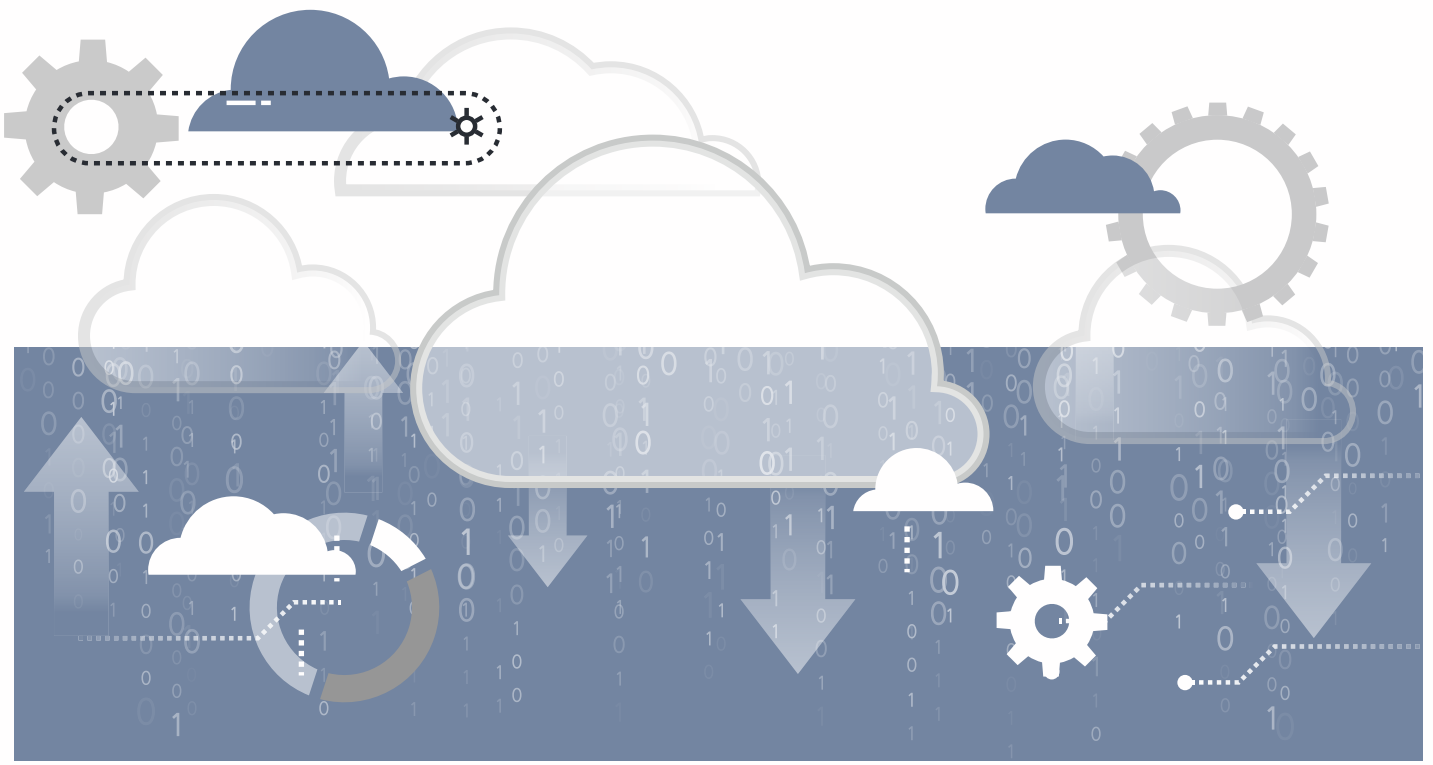
2.4. 小结

谈及云原生安全时，不少人还停留在传统安全意识和观念，对 Web 攻防、系统攻防加强防范，密码暴力破解和反弹 Shell 不会掉以轻心。然而，安全总是具有“短板效应”。有时，一个简单的端口暴露、未授权访问漏洞没有及时处理，就可能为攻击者提供不费吹灰之力长驱直入的机会，固若金汤的城池也会因此沦陷。

此外，云原生自身脆弱性带来的风险，在未来数年内，很可能被攻击者所利用，进而发动针对云原生系统的攻击，因而云原生面临的威胁和安全风险不可不察。

3

云原生面临的安全威胁和风险



3.1. 容器化基础设施的威胁和风险不会更少

以容器和容器编排系统为代表的云原生基础设施，成为支撑云原生应用的重要载体，其安全性将直接影响整个云原生系统的安全性。

3.1.1. 针对容器镜像的软件供应链攻击

随着容器技术的普及，容器镜像也成为了软件供应链中非常重要的一部分。然而，业务依赖的基础镜像无论是存在安全漏洞还是包含恶意代码，这种潜在危害比黑客从外部发起攻击严重得多。下面，我们将介绍两种容器镜像软件供应链的攻击类型。

镜像漏洞利用

“镜像漏洞利用”指的是镜像本身存在漏洞时，依据镜像创建并运行的容器也通常会存在相同漏洞，攻击者利用镜像中存在的漏洞去攻击容器，往往具有事半功倍的效果。

备受开发者青睐的 Alpine 镜像曾曝出过一个漏洞，编号为 CVE-2019-5021。3.3 到 3.9 版本的 Alpine 镜像中，root 用户密码被设置为空，攻击者可能在攻入容器后借此提升到容器内部 root 权限。这个漏洞看起来很简单，但是 CVSS 3.0 评分高达 9.8 分 [10]。

镜像投毒

镜像投毒是一个宽泛的话题。它指的是攻击者通过某些方式——如上传镜像到公开仓库、入侵系统后上传镜像到受害者本地仓库以及修改镜像名称假冒正常镜像等，欺骗、诱导受害者使用攻击者指定的恶意镜像创建并运行容器，从而实现入侵或利用受害者的主机进行恶意活动的行为。

根据目的不同，常见的镜像投毒有三种类型：投放恶意挖矿镜像、投放恶意后门镜像和投放恶意 exploit 镜像。

投递恶意挖矿镜像

这种投毒行为主要是为了欺骗受害者在机器上部署容器，从而获得经济收益。2018 年 6 月，一份

[10] https://nvd.nist.gov/vuln/detail/CVE-2019-5021

研究报告指出[11]，一个名为 docker123321 的账号向 Docker Hub 上陆续上传了 17 个包含挖矿代码的恶意镜像。截至 Docker Hub 官方移除这些镜像时，它们已经累计被下载超过 500 万次。据统计，黑客借助这一行为获得了时值约 9 万美元的门罗币。

投放恶意后门镜像

这种投毒行为主要是为了实现对容器的控制。通常，受害者在机器上部署容器后，攻击者会收到容器反弹过来的 Shell。

相比之下，这种类型的投毒可能会少一些——因为在隔离有效的情况下，即使攻击者拿到一个容器内部的 Shell，攻击面仍然有限。在 2017 年 9 月，有用户在 Docker Hub 的反馈页面中反馈前述 docker123321 账号上传的 Tomcat 镜像包含了后门程序[12]。结合该账号上传的其他恶意镜像的挖矿行为来看，有理由推测攻击者在连接上述后门后可能会部署挖矿程序，获得经济收益。

投放恶意 exploit 镜像

这种投毒行为是为了在受害者部署容器后尝试寻找宿主机上的各种漏洞来实现容器逃逸等目的，以实现对受害者机器更强的控制。

从攻防对抗的角度来看，恶意 exploit 镜像只不过是一种攻击载荷投递方式，其特点在于隐蔽性和可能的巨大影响范围。试想，如果 Docker Hub 上某一热门镜像包含了某个 1day 甚至 0day 漏洞的利用程序，理论上，攻击者将可能一下子获取上百万台计算机的控制权限。

3.1.2. 容器逃逸

与其他虚拟化技术类似，逃逸是最为严重的安全风险，直接危害了底层宿主机和整个云计算系统的安全。

根据层次的不同，容器逃逸的类型可以分为以下三类：

[11] https://mackeeper.com/blog/post/cryptojacking-invades-cloud-how-modern-containerization-trend-is-exploited-by-attackers/

[12] https://github.com/docker/hub-feedback/issues/1121#issuecomment-326664651

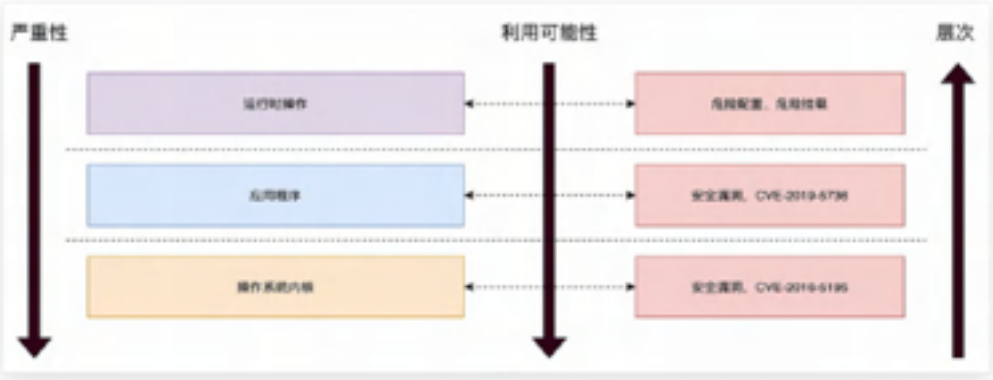


图 3.1　不同层次的容器逃逸问题

上图可以进一步展开为：危险配置导致的容器逃逸；危险挂载导致的容器逃逸；相关程序漏洞导致的容器逃逸；内核漏洞导致的容器逃逸。

危险配置导致的容器逃逸

在这些年的迭代中，容器社区一直在努力将纵深防御、最小权限等理念和原则落地。然而，无论是细粒度权限控制还是其他安全机制，用户都可以通过修改容器环境配置或在运行容器时指定参数来缩小或扩大约束。如果用户为不完全受控的容器提供了某些危险的配置参数，就为攻击者提供了一定程度的逃逸可能性。

最初，容器特权模式的出现是为了帮助开发者实现 Docker-in-Docker 特性 [13]。然而，在特权模式下运行不完全受控容器将给宿主机带来极大安全威胁。例如，攻击者可以直接在容器内部挂载宿主机磁盘，然后将根目录切换过去，从而实现容器逃逸。

危险挂载导致的容器逃逸

在某些特定场景下，为了实现特定功能或方便操作（例如为了在容器内对容器进行管理将 Docker Socket 挂载到容器内），用户会将外部敏感卷挂载入容器。随着容器技术应用的逐渐深化，挂载操作变得愈加广泛，由此而来的安全问题也呈现上升趋势。

[13]　https://www.docker.com/blog/docker-can-now-run-within-docker

Docker Socket 是 Docker 守护进程监听的 Unix 域套接字，用来与守护进程通信，查询信息或下发命令。

如果在攻击者可控的容器内挂载了该套接字文件（/var/run/docker.sock），容器逃逸就相当容易了，除非有进一步的权限限制。一种逃逸方法是：（1）在容器内安装 Docker 命令行客户端；（2）使用该客户端通过 Docker Socket 与 Docker 守护进程通信，发送命令创建并运行一个新的容器，将宿主机的根目录挂载到新创建的容器内部；（3）在新容器内执行 chroot 将根目录切换到挂载的宿主机根目录。

相关程序漏洞导致的容器逃逸

所谓相关程序漏洞，指的是那些参与到容器生态中的服务端、客户端程序自身存在的漏洞。

CVE-2019-5736 正是这样一个存在于 runC 的容器逃逸漏洞，它由波兰 CTF 战队 Dragon Sector 在 35C3 CTF 赛后基于赛中一道沙盒逃逸题目获得的启发，对 runC 进行漏洞挖掘，成功发现的一个能够覆盖宿主机 runC 程序的容器逃逸漏洞。该漏洞于 2019 年 2 月 11 日通过邮件列表披露，具体分析可参见“容器逃逸成真：从 CTF 解题到 CVE-2019-5736 漏洞挖掘分析”[14]。

内核漏洞导致的容器逃逸

Linux 内核漏洞的危害之大、影响范围之广，使得它在各种攻防话题下都占据非常重要的一席。

近年来，Linux 系统曝出过无数内核漏洞，其中能够用来提权的也不少，脏牛（CVE-2016-5195）大概是其中最有名气的漏洞之一。事实上，脏牛漏洞也能用来进行容器逃逸，一种利用方法的核心思路是向 vDSO 内写入 shellcode 并劫持正常函数的调用过程。在成功触发漏洞后，攻击者可以获得宿主机上反弹过来的 Shell，实现容器逃逸。

其他

作为一种轻量级虚拟化技术，传统容器与宿主机共享内核，这意味着系统内核权限提升漏洞往往也可用来实施容器逃逸。为了彻底解决这一问题，在轻量与安全性之间达到较好的平衡，安全容器应运而生。Kata Containers 是一种安全容器的具体实现，其他主流的安全容器还有 Google 推出的 gVisor 等。

无论是在理论上，还是在实践中，安全容器都具有非常高的安全性。然而，在 2020 年 Black Hat

[14] https://mp.weixin.qq.com/s/UZ7VdGSUGSvoo-6GVl53qg

北美会议上，Yuval Avrahami 分享了利用多个漏洞成功从 Kata containers 逃逸的议题 [15]，安全容器首次被发现存在逃逸可能性。他使用发现的三个漏洞（CVE-2020-2023、CVE-2020-2025 和 CVE-2020-2026）组成漏洞利用链先后进行容器逃逸和虚拟机逃逸，成功从容器内部逃逸到宿主机上。

3.1.3. 资源耗尽型攻击

同为虚拟化技术，容器与虚拟机既存在相似之处，也有显著不同。在资源限制方面，我们需要为即将创建的虚拟机设定明确的 CPU、内存及硬盘资源阈值，在虚拟机内部的进程看来，它真的处于一台被设定好的独立计算机之中。

然而，容器运行时默认情况下并未对容器内进程在资源使用上做任何限制，以 Pod 为基本单位的容器编排管理系统也是类似的，Kubernetes 在默认情况下同样未对用户创建的 Pod 做任何 CPU、内存使用限制 [16]。

限制的缺失使得云原生环境面临资源耗尽型攻击风险。攻击者可能通过在一个容器内运行恶意程序，或针对一个容器服务发起拒绝服务攻击来占用大量宿主机资源，从而影响到宿主机或宿主机上其他容器的正常运行。

3.2. 微服务架构为云原生应用安全带来新的安全风险

在如今应用的开发、测试、运维均追求敏捷开发的时代，微服务应运而生。但随着应用的微服务化升级，新的安全风险也不容忽视。

首先，随着微服务的增多，暴露的端口数量也急剧增长，进而扩大了攻击面，且微服务间的网络流量多为东西向流量，网络安全防护维度发生了改变。

其次，不同于单体应用只需解决用户或外部服务与应用的认证授权，微服务间的相互认证授权机制更为复杂，人为因素导致认证授权配置错误成为了一大未知风险。

最后，微服务通信依赖于 API，随着业务规模的增大，微服务 API 数量激增，恶意的 API 操作可能会引发数据泄漏、越权访问、中间人攻击、注入攻击、拒绝服务等风险。

[15] https://i.blackhat.com/USA-20/Thursday/us-20-Avrahami-Escaping-Virtualized-Containers.pdf

[16] https://kubernetes.io/docs/concepts/policy/limit-range/

3.2.1. 微服务数据泄露

应用存储的数据基于 API 进行访问，若应用中某个 API 含有漏洞或通信未采用加密协议，攻击者便可利用漏洞进行越权或中间人攻击，从而达到数据窃取的目的。

容器镜像为承载云原生应用的实体，开发者将敏感信息写入 Dockerfile、源码或以环境变量的方式进行传递，这种行为增加了数据泄露的风险，随着微服务数量的不断增多，其造成的损失也呈指数增长。

Kuberenetes 作为主流的微服务管理平台内置了许多安全机制，但默认值通常并不安全，例如我们创建了 Secrets 资源用于存储数据库登录信息，虽然这是一种正确存储敏感信息的方式，但默认情况下，Kubernetes 在 Etcd 组件中存储的信息是明文的，如果未对 Etcd 组件开启 TLS 访问，攻击者一旦进入 Kubernetes 集群内部便可对 Etcd 组件进行访问从而造成大量微服务密钥信息泄露。

服务网格场景中，当微服务数量达到一定规模，API 数量将不断递增，从而增大了数据泄露的风险。另外，微服务间有着复杂的网络拓扑，如果服务间的通信流量未加密，攻击者一旦进入服务内部网络，便可通过截获请求中的敏感传输数据（例如用户名、密码、密钥信息等）进行中间人攻击。

3.2.2. 微服务间请求伪造

应用架构的改变使微服务授权面临新的风险。在单体应用架构下，应用作为一个整体对用户进行授权。在微服务场景下，所有的服务均需对各自的访问进行授权，明确当前用户的访问控制权限。传统的单体应用访问来源相对单一，基本为浏览器，而微服务的访问来源还包括内部的其它微服务，因此，微服务授权除了服务对用户的授权，还增加了服务对服务的授权。

Kubernetes 中，内部容器间访问控制和隔离可使用网络策略（NetworkPolicy）实现，但其默认情况下 Pod 之间是互通的，这也就意味着网络策略是以白名单的模式出现，因此如果不对 Pod 之间的访问进行显示授权，一旦集群内部的某一 Pod 失陷，便会极速扩展至整个集群的 Pod 失陷。

3.2.3. 微服务 Web 应用的风险

OWASP 发布了 Web 应用安全 10 大风险，其中对常见的 Web 攻击风险进行了详细的总结 [17]。Web 攻击作为应用层的主要攻击类型，在微服务架构中也依然存在，其风险同时不容小觑。

[17] https://owasp.org/www-project-top-ten

3.2.4. 微服务业务的风险

与网络层的安全不同，业务层面的安全往往没有明显的网络攻击特征。攻击者可以利用业务系统的漏洞或者规则对业务系统进行攻击来获利，给业务系统造成损失。

在微服务架构下，各服务如果安全措施不完善，例如用户鉴权、请求来源校验不到位，将会导致针对微服务业务层面的攻击变得更加容易。例如针对一个电商应用，攻击者可以对特定的服务进行攻击，例如通过 API 传入非法数据，或者直接修改微服务的数据库系统等。攻击者可以绕过验证码服务，直接调用订单管理服务来进行薅羊毛操作。攻击者甚至可以通过直接修改订单管理和支付所对应的微服务系统，绕过支付的步骤，直接成功购买商品等。

3.3. 管理编排系统面临的安全风险不容小觑

在云原生环境中，编排系统无疑处于重中之重的地位，而 Kubernetes 早已成为容器编排系统的“事实标准”。本节，我们来从控制权限陷落、拒绝服务攻击和网络安全风险三个方面去介绍管理编排系统的潜在风险。

3.3.1. Kubernetes 控制权限陷落

Kubernetes 组件的不安全配置。由于 Kubernetes 组件众多、各组件配置复杂，不安全配置引起的风险不容小觑。比如 Kubernetes API Server 的未授权访问，Kubernetes Dashboard 的未授权访问，Kubelet 的未授权访问等。

以 API Server 的未授权访问为例，默认情况下，API Server 能够在两个端口上对外提供服务：8080 和 6443，前者以 HTTP 协议提供服务，无认证和授权机制；后者以 HTTPS 协议提供服务，支持认证和授权服务。

在较新版本的 Kubernetes 中，8080 端口的 HTTP 服务默认不启动。然而，如果用户在启用了 8080 端口并重启 API Server，那么只要网络可达，攻击者即可通过此端口操控集群。

Kubernetes 权限提升。以 CVE-2018-1002105 漏洞为例，这是一个 Kubernetes 的权限提升漏洞，允许攻击者在拥有集群内低权限的情况下提升权限至 Kubernetes API Server 权限，CVSS 3.x 评分为

9.8[18]。通过构造一个对 Kubernetes API Server 的特殊请求，攻击者能够借助 Kubernetes API Server 作为代理，建立一个到后端服务器的连接。利用该连接，攻击者能够以 Kubernetes API Server 的身份向后端服务器发送任意请求，实现权限提升。

突破隔离，访问宿主机文件系统。以 CVE-2017-1002101 漏洞为例，这是一个 Kubernetes 的文件系统逃逸漏洞，允许攻击者使用 subPath 卷挂载来访问卷空间外的文件或目录，CVSS 3.x 评分为 9.8[19]。所有 v1.3.x、v1.4.x、v1.5.x、v1.6.x 及低于 v1.7.14、v1.8.9 和 v1.9.4 版本的 Kubernetes 均受到影响。

这个漏洞本质上是 Linux 符号链接特性与 Kubernetes 自身代码逻辑两部分结合的产物。符号链接引起的问题并不新鲜，这里它与虚拟化隔离技术碰撞出了逃逸问题，以前还曾有过在传统主机安全领域与 SUID 概念碰撞出的权限提升问题等 [20]。

3.3.2. 针对 Kubernetes 的拒绝服务攻击

拒绝服务攻击有多种类型。常见的是基于流量的拒绝服务攻击和基于漏洞的拒绝服务攻击。

传统环境和云原生环境的流量攻击的差异性较小，攻击效果通常取决于流量大小。基于漏洞的拒绝服务攻击则不然，存在于云原生组件的拒绝服务漏洞很可能并不存在于传统主机环境，典型的漏洞包括 CVE-2019-11253、CVE-2019-9512、CVE-2019-9514 等。

3.3.3. 云原生网络安全风险

默认情况下，Kubernetes 集群中所有 pod 组成了一个小型的局域网络，那么就可能发生像中间人攻击这样的针对局域网的攻击行为。

假如攻击者借助 Web 渗透等方式攻破了某个 Pod，就有可能针对集群内的其他 Pod 发起中间人攻击，进而进行 DNS 劫持等。例如，攻击者攻破一个前端服务（Web APP Pod）之后，通过 ARP 欺骗诱导另一个后端服务（Backend Pod）以为 Web APP Pod 是集群的 DNS 服务器，进而使得 Backend Pod 在对外发起针对某域名（example.com）的 HTTP 请求时首先向 Web APP Pod 发起 DNS 查询请求。Backend Pod 以为自己拿到了正确的信息，其实不然。攻击者能够潜伏在集群中，不断对其他 Pod 的

[18] https://nvd.nist.gov/vuln/detail/cve-2018-1002105
[19] https://nvd.nist.gov/vuln/detail/cve-2017-1002101
[20] https://en.wikipedia.org/wiki/Symlink_race

网络流量进行窃听，甚至可以悄无声息地劫持、篡改集群其他 Pod 的网络通信，危害极大。

3.4. 服务网格安全风险

服务网格是一个微服务的基础设施层，主要用于处理服务间的通信。通常云原生应用会有大量微服务，微服务组成了复杂的服务拓扑，服务网格负责在这些拓扑中实现请求的可靠传递。在实践中，服务网格通常实现为一组轻量级网络代理，它们与应用程序部署在一起，而对应用程序透明。Istio 是一款典型的微服务管理和服务网格框架项目，其架构图 [21] 如下所示：

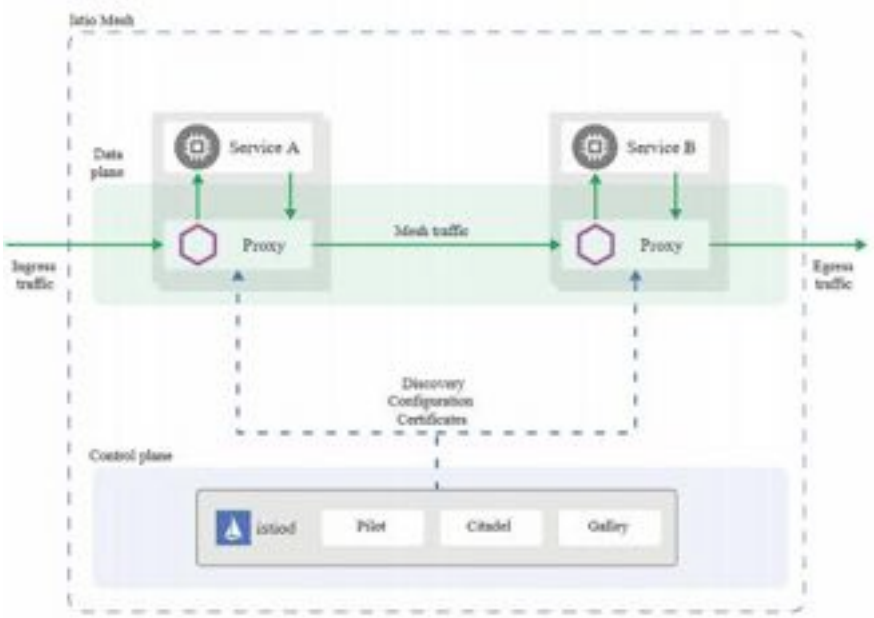


图 3.2 Istio 架构图

Istio 架构的设计类似软件定义网络，主要分为控制平面和数据平面两个部分，其中数据平面由一组代理组成并以 Sidecar 的形式部署在服务旁，这些代理负责管理服务间的所有网络通信；控制平面用于对数据平面的代理进行管理和配置，具有应用无感知、可扩展、统一管理等优点。然而，引入新的架构设计同时会引入新的风险，服务网格带来的安全风险不容小觑。

[21] https://istio.io/latest/docs/ops/deployment/architecture/

3.4.1. 服务间易受中间人攻击

在当前的服务网格中，服务与服务间通过 Sidecar 进行通信，Istio 的 Sidecar 使用 Envoy 作为边车代理，默认情况下，由于 Envoy 未对服务间的通信采用双向 TLS 认证，所以攻击者一旦入侵某服务，便可在服务网格中自由进行横向移动，进而发起攻击。一方面，攻击者可在服务通信时进行中间人攻击，通过截取、修改、伪造通信内容造成数据泄漏；另一方面，攻击者也可以伪造服务身份进行通信，从而破坏整个微服务网络。

3.4.2. 服务间易受越权攻击

服务网格认证授权面临安全风险主要分为以下两种类型：

东西向的认证授权：东西向流量是微服务间的流量，即服务网格内部流量。由于请求在到达业务容器前首先经过 Sidecar 容器，因此为了向服务间下发访问控制策略，Sidecar 容器应具备有流量管理、元数据收集、安全控制等功能，Istio 主要采用 Envoy 内部的 filter 机制实现上述功能。默认情况下，Envoy 容器通过修改 Pod 底层 iptbales 实现流量的重定向，这一操作只是起到了流量传递的作用，并未对流量进行任何限制，这无疑是非常危险的。如果未对服务网格的服务间下发有效认证授权策略，一旦服务网格中的服务 A 有权限对服务 B 进行访问，便也可对网格中的其它服务进行访问，那么在微服务内网被攻破的前提下，攻击者可以进行任意越权操作，从而造成严重损失。

南北向的认证授权：外界服务访问微服务的流量为南北向流量。在服务网格设计中，通常南北向流量会经过服务网格边界处的 API 网关，后者会对外部的 API 请求做负载均衡、请求频次控制和安全控制等，若 API 网关提供的防护能力不强，则攻击者可利用应用程序漏洞恶意访问服务，并利用服务间未授权访问漏洞进行进一步的提权操作，从而最终控制整个微服务网络，造成了严重的损失。

3.5 Serverless 面临的威胁

2016 年 8 月，martinfowler.com 网站上发表的《Serverless》[22] 一文中对 Serverless 概念做了阐述：*Serverless 可在不考虑服务器的情况下构建并运行应用程序和服务，它使开发者避免了基础设施管理，例如集群配置、漏洞修补、系统维护等。Serverless 并非字面理解的不需要服务器，只是服务器均交由第三方管理。*

[22] https://martinfowler.com/bliki/Serverless.html

FaaS（Function as a Service）是 Serverless 主要的实现方式，开发者通过编写一段代码，并定义何时以及如何调用该函数，随后该函数在云厂商提供的服务端运行，全程开发者只需编写并维护一段功能代码即可，比如广泛使用的 AWS Lambda 服务。

Serverless 是新的云计算模式，在给开发者带来便利的同时，其安全风险也备受关注。由于 Serverless 架构包含开发者侧及服务提供商侧，因此其安全架构也有相应的责任划分。

下文主要从开发者面临的安全威胁出发，大致分为五类。

3.5.1. 针对应用程序代码的注入攻击

如果应用程序未对外界输入数据进行过滤或编码校验，那么它们就可能存在 SQL 注入、系统命令执行等攻击成功的风险。在传统应用程序开发中，开发者根据自身实践经验，在数量有限的可能性中可判定出恶意输入来源，但 Serverless 模式下函数调用是由事件源触发，如下图所示，输入来源的不确定性限制了开发者的判定。

图 3.3 函数事件源触发示意图

通常来说，当函数订阅一个事件源后，该函数在该类型的事件发生时被触发，这些事件可能来源于 FaaS 平台，也可能来源于未知的事件源，对于来源未知的事件源可以被标注为不受信任。在实际应用场景中，如果开发者没有良好的习惯对事件源进行分类，则会经常导致将不受信任的事件错认为是 FaaS 平台事件，进而将其视为受信任的输入来处理，最终受到了大量注入攻击。

关于 Serverless 下应用程序的注入攻击实例可以参考 OWASP（Open Web Application Security Platform）组织在 2017 年发布的《Top 10 Interpretation for Serverless》报告 [23]。

3.5.2. 针对应用程序依赖库漏洞的攻击

开发者在编写应用程序时不可避免的会引入第三方依赖库，这就引入了一个非常严峻的问题：开发者是否使用了含有漏洞的依赖库。

据 Synk 公司在 2019 年的开源软件安全报告 [24] 中透露，已知的应用程序安全漏洞在过去两年增加了 88%。即便开发者编写的函数只有短短几十行无漏洞的代码，但只要引入了第三方含有漏洞的依赖库，那么该函数也是存在脆弱性的。

此外，引入了第三方依赖库也会实际增加应用部署至服务器的代码总量，例如 python 库，其代码量可能是上千行，node.js 的 npm 包中的代码量就更大了，可能会导致上万行。即便没有已知漏洞，但随着代码量的增多，潜在的风险也相应增加，从而给 Serverless 应用带来了极大安全隐患。

近年来，随着业界对不安全的第三方依赖库的重视，许多行内报告例如 OWASP Top 10[25] 项目提出了使用已知漏洞依赖库的安全风险，这些含有漏洞的依赖库可在 CVE、NVD 等网站上进行查询，例如 Node.js 库 CVE 漏洞列表 [26]、Java 库 CVE 漏洞列表 [27]、Python 库 CVE 漏洞列表 [28]。

3.5.3. 针对应用程序访问控制权限的攻击

针对应用程序访问控制权限的攻击在 Serverless 场景下也同样存在，例如函数对某资源的访问权限、可以触发函数执行的事件等。下面以数据库举例，函数执行业务逻辑时不可避免会对数据库进行 CRUD 操作，在此期间，我们需要给予函数对数据库的读写权限。在不对数据库进行其它操作时，我们应当给予只读权限或关闭其权限，如果此时开发者将权限错误的更改为读写操作，攻击者会利用此漏洞对数据库展开攻击，从而增加了攻击面。

---

[23] https://www.owasp.org/index.php/OWASP_Serverless_Top_10_Project

[24] https://snyk.io/opensourcesecurity-2019

[25] https://www.owasp.org/index.php/OWASP_Serverless_Top_10_Project

[26] https://www.npmjs.com/advisories

[27] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=java

[28] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=python

Serverless 应用可能会由许多函数组成，函数间的访问权限，函数与资源的权限映射可能会非常多，高效率管理权限和角色成为了一项繁琐的问题，许多开发者可能暴力地为所有函数配置单一权限和角色，那么进而会导致单一漏洞扩展至整个应用的风险，这也是当前面临的一大问题。

3.5.4. 针对应用程序数据泄漏的攻击

在应用程序中，敏感数据信息泄漏、应用程序日志泄漏、应用程序访问密钥泄漏、应用程序未采用 HTTPS 协议进行加密等是一些常见的数据安全风险。通过调研发现，这些事件的产生多是由于开发者的不规范操作引起，例如将密钥信息，数据库连接密码等敏感信息硬编码在应用程序中。

需要注意的是，在 Serverless 中以上这些威胁同样存在，但与传统应用程序不同的是：

1. 针对攻击数据源的不同，传统应用只是从单一服务器上获取敏感数据，而 Serverless 架构中攻击者可针对各种数据源进行攻击，例如云存储（AWS S3）或 DynamoDB 等，因此攻击面更广；

2. Serverless 应用由众多函数组成，无法像传统应用程序使用单个集中式配置文件存储的方式，因此开发人员多使用环境变量替代。虽然存储更为简单，但使用环境变量本是一个不安全的行为；

3. 传统的应用开发人员并不具备丰富的 Serverless 的密钥管理经验，不规范的操作易造成敏感数据泄露的风险；

2018 年 6 月，著名开源 Serverless 平台 Apache OpenWhisk 曝出 CVE-2018-11756 漏洞 [29]，应用程序含有漏洞的情形下，攻击者可能会利用漏洞覆盖被执行的 Serverless 函数源代码，并持续影响函数后续的每次执行，如果攻击者对函数代码进行精心伪造，可进一步造成数据泄露、RCE（远程代码执行）等风险。

3.5.5. 针对 Serverless 平台账户的拒绝服务攻击

针对平台账户的攻击主要为 DoW（Denial of Wallet）攻击，顾名思义指拒绝钱包攻击的意思，为拒绝服务 DoS 攻击的变种，目的为耗尽账户账单金额。

Serverless 具备一个重要特性为自动化弹性扩展，开发人员只需为函数调用次数付费，而函数弹性扩展的事情交给了云厂商，这一特性是 Serverless 备受欢迎的原因之一，但为此特性产生的费用通常没

[29] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11756

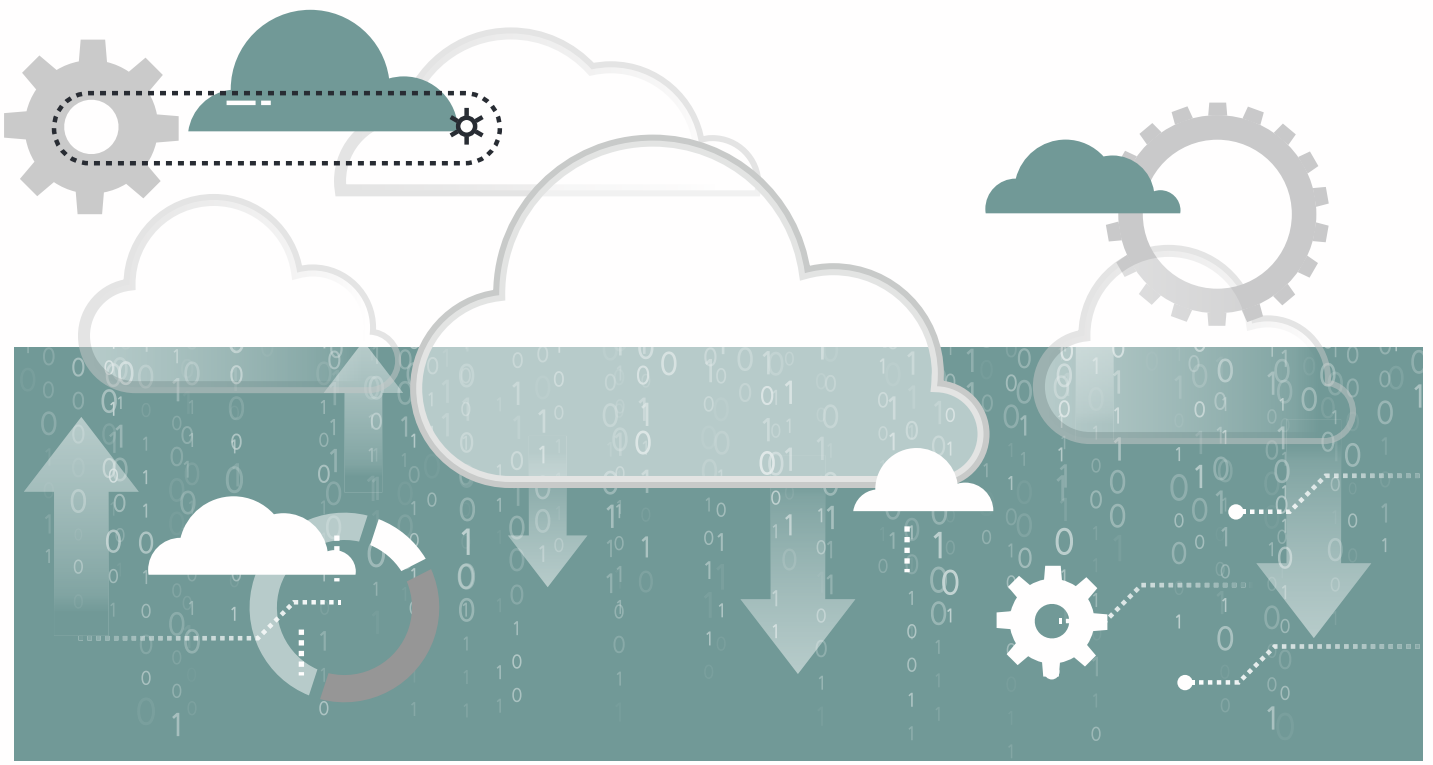
有一定的限制。如果攻击者掌握了事件触发器，并通过 API 调用了大量函数资源，那么在未受保护情况下函数将极速扩展，随之产生的费用也呈指数增长，最终会导致开发者的账户受到重大损失。

2018 年 2 月， NodeJS 的“aws-lambda-multipart-parser”库被曝出 ReDoS 漏洞（CVE-2018-7560）[30], 该漏洞可导致使用了该库的 AWS Lambda 函数运行超时，攻击者可利用此漏洞构造大量并发请求从而耗尽服务器资源或对开发者账户造成 DoW 攻击。

[30] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7560

4

云原生安全防护思路



4.1. 云原生安全设计原则

以 DevSecOps 为体系的敏捷开发体系下，开发安全和运营安全是一体化的。安全运营的负责人（一般是 CIO 或 CISO），则需要考虑如何在给定的安全预算前提下，将整体的 IT 安全防护能力最大化，本节介绍了三个基本原则。

4.1.1. 安全左移

在云原生安全建设的早期，运行时容器生命周期短、业务复杂，而且存在与操作系统虚拟化的环境，现有的物理或虚拟化的安全设备无法工作，因而在这种困境下，增加运行时的安全投入无助于提高整体的安全水平。

国内外在过去两年提出了安全左移（Shift Left）的思路，即在云原生安全建设初步将安全投资更多地放到开发安全，包括安全编码、供应链（软件库、开源软件）安全、镜像（仓库）安全等。这些方面资源大多是白盒的，相应的安全投资相对较少；而且这些资源生命周期较长，如果能保证安全性，攻击者在攻击运行时实例得手后更难持久化。

4.1.2. 聚焦不变

当云原生安全体系大体完成了开发侧的安全建设后，安全团队必然还是要将目光移到运行时，因为这部分永远是攻击者最常用的入侵环节。

然而，云原生运行时存在诸多挑战，例如容器存活时间短、编排业务繁多、服务网格业务交互模式复杂，这些挑战来源于运行时的资源和行为快速变化，无法通过规则等固定的模式进行检测和处置。

既然运行时安全是重要且必要的，那就需要分析这些资源和行为的变化规律，只要规律模式是可预测的，那就可以聚焦在这些不变的规律，从而找到异常和正常的边界。

具体地，容器对应的镜像、文件系统等，都是不变的，由相同或继承的镜像启动的容器的进程和其行为是相似的，用于微服务的容器进程是少数且其行为是可预测的，可以通过学习进行画像的。

4.1.3. 关注业务

微服务所承载的都是各位对外和对内的业务，无服务则更进一步，将容器、依赖库等简化，使得开发者只需编写业务逻辑的代码即可，因而，云原生整个体系中最贴近最终价值的是处于最顶层的业务。

因而，安全团队在完成开发阶段和运营阶段安全机制的部署之后，应该重点关注业务安全，及时检测针对业务的欺诈、滥用行为，如是公有云原生环境可对外第三方用户提供增值服务，获得更高的收益。

4.2. 面向云原生全生命周期的安全防护

报告的开篇我们提到，云原生安全包含了两层含义：“面向云原生环境的安全”和“具有云原生特征的安全”。因此，做好云原生安全，则必然需要使用“具有云原生特性的安全”技术去实现“面向云原生环境的安全”，采用原生安全的方式构建云原生安全防护体系。

云原生安全究其根本是要保证云原生应用的安全，因此，云原生安全防护体系的建设也要围绕云原生应用的生命周期来进行构建，这同样也是敏捷开发中 DevSecOps 所倡导的应用安全模式。

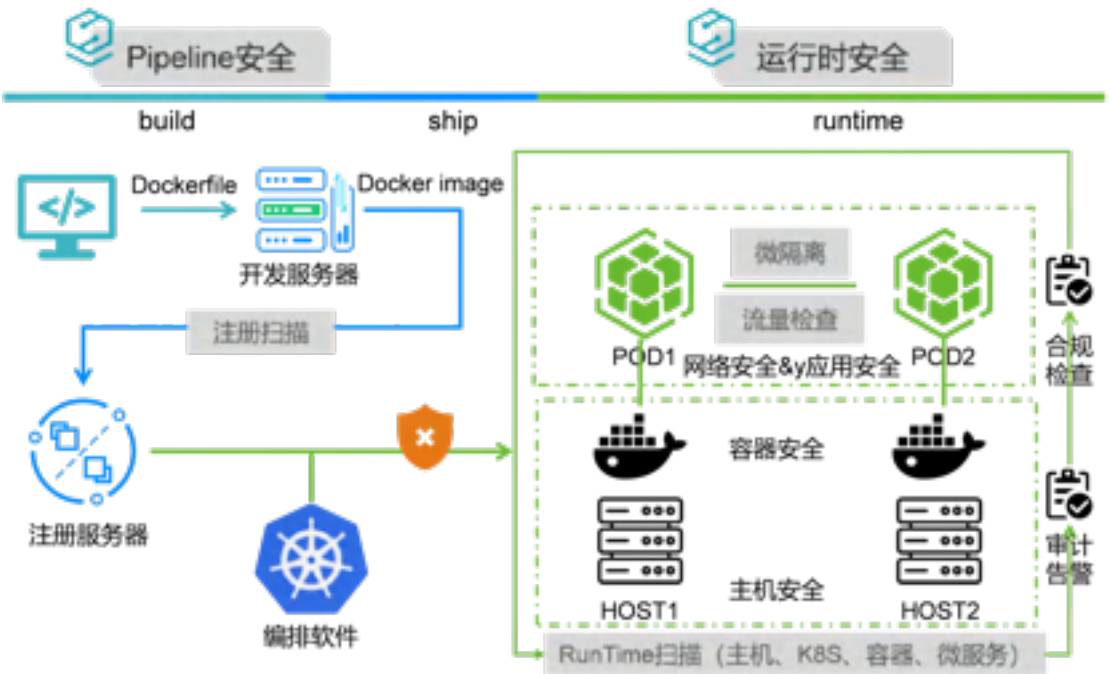


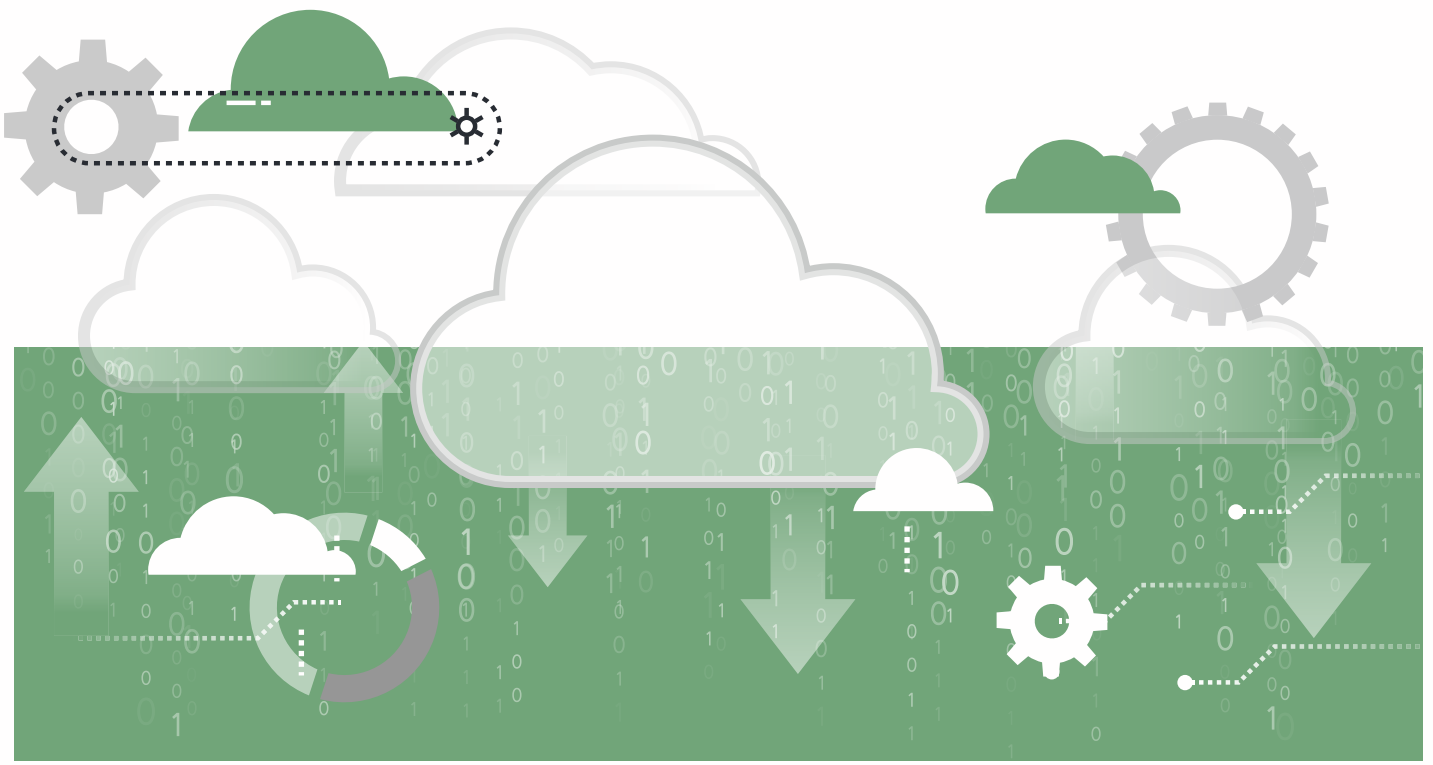
图 4.1　云原生全生命周期的安全防护

云原生应用的整个生命周期，大致可以分为两个阶段：第一就是在开发 / 测试阶段（Development），完成应用源代码的编写、容器镜像的构建以及容器镜像的存储；第二是在生产环境中的上线运营阶段（Operations），完成应用容器的部署、上线运行。

因此，云原生安全防护体系的核心就是覆盖应用的整个生命周期，保证云原生应用在镜像制作、镜像存储、镜像传输、容器运行时的安全。

5

云原生安全加固



5.1. 镜像安全检测与加固

5.1.1. 源代码安全审计

针对容器镜像，采用分析工具与人工审计相结合的方式，根据提供的应用开发过程文档，审阅系统实现的技术方案，对架构的安全性进行审计和评估，分析系统防护薄弱点及可能存在的安全风险。

审计实施人员对系统重要业务场景进行风险分析并审计源代码，如转账、查询等涉及资金和用户敏感信息的功能场景，在人工分析的过程中，实施人员会通过源代码安全审计工具，对全部代码进行自动化审计，以保证源代码安全审计的全面性。

源代码安全审计过程中，除源代码脆弱性审计外，我们还会参照相关标准和规范，对业务实现的合规性进行审计。

5.1.2. 镜像扫描

漏洞的检测是对镜像进行安全加固的重要措施之一，这里的漏洞检测主要还是针对已知的 CVE 进行扫描分析。比如可以使用 Clair、Anchore 等开源工具，或者某些商业的漏洞扫描产品，也同样支持对容器镜像的漏洞扫描。

除了漏洞之外，还有一些通过扫描镜像中的环境变量、操作命令以及端口开放信息来识别恶意镜像的方案，但仍然需要使用者自己基于结果来判断，还不能直接给出一个明确的结果。

5.2. 主机安全加固

5.2.1. 主机基本安全加固

容器与宿主机共享操作系统内核，因此宿主机的配置对容器运行的安全有着重要的影响，比如宿主机中安装有漏洞的软件可能会导致任意代码执行风险，端口无限制开放可能会导致任意用户访问风险，防火墙未正确配置会降低主机的安全性，没有按照密钥的认证方式登录可能会导致暴力破解并登陆宿主机。

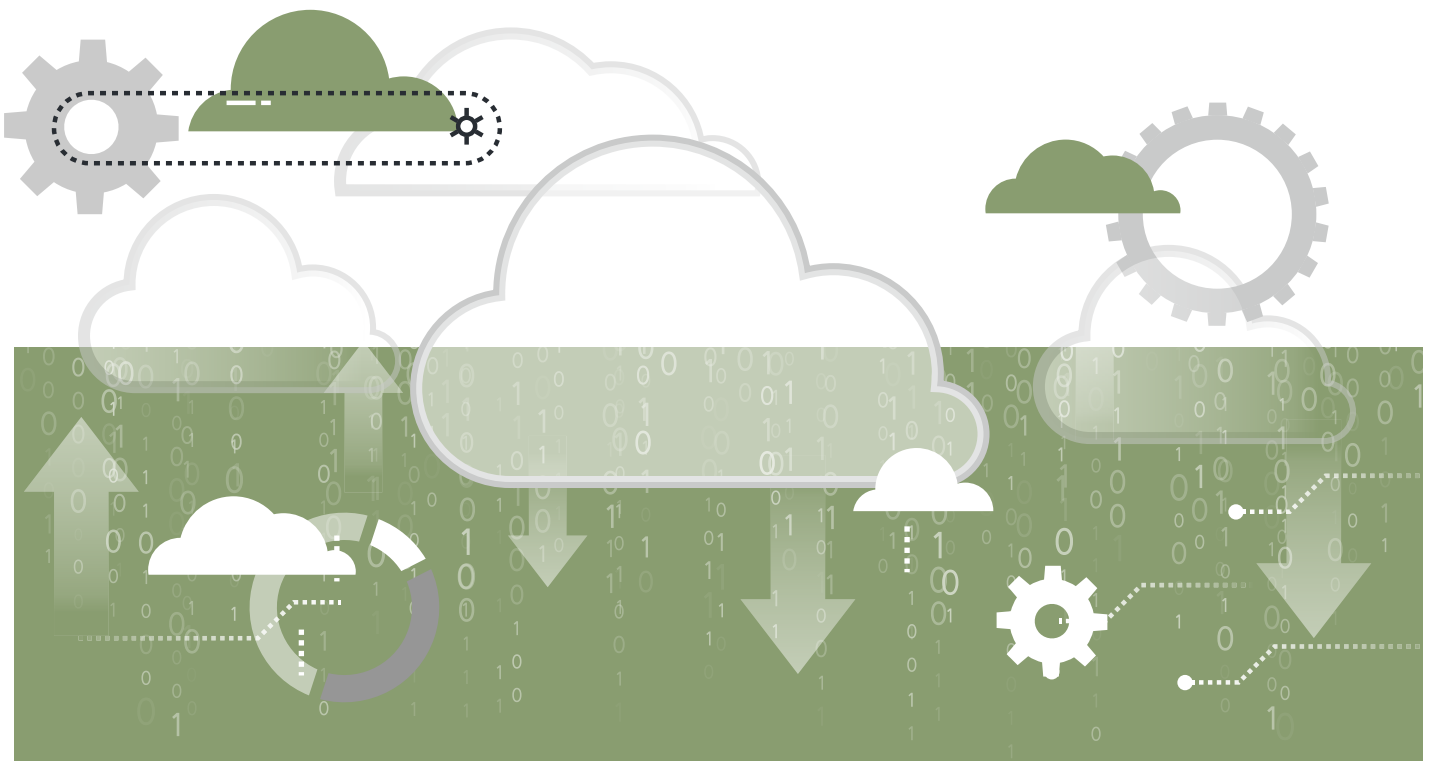
从安全性考虑，容器主机应遵循如最小安装化，不应当安装额外的服务和软件以免增大安全风险；配置交互用户登陆超时时间；关闭不必要的数据包转发功能；禁止 ICMP 重定向；配置可远程访问地址范围；删除或锁定与设备运行、维护等工作无关的账号、重要文件和目录的权限设置；关闭不必要的进程和服务等安全加固原则。

5.2.2. 云原生相关安全加固

Docker 公司与美国互联网安全中心（CIS）合作，制定了 docker 的最佳安全实践，包括主机安全配置、docker 守护进程配置、docker 守护程序配置文件、容器镜像和构建、容器运行安全、docker 安全操作六大项，99 个控制点。几乎覆盖了 docker 安全要求的各个方面。这也是各安全方案和安全工具的重要参考。关于加固，具体可参见《2018 绿盟科技容器安全技术报告》。

6

云原生环境的可观察性



可观察性（Observability）一词最早来源于 Apple 工程师 Cindy Sridharan 的博文《监控与观察》（Monitoring and Observability）[31]。然而，Google 著名的 SRE[32] 体系在这之前就已经奠定了可观察性的理论基础，只不过那时候将这套理论统称为“监控”，SRE 特别的强调了“白盒监控”的重要性，而“白盒监控”正是贴合了可观察性中“主动”的理念。

随着云原生、微服务等新架构、新生态的引入和发展，可观察性越来越多的被提及和重视，而可观察性和监控这两个概念也逐渐的被区分开来。简单来讲，这种区别可以认为是，“监控告诉我们系统的哪些部分是不工作的，而可观察性可以告诉我们那里为什么不工作了”[33]。

6.1. 云原生环境为什么需要实现可观察性

在回答云原生为什么需要可观察性之前，我们先来看一下 CNCF 对云原生的定义，英文原文如下：

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

从定义中可以看出，云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式 API。通过这些技术，可以构建容错性好、易于管理和可观察的松耦合系统，再结合可靠的自动化手段，对系统做出频繁和可预测的重大变更。

CNCF 的这个定义中明确的提出了可观察（Observable）这一特性。而为什么云原生一定需要可观察性，我们从以下几个方面进行分析。

[31] https://copyconstruct.medium.com/monitoring-and-observability-8417d1952e1c

[32] https://landing.google.com/sre/

[33] https://mp.weixin.qq.com/s?__biz=MzAxNTA1OTY4OQ==&mid=2457362109&idx=1&sn=b90b15ded62d1f6f52545d025ef80014&scene=21#wechat_redirect

6.1.1. 云原生主机系统行为更复杂

传统的应用部署，无论是基于物理主机，还是基于虚拟机，其操作系统上承载的应用相对固定。而对于大多数采用单体架构的系统，其系统内部的通信也相对简单。

到了云原生时代，从其代表技术来看，容器化的基础设施使得应用自身变的更快、更轻，一台主机上可以快速部署运行几十个、甚至上百个容器。而 Kubernetes 等容器编排平台，又提供了良好的负载均衡、任务调度、容错等管理机制。这样，在云原生中，一台主机上应用的部署密度以及变化频率，较传统环境，有着巨大的变化。因此，需要可观察性来清晰的发现和记录主机快速变化的应用行为。

另外，应用架构的微服务化，使得应用之间的访问关系变得异常复杂，客户端的一次服务请求，通常会产生大量的包括服务和中间件在内的众多调用关系。清晰地观察到这种调用关系，无论是对于应用性能提升、故障定位还是安全检测，都有着重要的意义。

6.1.2. 可见才可防

正所谓“未知攻焉知防”，面对云原生架构下的大规模集群以及海量灵活的微服务应用，如果不知道集群中都运行了什么，服务都在做什么事情，又何谈保护和防范。

从云原生的最终目标来看，通过自动化手段，实现敏捷的松耦合系统。因此，云原生安全也一定是符合这种自动化目标的，自动化的安全检测就需要有详细准确的运行状态数据作为支撑，为自动化的云原生安全提供充足的决策依据。可观察性恰恰天然地提供了这样的能力。

6.1.3. 助力等保 2.0 可信计算需求

从合规角度来看，等保 2.0 将可信提升到一个新的高度，等保一级到四级均提出了关于可信的要求，包括：计算环境可信、网络可信以及接入可信。等保 2.0 的四级要求尤其对应用可信提出了明确的动态验证需求，如何在不影响应用的功能、性能，保证用户体验的前提下，做到应用的动态可信验证成为了重要的挑战。

而在云原生环境中，解决这个问题的核心在于，如何准确地选择应用的可信度量对象、如何高性能地确定指标的度量值，以及如何收集和管理验证这些基准值，这些都是对云原生环境实现可观察的重要意义和应用价值。

6.2. 云原生可观察性需要观察什么

我们知道，计算机系统一直遵循分层设计的理念，云原生同样也不例外。要实现对整个云原生的可观察性，可以分别逐层实现对应的可观察。

从基础设施层来看，这里的可观察性跟传统的主机监控会有一些相似和重合，比如对计算、存储、网络等主机资源的监控，进程、磁盘 IO、网络流量等系统指标的监控等。对于云原生的可观察，这些传统的监控指标依然存在，但是考虑到云原生中采用的容器、服务网格、微服务等新技术、新架构，其可观察性又会有新的需求和挑战。比如在资源层面要实现 CPU、内存等在容器、Pod、Service、Tenant 等不同层的识别和映射关系；在进程的监控上要能够精准识别到容器，甚至还要细粒度到进程的系统调用、内核功能调用等层面；在网络上，除了主机物理网络之外，还要包括 Pod 之间的虚拟化网络，甚至是应用之间的 Mesh 网络流量的观察。

从应用层来看，微服务架构下的应用，使得主机上的应用变的异常复杂，这既包括应用本身的平均延时，应用间的 API 调用链，调用参数等，同时还包括应用所承载的业务信息，比如业务调用逻辑、参数、订单数量，商品价格等信息。

6.3. 云原生可观察性实现手段

实现可观察性，通常可以有多种手段，不同的方法，其侧重点往往略有差别，下图是可观察性领域经典的一张分类图，其描述了几种方法对应的作用域以及相互之间的关联和区别。本节接下来将简要分析一下这几种实现方法。

图 6.1　可观察性组成

6.3.1. 日志（Logging）

日志展现的是应用运行产生的事件或者程序在执行的过程中产生的一些记录，可以详细解释系统的运行状态。日志描述了一些离散的、不连续的事件，对于应用程序的可见性是很好的信息来源，日志同样也为对应用程序的分析提供了精确的数据源。

但是日志数据存在一定的局限性，它依赖于开发者暴露出来的内容，而且其存储和查询需要消耗大量的资源，往往需要使用过滤器减少数据量。

6.3.2. 指标（Metrics）

Metrics 与 Logging 有所不同，Logging 提供的是显示的数据，而 Metrics 是通过数据的聚合，来对一个程序在特定时间内的行为进行衡量。Metrics 是可累加的，他们具有原子性，每个都是一个逻辑计量单元，或者一个时间段内的柱状图。Metrics 可以观察系统的状态和趋势，但对于问题定位缺乏细节展示。

6.3.3. 追踪（Tracing）

追踪面向的是请求，可以轻松分析出请求中的异常点，但与 Logging 有相同的问题就是资源消耗较大。通常也需要通过采样的方式减少数据量。比如一次请求的范围，也就是从浏览器或者手机端发起的任何一次调用，一个流程化的东西，我们需要轨迹去追踪。追踪的最大特点就是，它在单次请求的范围内处理信息。任何数据、元数据信息都被绑定到系统中的单个事务上。

6.4. 云原生环境下的服务追踪

在基于微服务的云原生架构中，客户端的一次服务调用，会产生大量的包括服务和中间件在内的众多调用关系。针对这些复杂的调用过程进行追踪，对于微服务的安全性分析、故障定位、以及性能提升等，有着重要的作用。因此，分布式追踪系统是微服务架构下不可或缺的重要组成部分。

6.4.1. 服务追踪概述

当前的互联网服务，大多数都是通过复杂的、大规模的分布式集群来实现，而随着云原生、微服务等架构的逐步成熟，传统的单体架构设计，向着更加松耦合的微服务架构进行演进。同时，考虑到微服务架构下的负载均衡以及高可用等设计，服务间通信以及调用关系的复杂度，将变得异常庞大。

我们先看一个搜索查询的例子，比如前端发起的一个 Web 查询请求，其目标将可能是后端的上百

个查询服务，每一个查询都有自己的 Index。这个查询可能会被发送到多个后端的子系统，这些子系统分别用来进行广告的处理、拼写检查或是查找一些图片、视频或新闻这样的特殊结果。根据每个子系统的查询结果进行筛选，进而得到最终结果，汇总到返回页面上。

总的来说，服务追踪有助于在微服务架构下有效地发现并且解决系统的性能瓶颈问题，以及在请求失败时在这样错综复杂的服务之中准确地进行故障定位。此外，服务追踪也可以用于发现服务中的威胁。一旦前端暴露的服务被攻击者攻陷，或者是内部某个服务已被攻击者攻陷，那么，面对系统内部微服务之间庞大复杂的调用关系，这些调用是否全部是完成这次搜索所必须发生的业务联系，是否存在 API 探测、API 滥用，是否存在针对某个服务的 DDoS，客户端服务发送的请求是否包含注入攻击，如何在海量的正常业务调用逻辑中发现非法的异常调用请求，这些问题均可以通过服务追踪结合有效的分析算法进行快速的检测发现。

6.4.2. 服务追踪的实现

分布式追踪技术对于微服务的故障定位是十分必要的，那么分布式追踪技术是如何实现的呢？若要对一个微服务业务系统进行分布式追踪，会产生两个基本问题。第一，业务系统运行时可能会产生很多脏数据或发生数据丢失，需要在这种环境下准确地生成追踪数据。第二，面对成百上千的服务所生成的追踪数据，则需要设计合适的收集与存储方案。下面具体介绍分布式追踪技术是怎样解决这两个问题的。

在介绍追踪数据的生成之前，需要明确追踪数据中的两个基本概念：跨度（Span）和追踪链路（Trace）。跨度是追踪数据中基本数据结构，它代表追踪中的一次操作，跨度中包含操作名、跨度 ID、当前跨度的前一个跨度（根跨度除外）、操作起始与终止时间戳、属性键值对等信息，下面是一个跨度信息示例：

```
{
    operationName: "api/getCommodity",
    spanId: "7630fb07cb6e2d6",
    tags: [
        {
            key: "commodityName",
            value: "Huawei"
        }
    ]
    startTime: 1599818026974512,
    duration:29021
}
```

追踪链路是以跨度为根结点的树形数据结构，在微服务中，从客户端发起一次 API 调用时往往后面会产生多次服务间的 API 调用以完成此次操作，因此追踪链路代表一次完整操作，其中包含了很多子操作。

生成追踪数据所要做的，是针对每次操作生成跨度以及将跨度串联成追踪链路。对于跨度的生成，追踪器会在网络请求或数据库访问发生时抓取有效数据并对此次操作生成一个独有的跨度 ID。对于将跨度串联成追踪链路，在一次跨度信息生成后，追踪器会将跨度 ID 加进下一次网络请求中，当请求被接收时，追踪器会检验请求中是否包含跨度 ID 信息，若包含，则创建新的跨度作为请求中跨度的子跨度，否则创建一个新的根跨度。随后，有父子关系的跨度信息会被发送到收集器中，收集器为这组有父子关系的跨度数据加上链路 ID，具有同一链路 ID 的跨度数据组成一次完整的追踪链路。至此，包含跨度信息和追踪链路的追踪数据被成功生成。

6.4.3. 服务追踪的应用

自 Google Dapper 首先提出分布式链路追踪的设计理念以来，许多分布式追踪工具不断涌现，这些工具应用在微服务业务系统的资源监控，故障定位，服务依赖分析，服务组织架构理解等方面，为微服务业务系统的运维工作提供了极大便利。

当前，常见的分布式追踪工具包括 Dapper，Zipkin，Jaeger，Skywalking，Canopy，鹰眼，Hydra，Pinpoint 等，其中常用的开源分布式追踪工具为 Zipkin，Jaeger，Skywalking 和 Pinpoint。这些分布式追踪工具可分为三个大类：基于 SDK 的，基于探针的，基于 Sidecar。下面将简要介绍这三类追踪工具的使用方法与各自的优缺点：

基于 SDK 的分布式追踪工具。以 Jaeger 为例，Jaeger 提供了大量可供追踪使用的 API，通过侵入微服务业务的软件系统，在系统源代码中添加追踪模块实现分布式追踪。此类工具可以最大限度地抓取业务系统中的有效数据，为运维工作提供了足够的可参考指标，但其通用性较差，需要针对每个服务进行重新实现，部署成本较高，工作量较大。

基于探针的分布式追踪工具。以 Skywalking Java 探针为例，在使用 Skywalking Java 探针时，需将探针文件打包进服务容器镜像中，并在镜像启动程序中添加 -javaagent agent.jar 命令，以完成 Skywalking 在微服务业务上的部署。Java 探针的实现原理为字节码注入，将需要注入的类文件转换成 byte 数组，通过设置好的拦截器注入到正在运行的程序中。Java 探针通过控制 JVM 中类加载器的行为，

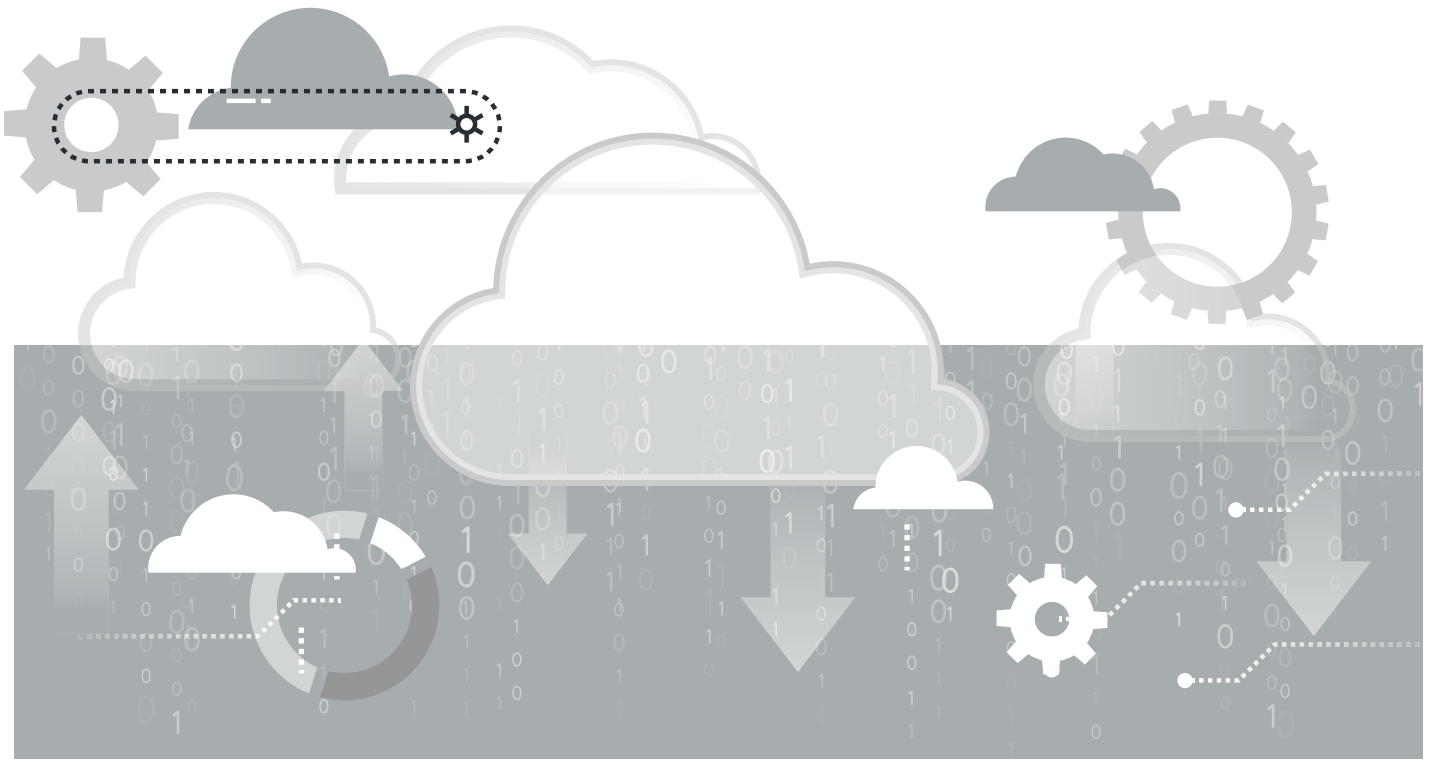
侵入运行时环境实现分布式追踪。此类工具无需修改业务系统的源代码，相对 SDK 有更好的通用性，但其可获取的有效数据相对 SDK 类工具较少。

基于 Sidecar。Sidecar 作为服务代理，为其所管理的容器实现服务发现，流量管理，负载均衡和路由等功能。在流量管理过程中，Sidecar 可以抓取进出容器的网络请求与响应数据，这些数据可以记录该服务所完成的一次单个操作，可与追踪中的跨度信息对应，因此可将 sidecar 视为一种基于数据收集的分布式追踪工具。Sidecar 无需修改业务系统代码，也不会引入额外的系统的开销。但由于 sidecar 所抓取的跨度不包含追踪链路上下文，要将 sidecar 所抓取的跨度数据串联成追踪链路是很困难的。

虽然分布式追踪技术在应用方面已经取得了一些进展，但其仍然存在着一定的局限性。当前的分布式追踪工具或者需要侵入微服务软件系统的源代码，或者需要侵入业务系统的镜像与运行环境，或者在生成的跨度信息与追踪链路的准确性与完整性上仍有缺失。因此，如何实现低开销，低侵入地构建准确且完整的追踪信息仍需要许多后续工作。

7

微隔离实现零信任的云原生网络



随着企业网络基础设施的日益复杂，尤其是云计算等虚拟化网络应用的普及，这种复杂性超越了传统网络边界安全的防护方法。基于传统物理、固定边界的网络安全也被证明是不够用的，“数据中心内部的系统和网络流量是可信的”这一假设是不正确的。网络边界的安全防护一旦被突破，即使只有一台机器被攻陷，攻击者也能够在所谓“安全的”数据中心内部横向移动。

NIST 在 2020 年 8 月，发布了最新的零信任架构 [34]，在零信任安全模型中，会假设环境中随时可能存在攻击者，不能存在任何的隐形信任，必须不断的分析和评估其资产、网络环境、业务功能等的安全风险，然后制定相应的防护措施来缓解这些风险。在零信任中，这些防护措施通常要保证尽可能减少对资源（比如数据、计算资源、应用和服务等）的访问，只允许那些被确定为需要访问的用户和资产访问，并且对每个访问请求的身份和安全态势进行持续的认证和授权。

微隔离作为实现零信任的关键技术之一，在云原生网络安全建设中，同样起着重要的作用。本章将重点介绍如何在云原生网络中实现零信任的微隔离系统。

7.1. 概述

在云原生环境中，尤其是云原生网络安全的建设和规划中，以零信任的架构和思路，实现云原生网络的微隔离和访问控制是必要的。

7.1.1. 什么是微隔离？

微隔离（Micro-Segmentation）最早是 VMware 为应对虚拟化隔离技术提出来的。Gartner 更是从 2016 年开始，连续三年将其列为年度 Top10 的安全技术和项目（Micro-Segmentation and Flow Visibility，微隔离和流量可视化），通过对网络流量的可视化和监控，让运维人员能够详细了解系统内部网络数据包的流向，进而使微隔离能够更好的设置安全策略。

微隔离，顾名思义就是一种更细粒度的网络隔离技术，其核心能力的诉求也是聚焦在东西向流量，对传统环境、虚拟化环境、混合云环境、容器环境等东西向流量进行隔离和控制，重点用于阻止攻击者进入数据中心网络或者云虚拟网络后进行的横向移动。

微隔离有别于传统的基于边界的防火墙隔离技术，微隔离技术通常是采用一种软件定义的方式，其

[34] https://csrc.nist.gov/publications/detail/sp/800-207/final

策略控制中心与策略执行单元是分离的，而且通常具备分布式和自适应等特点。策略控制中心是微隔离系统的核心控制单元，可视化展现内部系统之间以及业务应用之间的网络访问关系，并且能够按照角色、标签等快速的对需要隔离的工作负载进行分类分组，并且高效灵活的配置工作负载以及业务应用之间的隔离策略。策略执行单元主要用于网络流量数据的监控以及隔离策略的执行，通常实现为虚拟化设备或者主机上的 Agent。

7.1.2. 云原生为什么需要微隔离？

在传统网络，或者虚拟化网络中，已经存在了像 VLAN、VPC 之类的网络隔离技术，但是，这些隔离技术，更偏向于针对确定性网络，或者是租户网络的隔离。

在云原生环境中，容器或者微服务的生命周期相比较传统网络或者租户网络，变的短了很多，其变化频率要高很多。微服务之间有着复杂的业务访问关系，尤其是当工作负载数量达到一定规模以后，这种访问关系将会变得异常的庞大和复杂。因此，在云原生环境中，网络的隔离需求已经不仅仅是物理网络、租户网络等资源层面的隔离，而是变成了服务之间应用层面的隔离。

因此，就网络隔离而言，一方面需要能够针对业务角色，从业务视角更细粒度的实现微服务之间的访问隔离；同时，还要针对业务之间的关系，在隔离基础上实现访问控制，从而降低网络攻击在东西向上的横向移动。另一方面，这种灵活快速的网络状态变化，也带来了全新的隔离和访问控制策略更新的需求，隔离策略与访问控制策略，需要能够完全自动化的适应业务和网络的快速变化，实现快速高效的部署和生效。

7.2. 云原生网络组网架构

随着容器技术的不断发展和演进，实现容器间互联的云原生网络架构也在不断的进行优化和完善。从 Docker 本身的动态端口映射网络模型，到 CNCF 的 CNI 容器网络接口，再到 Service Mesh + CNI 层次化的 SDN 网络。

7.2.1. 端口映射

Docker 自身在网络架构上，默认采用桥接模式，即 Linux 网桥模式，创建的每一个 Docker 容器，都会桥接到这个 docker0 的网桥上，形成一个二层互联的网络。同时还支持 Host 模式、Container 模式、None 模式的组网，具体组网细节，在《2018 容器安全技术报告》中已有详细介绍，本文将不再赘述。

Docker 在这种组网架构下，容器内的端口通过在主机上进行端口映射，完成相关的通信支持。

7.2.2. 容器网络接口（CNI）

CNI（Container Network Interface）是 Google 和 CoreOS 主导制定的容器网络标准，综合考虑了灵活性、扩展性、IP 分配、多网卡等因素。CNI 旨在为容器平台提供网络的标准化。不同的容器编排平台能够通过相同的接口调用不同的网络组件。这个协议连接了两个组件：容器编排管理系统和网络插件，具体的事情都是插件来实现的，包括：创建容器网络空间（Network Namespace）、把网络接口（Interface）放到对应的网络空间、给网络接口分配 IP 等。

目前采用 CNI 提供的网络方案一般分为两种，Overlay 组网方案和路由组网方案。

Overlay 组网

以 Flannel、Cilium、Weave 等为代表的容器集群组网架构，默认均采用基于隧道的 Overlay 组网方案。比如，Flannel 会为每个主机分配一个 Subnet，Pod 从该 Subnet 中分配 IP，这些 IP 可在主机间路由，Pod 间无需 NAT 和端口映射就可以跨主机通讯。而在跨主机间通信时，会采用 UDP、VxLAN 等进行隧道封装，形成 Overlay 网络。

路由组网

以 Callico 为代表的容器组网架构，并没有使用 Overlay 的方式做报文转发，而是提供了一个纯三层的网络模型。在这种三层通信模型中，每个 Pod 都通过 IP 直接通信。Callico 采用 BGP 路由协议，使得所有的 Node 和网络设备都记录下全网路由，每个容器所在的主机节点就可以知道集群的路由信息。整个通信的过程中始终都是根据 BGP 协议进行路由转发，并没有进行封包、解包的过程，这样转发效率就会快很多。然而这种方式会产生很多无效的路由，同时对网络设备路由规格要求较大。

7.2.3. 服务网格

服务网格（Service Mesh）当前在云原生网络中，是一个非常流行的架构方案。与 SDN 类似，Service Mesh 通过逻辑上独立的数据平面和控制平面来实现微服务间网络通信的管理。

但是 Service Mesh 并不能替代 CNI，它需要与 CNI 一起提供层次化微服务应用所需要的网络服务。这就可以看出，Service Mesh 与 SDN 还是有着一定的区别，SDN 主要解决的是 L1-L4 层的数据包转发问题，而 Service Mesh 则主要是解决 L4/L7 层微服务应用间通信的问题。二者可以通过互补配合的方式，

共同实现云原生网络架构。

7.3. 云原生网络的微隔离实现技术

前文介绍了常见的容器网络组网架构，什么是微隔离，以及为什么云原生网络建设中需要通过微隔离来实现应用服务之间的访问控制管理。接下来，本章将介绍几种常见的云原生环境中实现微隔离的技术。

目前，对于微隔离的技术实现，还没有统一的产品标准，属于比较新的产品形态。在 IaaS 层面的微隔离机制一般而言有三种形态：基于虚拟化技术（Hypervisor）、基于网络（Overlay、SDN），以及基于主机代理（Host-Agent），而在容器环境中，则有很大的不同。

首先，容器是非常轻量级的，且一个宿主机中容器数量较多，因而，每个容器上部署一个主机代理的方式是非常昂贵不实用的。

其次，基于虚拟化技术和基于网络的技术都是在访问的主体和客体间部署网络访问控制策略，区别只是与 IaaS 系统的对接机制不同。而在容器环境中，已有标准的 CNI 组网机制和 Network Policy 网络访问控制策略，因而可融合为一类。

最后，云原生环境中存在大量的微服务，微隔离应更多关注微服务业务，而非简单容器隔离，如 Service Mesh 架构中的 Sidecar 模式做反向代理的应用微隔离成为新的形态。

下面简单介绍云原生中两种微隔离机制。

7.3.1. 基于 Network Policy 实现

Network Policy 是 Kubernetes 的一种资源，用来说明一组 Pod 之间是如何被允许互相通信，以及如何与其它网络 Endpoint 进行通信。Network Policy 使用标签（Label）选择 Pod，并定义选定 Pod 所允许的通信规则。每个 Pod 的网络流量包含流入（Ingress）和流出（Egress）两个方向。

在默认的情况下，所有的 Pod 之间都是非隔离的，可以完全互相通信，也就是采用了一种黑名单的通信模式。当为 Pod 定义了 Network Policy 之后，只有 Policy 允许的流量才能与对应的 Pod 进行通信。通常情况下，为了实现更有效、更精准的隔离效果，会将这种默认的黑名单机制更改为白名单机制，也就是在 Pod 初始化的时候，就将其 Network Policy 设置为 deny all，然后根据服务间通信的需求，制定细粒度的 Policy，精确的选择可以通信的网络流量。下面是一个简单的 Network Policy 示例。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

CNI 针对 Network Policy 只是制定了相应的接口规范，Kubernetes 的 Network Policy 功能也都是由

第三方插件来实现的，因此，通常只有支持 Network Policy 功能的网络插件或者安全插件，才能进行相应的网络策略配置，比如 Calico、Cilium 等。

各种插件在 Network Policy 的实现上，通常会采用 iptables 的方式，通过在主机上配置一系列的 iptables 规则，来实现不同的隔离策略。但是，随着云原生网络的不断发展，承载业务规模的不断增大，在主机上构建成千上万的网络规则，会让 iptables 不堪重负，而且还要频繁的更新、生效策略，导致这种实现方法越来越难以满足大规模复杂网络的隔离需求。

另外一种实现方法，就是采用 eBPF，主要解决的是大规模云原生网络的性能问题。这种实现模式下，控制平面将相应的隔离策略通过 eBPF 指令作用于对应的网卡，进而控制数据包的通过与阻断。这是 Cilium 网络插件一直主张的技术路线，Calico 在 3.13 版本中，也增加了基于 eBPF 的数据平面模式，例如，Calico 的 eBPF 数据平面将 eBPF 程序挂载到每个 Calico 的接口以及 Pod 的数据或隧道接口的 tc hook 上，这样就能够及早发现数据包，并通过绕过 iptables 和内核通常进行的其他包处理流程，进而实现一种快速包处理路径。

7.3.2. 基于 Sidecar 实现

另外一种微隔离的实现方式，就是采用 Service Mesh 架构中的 Sidecar 方式。Service Mesh（比如 Istio）的流量管理模型通常和 Sidecar 代理（比如 Envoy）一同部署，网格内服务发送和接收的所有流量都经由 Envoy 代理，这让控制网格内的流量变得异常简单，而且不需要对服务做任何的更改，再配合网格外部的控制平面，可以很容易的实现微隔离。

Istio 官方提供的安全架构 [35] 如下所示，图中展示了 Istio 的认证和授权两部分，Istio 的安全机制涉及诸多组件。控制平面由核心组件 Istiod 提供，其中包含密钥及证书颁发机构（Certificate authority，CA）、认证授权策略、网络配置等；数据平面则由 Envoy 代理、边缘代理（Ingress 和 Egress）组件构成。

Istio 的认证机制借助控制平面 Istiod 内置的 CA 模块，实现为服务网格中的服务提供签名证书，同时可将证书分发至数据平面各个服务的 Envoy 代理中，当数据平面的服务间建立通信时，服务旁的 Envoy 代理会拦截请求并采用签名证书和另一端服务的 Envoy 代理进行双向 TLS 认证从而建立安全传输通道，保障了数据安全。

[35] https://istio.io/latest/docs/concepts/security/arch-sec.svg

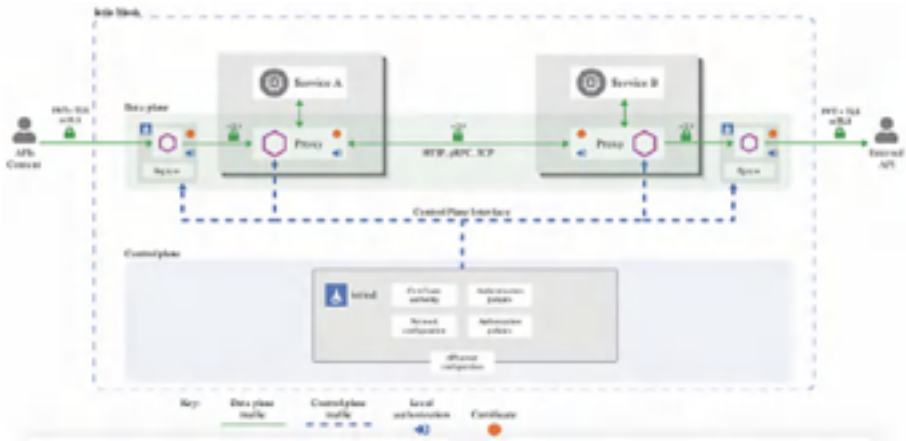


图 7.1　Istio 安全架构

有了身份认证机制作为基础，Istio 还提供授权机制[36]，下图为 Istio 的授权流程。管理员 Administrator 使用 yaml 文件指定 Istio 授权策略并将其部署至 istiod 核心组件中，Istiod 通过 API Server 组件监测授权策略变更，若有更改，则获取新的策略，Istiod 将授权策略下发至服务的 Sidecar 代理，每个 Sidecar 代理均包含一个授权引擎，在引擎运行时对请求进行授权。

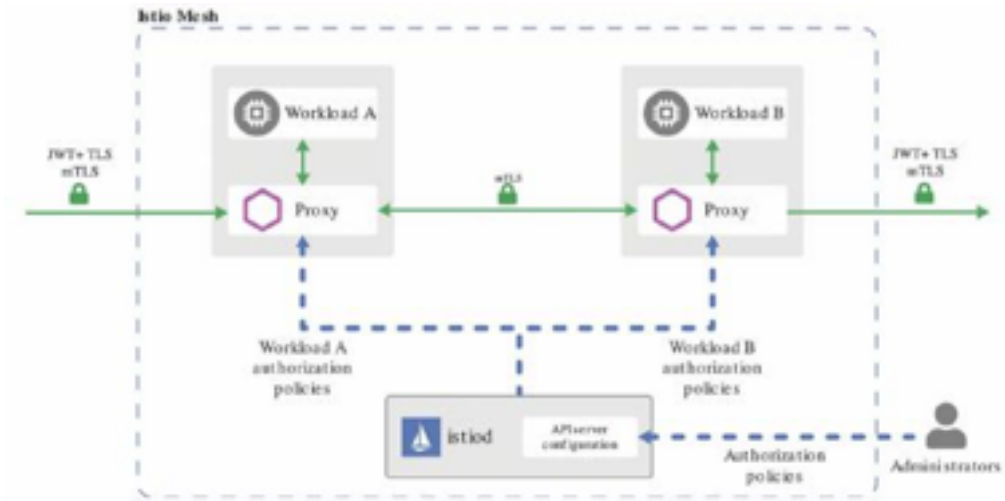


图 7.2　Istio 授权架构图

[36]　https://istio.io/latest/docs/concepts/security/authz.svg

7.4. 案例介绍

Cilium 是一种开源的云原生网络实现方案，与其他网络方案不同的是，Cilium 着重强调了其在网络安全上的优势，可以透明的对 Kubernetes 等容器管理平台上的应用程序服务之间的网络连接进行安全防护。

Cilium 在设计和实现上，基于 Linux 的一种新的内核技术 eBPF，可以在 Linux 内部动态插入强大的安全性、可见性和网络控制逻辑，相应的安全策略可以在不修改应用程序代码或容器配置的情况下进行应用和更新。

Cilium 在其官网上对产品的定位称为“eBPF-based Networking, Observability, and Security”，因此可以看出，其特性主要包括这三方面：（1）提供 Kubernetes 中基本的网络互连互通的能力，实现容器集群中包括 Pod、Service 等在内的基础网络连通功能；（2）依托 eBPF，实现 Kubernetes 中网络的可观察性以及基本的网络隔离、故障排查等安全策略；（3）依托 eBPF，突破传统主机防火墙仅支持 L3、L4 微隔离的限制，支持基于 API 的网络安全过滤能力。Cilium 提供了一种简单而有效的方法来定义和执行基于容器 /Pod 身份（Identity Based）的网络层和应用层（比如 HTTP/gRPC/Kafka 等）安全策略。

Cilium 的参考架构 [37] 如下图所示，Cilium 位于容器编排系统和 Linux Kernel 之间，向上可以通过编排平台为容器进行网络以及相应的安全配置，向下可以通过在 Linux 内核挂载 eBPF 程序，来控制容器网络的转发行为以及安全策略执行。

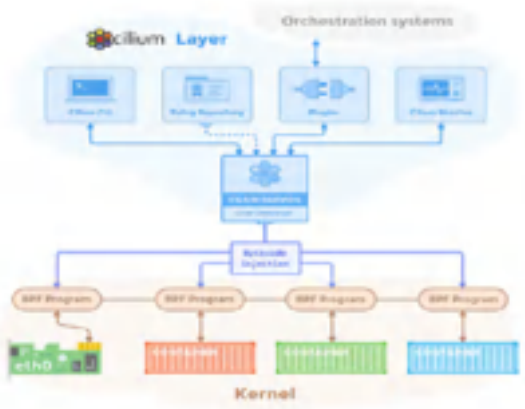


图 7.3　Cilium 架构

[37]　https://cilium.readthedocs.io/en/stable/concepts/overview/

Cilium Agent 在进行网络和安全的相关配置时，采用 eBPF 程序进行实现。Cilium Agent 结合容器标识和相关的策略，生成 eBPF 程序，并将 eBPF 程序编译为字节码，将它们传递到 Linux 内核。

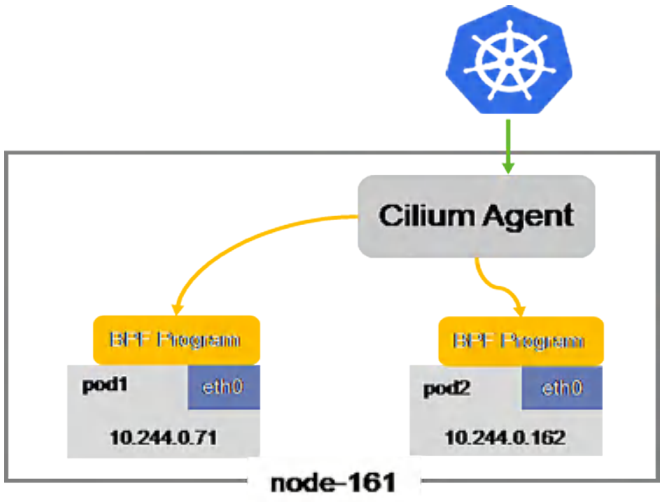


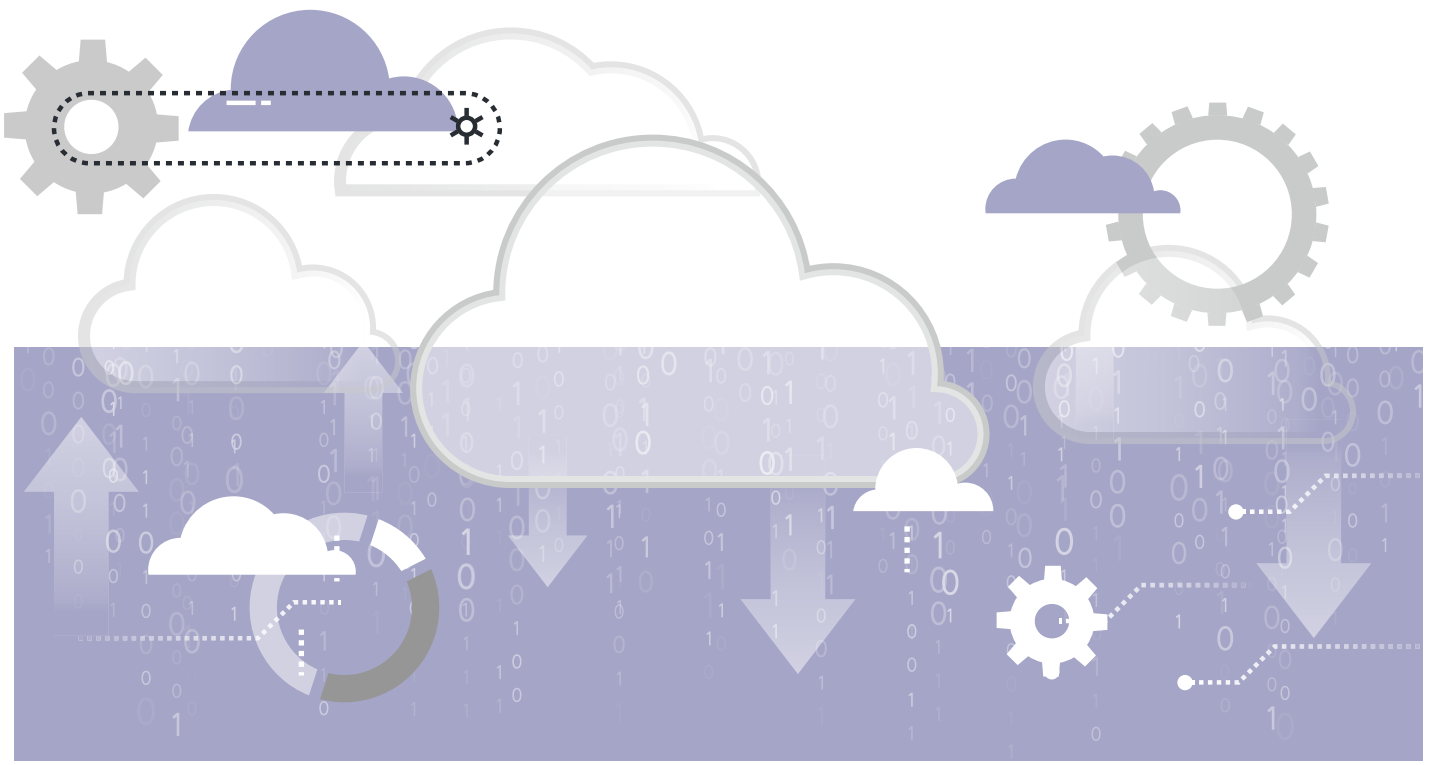
图 7.4　Cilium 部署架构

默认情况下，Cilium 与其他网络插件一样，提供了整个集群网络的完全互联互通，用户需要根据自己的应用服务情况设定相应的安全隔离策略。如下图所示，每当用户新创建一个 Pod，或者新增加一条安全策略，Cilium Agent 会在主机对应的虚拟网卡驱动加载相应的 eBPF 程序，实现网络连通以及根据安全策略对数据包进行过滤。比如，可以通过采用下面的 NetworkPolicy 实现一个基本的 L3/L4 层网络安全策略。

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
description: "L3-L4 policy to restrict deathstar access to empire ships only"
metadata:
  name: "rule1"
spec:
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
  ingress:
  - fromEndpoints:
    - matchLabels:
        org: empire
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
```

8

云原生异常检测



8.1. 异常分类

根据异常表现位置的不同，我们可以将其划分为网络侧异常和主机侧异常。这与传统的网络侧入侵检测系统（NIDS）、主机侧入侵检测系统（HIDS）的分类是一致的。

网络异常指的是集群东西或南北向流量上体现出来的异常，又可以细分为：行为型异常（例如：后端数据库被发现有集群外通信的南北向流量，短时间内产生大量网络连接等）和载荷型异常（例如：某 Pod 被检测出 Webshell 流量等）。

为了区别传统的**主机异常**，我们将主机侧的异常限制在容器级别，也就是集群内部 Pod 自身所表现的异常。根据异常涉及因素的不同，我们可以将其划分为行为型异常（例如：有新进程运行、进程以新用户身份运行等）和资源型异常（例如：CPU/ 内存资源占用率过高、可用存储空间减小过快等）。

8.2. 云原生网络侧异常检测方法

由于云原生环境下生态的开放性和容器网络的复杂性，传统 NIDS 不能很好地监控检测到云原生环境中网络流量中隐藏的入侵行为。云原生环境需要与之相适应的网络入侵检测机制。

具体来说，云原生网络入侵检测机制需要实现对 Kubernetes 集群每个节点上 Pod 相关的东西及南北向流量进行实时监控，并对命中规则的流量进行告警。告警能够定位到哪一台节点上、哪一个命名空间内的哪一个 Pod。

针对这些需求，我们可以采用基于 Pod 的流量检测方法：在每个节点上部署一个流量控制单元和策略引擎，流量控制单元负责将特定 Pod 的流量牵引或镜像到策略引擎中，策略引擎对接入流量进行异常检测（类似传统 IDS）并向日志系统发送恶意流量告警。集群内另有一编排引擎负责监控日志系统的告警事件，根据告警定位到节点、命名空间、Pod，然后构建并下发阻断规则给流量控制单元，形成从检测到阻断的完整闭环。该机制如下图所示：

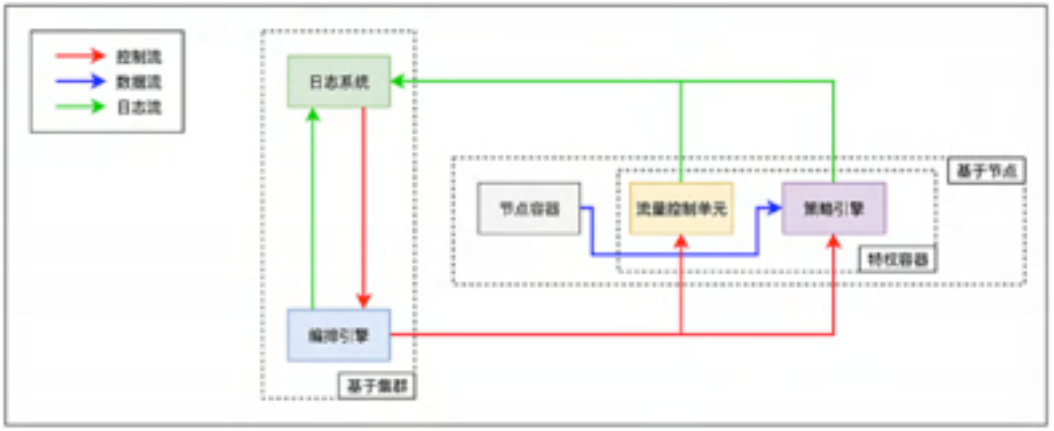


图 8.1　基于 Pod 的流量检测方法示意图

8.3.　云原生主机侧异常检测方法

8.3.1.　基于进程模型的异常行为检测

一般来说，不同应用的业务场景可能不同，但是对于给定集群来说，其在一段生产运营时间内的业务模式是相对固定的。这意味着，业务依赖的程序和模块是确定的。

我们可以进一步得出，该集群在一段时间内运行的进程是确定的，进程行为模式在一个合理的范围内。在容器环境下，业务通常以镜像为单位进行交付，因此，无论容器启动多少个，创建于相同镜像的容器内部的进程行为总是相似的。

基于上述合理假设，我们能够在镜像级别收集集群内部业务正常运行期间的进程集合及进程行为、属性集合，采用自学习的方式，自动构建出合理的镜像进程模型。学习结束后，可以采用这些模型进行容器内异常检测。

检测系统定期获取容器进程数据，当收到尚未建立进程行为模型的容器进程数据时，将自动为该镜像建立并开始训练一个新的模型。在经过指定的训练学习周期（例如一周、一个月等），模型训练结束，自动转为检测模式。此后，再收到该镜像相关的容器进程数据时，检测系统将利用模型针对这些数据进行异常检测。系统检测出异常后，向运营人员发出告警通知，运营人员基于业务实际环境判断告警是否正确，如果告警正确则进行相应处置；否则，运营人员可以对检测模型进行小范围修正。

整个流程分为学习、检测两个阶段：

容器集群是由大规模容器组成的运行环境，容器是基于镜像创建而成的动态运行时客体。镜像与容器之间通常是一对多的关系，换言之，在容器集群环境中，来自同一个镜像的容器往往不止一个。

检测系统定期收集集群各宿主机节点上所有运行容器内部的所有进程数据，第一次收到数据后，利用本次数据为该次数据涉及到的每一个镜像初始化进程行为模型。行为模型是对该镜像生成的容器在运行时的正常行为范围的界定。

在学习期间，此后每一次收到与该镜像关联的容器进程信息时，利用这些新增信息不断更新该镜像的模型，即更新模型中的进程列表，直到学习期结束，将模型状态改为“已就绪”。

模型的学习流程如下图所示：

图 8.2　模型学习流程

在学习期结束后，每一次收到数据后，检测系统将该次数据中属于同一镜像的数据与数据库中该镜像的进程行为模型做匹配，如果待测数据中的进程属性不在模型的镜像进程列表描述的对应进程的属性中，或待测数据的 CPU、内存等资源使用情况超出了镜像进程列表中记录的对应进程的历史最大 CPU、内存资源使用率，则判定异常，并输出告警。

基于模型的检测流程如下图所示：

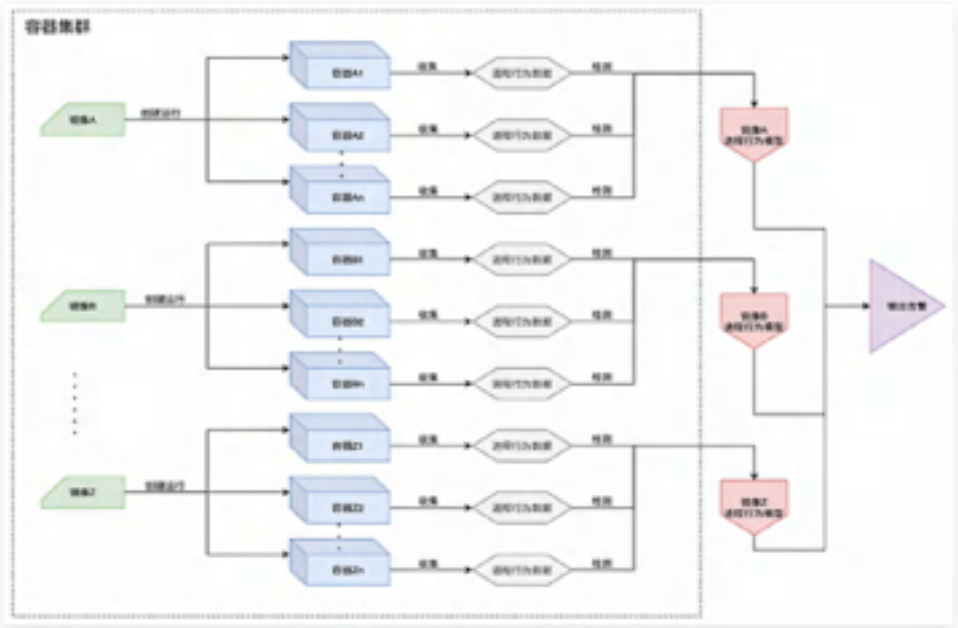


图 8.3　基于模型的检测流程

8.3.2.　基于系统调用的攻击行为检测

几乎任何一个有意义的进程在运行期间都会执行系统调用（System call），恶意进程也不例外，甚至调用得更多更为频繁。另外，恶意进程的调用方式往往具有区别于正常进程的特征。

例如，2019 年著名的 runC 容器逃逸漏洞 CVE-2019-5736，它的特征是，在利用过程中需要以 /proc/self/exe 为路径参数执行 open 系统调用，在后续过程中，还会以 /proc/self/fd/ 为路径参数前缀采用写方式执行 open 系统调用，以覆盖宿主机上的 runC 二进制文件。这样的参数内容在容器内业务进程中非常罕见的。因此，特定系统调用 + 特定参数就是该漏洞的明显特征。

再如，“脏牛”（CVE-2016-5195）漏洞可以用于容器逃逸，在漏洞的触发过程中，恶意进程需要同时高频调用 mmap 和 madvise 函数，或 mmap 和 ptrace 函数。“脏牛”属于竞态条件漏洞，其显著特征是攻击者不能保证自己某一次的攻击一定会成功，只能根据系统状态、CPU 类型等估计出一个大概的成功率。因此，为了成功触发漏洞，攻击者必须以尽可能快的速度进行系统调用。这种高频性是值得防守者关注的明显特征。

同 CVE-2019-5736 漏洞利用一样，多数攻击行为都可以通过直接检测某次系统调用或连续几次系统调用的异常来判断是否发生了某一类攻击行为。

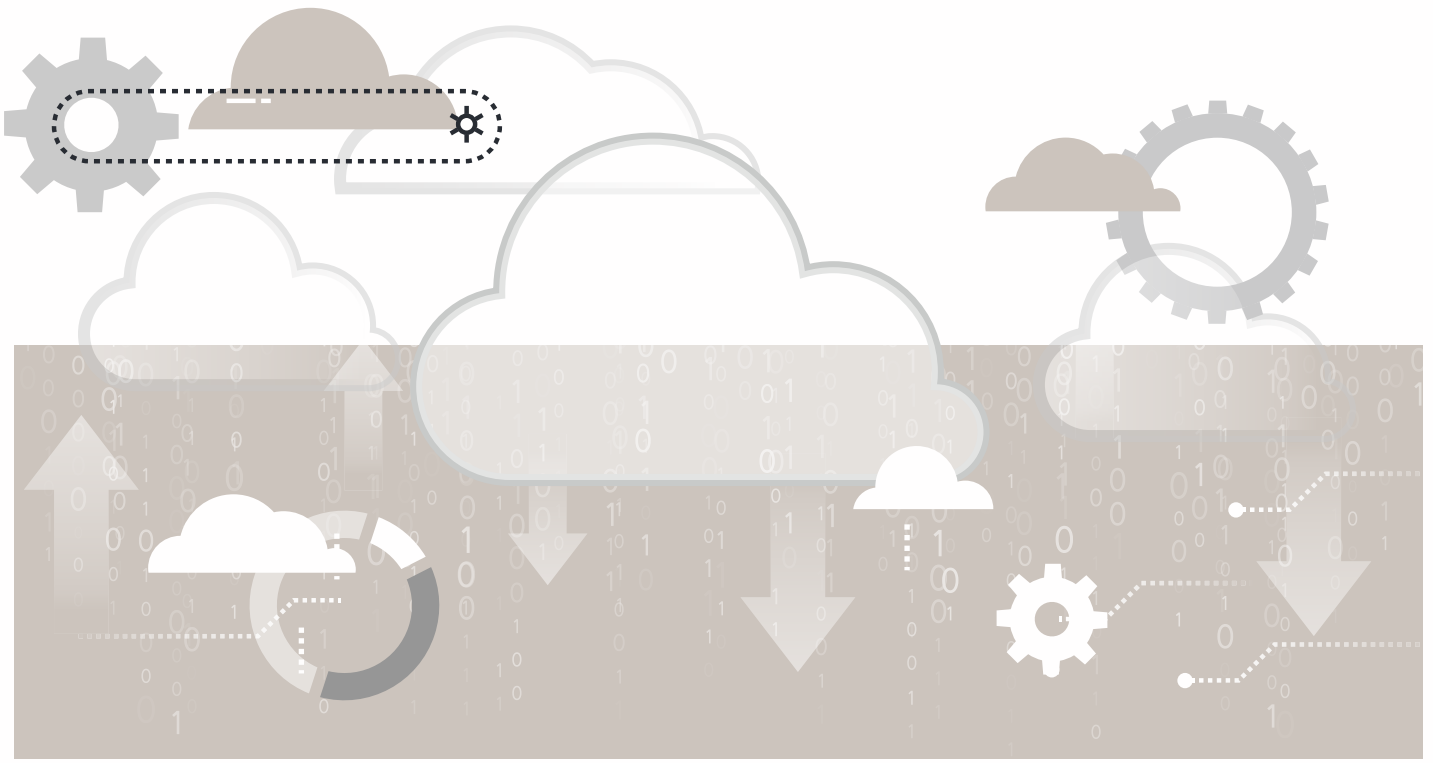
然而，与“脏牛”漏洞利用一样，还有一些攻击行为不能简单地通过单次或有限次系统调用的异常来判断。很可能每一个系统调用都是常见的，参数都是正常的，但是高频调用就能以一定概率触发漏洞。对于这类攻击行为，我们就不能简单地采用上面的方式来检测了，概率式的攻击需要统计式的检测。例如，我们可以检测一段时间窗口内系统调用的类型和频度，如果某个可能触发漏洞的系统调用集合内各函数的频度都超过了某阈值，且这种调用模式在正常业务中是不常见的，那么这段时间内就很可能发生了某漏洞的攻击利用行为。

8.4. 小结

在各行各业积极上云的今天，如何及时准确发现云原生环境内部的异常威胁并进行告警和处置，是云原生平台开发运维和应急响应团队必须考虑的问题。对大规模云原生环境进行及时有效的异常检测监控已经成为业务云原生化过程中的迫切需求。

9

云原生应用安全



9.1. 应用 API 安全

随着云计算模式的不断发展，企业拥抱数字化转型已逐渐成为行业共识，与此同时，应用也趋于云原生化，为适应现代应用架构趋势，包括移动访问、微服务设计模式、本地 / 云端部署的应用导致 API 数量倍增，可以说云原生应用中大部分交互模式以从 Web 请求 / 响应转向各类 API 请求 / 响应，例如 RESTful/HTTP、gRPC 等。因而，API 安全在云原生环境中就显得尤为重要。

9.1.1. 云原生应用的 API 安全现状分析

云原生应用的 API 安全基本归类于网络层面公开的 API 存在的安全问题，微服务场景中，API 安全防护变得更加复杂，传统的单一入口防护已无法完全覆盖 API 安全范畴。

近些年来，企业将各自的 Web 应用逐步向云原生应用的方向迈进，云原生应用中，API 既充当外部与应用的访问入口，也充当应用内部服务间的访问入口，可见 API 已然被用来发展业务和推动创新。然而，企业面对大量的 API 设计需求，其相应的 API 安全方案往往不够成熟，从而引起了 API 滥用的风险。据 Gartner 预测，到 2022 年，API 滥用将成为导致企业 Web 应用数据泄露最频繁的攻击载体 [38]。

API 的实现及处理方式通常为客户端 / 服务端的形式，客户端存在的 API 滥用现象最终是由于服务端的安全威胁导致，我们通过调研发现，这些安全威胁包括注入攻击、DDoS 攻击、认证授权失败、敏感信息泄露、中间人攻击、参数篡改等，关于每种威胁的攻击场景及实现原理由于篇幅限制，此处不做赘述，详细内容可参考 OWASP 在 2019 年发布的 API 安全 Top10 报告 [39]。

9.1.2. 云原生应用的 API 安全检测

9.1.2.1 针对 API 的脆弱性检测

针对 API 的脆弱性检测通常可使用扫描器进行周期性的漏洞扫描，国内各大安全厂商均提供扫描器产品，例如绿盟科技的远程安全评估系统（RSAS）[40] 及 Web 应用漏洞扫描系统（WVSS）[41]，除此之外，

[38] .https://www.cloudvector.com/owaspwhitepaper/#:~:text=Gartner%20predicts%20that%20%E2%80%9Cby%202022,breaches%20for%20enterprise%20web%20applications%E2%80%9D.&text=Risks%20due%20to%20uninspected%20API,to%20uncontrolled%20third%20party%20APIs

[39] https://owasp.org/www-project-api-security/

[40] https://www.nsfocus.com.cn/html/2019/209_1009/66.html

[41] https://www.nsfocus.com.cn/html/2019/206_0911/8.html

我们也可以使用其它商业版扫描器，例如 AWVS（Acunetix Web Vulnerability Scanner）、AppScan、Burp Suite、Nessus 等。

9.1.2.2 针对 API 的攻击检测

面对针对 API 的威胁，目前多使用传统 API 网关的形式进行防护，例如 Kong、Zuul、Orange 等都有其各自的 API 安全解决方案。不过随着应用的云原生化，许多云原生 API 网关应运而生，例如 Kong、Ambassador、Gloo 均可基于 Kubernetes 进行部署。

虽然 API 网关在一定程度上抵御了应用的 API 威胁攻击，但其防护是南北向流量，服务网格内部的东西向流量检测防护并未得到有效缓解。理想的解决方案应当是东西 / 南北向的全量检测。

针对此需求，业界已有了相应的解决方案，主要是云原生网关与服务网格的结合，其中服务网格负责东西向流量的检测防护，关于服务网格有采用主流的 Istio，也有采用 Kong 的服务网格 Kuma。

Istio 主要通过其内置的 Envoy 过滤器实现东西向流量检测，而 Kuma 则依托 Kong 的安全插件实现相应检测防护。笔者将 Gloo +Istio、Ambassador+Istio、Kong+Kuma 这三种解决方案的安全能力进行了对比，如

以下表格所示：

表 9.1　云原生环境下 API 防护方案安全能力对比

方案组合	Gloo+Istio	Ambassdar+Istio	Kong+Kuma
防护维度	东西向 / 南北向	东西向 / 南北向	东西向 / 南北向
WAF 支持	支持	支持	支持
访问控制	支持	支持	支持
授权机制	支持	支持	支持
认证机制	支持	支持	支持
证书管理	支持	支持	支持
网站锁	不支持	不支持	不支持
反爬虫	不支持	不支持	不支持
机器流量检测	不支持	不支持	支持
数据丢失防护	支持	不支持	不支持
CORS	支持	支持	支持

方案组合	Gloo+Istio	Ambassdar+Istio	Kong+Kuma
XSS	支持	不支持	不支持
CSRF	支持	支持	不支持
DDOS	支持	支持	不支持
黑白名单限制	支持	支持	支持
限速	支持	支持	支持
行为特征分析 AI	不支持	不支持	不支持
入侵检测	不支持	不支持	不支持
mTLS	支持	支持	支持

从以上表格统计来看，Gloo+Istio 解决方案支持的安全项更多些，Kong+Kuma 解决方案支持的安全项最少。

9.2. 应用业务安全

前文从网络，应用等层面介绍了微服务所面临的的威胁。除此之外，微服务架构还面临着许多业务层面的安全威胁。这些安全威胁往往会影响业务的稳定运行，同时会给业务系统带来经济损失。

9.2.1. 业务安全问题分析

具体来说，应用所面临的安全问题主要包括三类，即业务频率异常，业务参数异常，业务逻辑异常。我们以一个微服务架构的电商系统为例进行介绍业务安全，该电商系统的典型流程如图 9.1 所示。

图 9.1 电商系统业务流程图

业务频率异常。针对一个或者一组 API 的频繁调用。如前所述，业务系统往往通过图形验证码的方式来避免机器人刷单的操作。攻击者可以绕过验证码所对应的微服务，直接对订单进行操作，进而实现

机器刷单，对电商进行薅羊毛。

业务参数异常。API 调用过程中往往会传递相关的参数。参数的取值根据业务场景的不同会有不同的取值范围。例如商品数量必须为非负整数，价格必须大于 0 等。如果 API 对相应参数的监测机制不完善，那么攻击者往往可以通过输入异常参数导致业务系统受到损失。例如在电商系统中，如果商品价格只在商品介绍服务中进行校验，而在订单管理和支付服务中没有进行校验，那么攻击者就可以通过直接调用订单管理和支付服务的 API 来将订单价格修改为 0 元或者负值，给业务系统造成损失。

业务逻辑异常。相比于前两类异常，这类异常一般较为隐蔽。攻击者采用某些方法使 API 调用的逻辑顺序出现异常，包括关键调用步骤缺失、颠倒等。例如在电商系统中，攻击者可以利用漏洞绕过交费的步骤直接提交订单。这样就会出现业务逻辑出现关键步骤缺失的情况。对于前述业务频率异常中的验证码绕过异常，也属于业务逻辑异常。

9.2.2. 业务异常检测

针对上述业务层面安全问题，基于基线的异常检测是一类比较有效的方法：首先建立正常业务行为与参数的基线，进而找出偏移基线的异常业务操作，其中，基线的建立需要结合业务系统的特性和专家知识共同来完成。

在电商系统中，业务参数基线主要基于专家知识来建立。例如商品价格不仅与商品本身相关，也与时间和各类优惠活动等相关。这类基线需要运维人员持续的维护。对于业务逻辑基线的建立，由于业务系统在正式上线运行以后，其操作逻辑一般不会有较大的变化，同时异常操作所占的比例较少。因此可以采集业务系统历史的操作数据，结合统计分析与机器学习的方法建立业务逻辑的基线。相比于人工方法，这种方法可以提高基线建立的效率，有效减轻运维人员的工作量。

为此，可利用在 6.4.3 中所提到的分布式追踪工具采集到的数据，针对上述三种业务异常场景，设计并实现了业务异常检测引擎，如图 9.2 所示。其中，采集模块主要用于采集业务系统的运行数据，训练模块主要针对业务系统历史数据进行训练以提取行为特征数据，检测模块主要对正在运行的业务系统进行异常检测。

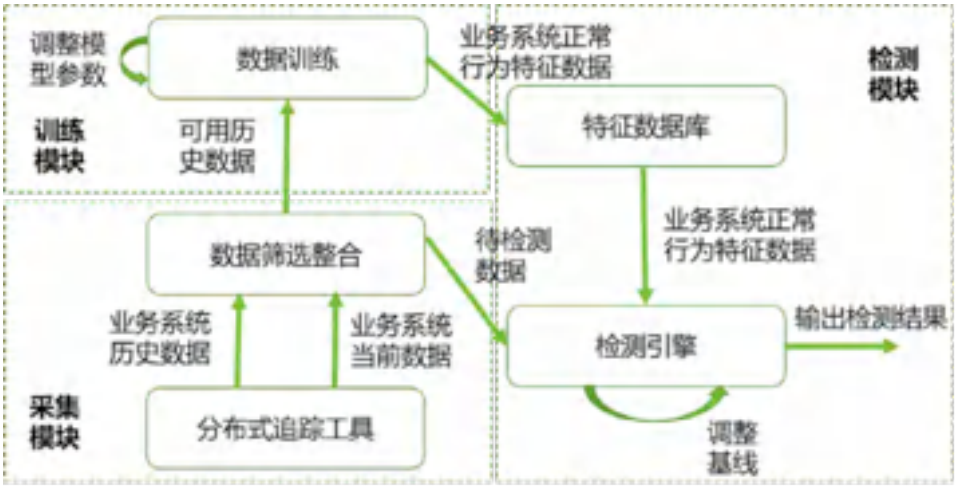


图 9.2　业务异常检测引擎设计图

检测引擎中每部分的具体功能为：

分布式追踪工具。主要为采集微服务业务系统运行时生成的数据。当前常用的开源分布式追踪工具为 Jaeger 与 Skywalking，此外 sidecar 也可以采集相应数据。经过对测试系统进行的采集实验，相比其他两种采集工具，Jaeger 可获取的数据字段最多，能够检测的异常场景最丰富，然而，Jaeger 需要在业务系统的源代码中进行插桩，对开发团队而言有较强的侵入性。相反，sidecar 模式没有代码和镜像的侵入性，但通过反向代理截取流量的模式也决定了它不能获得丰富的上下文，如 6.3.3 的追踪 API 调用关系树（TraceID）是无法获得的。如何利用侵入性更低的采集工具收集到的数据来实现覆盖更多场景的异常检测仍需要很多后续工作。

数据筛选与整合模块。此模块主要功能为过滤掉数据集中的脏数据，以及提取出可以表示业务系统行为的数据。在微服务下，可以表示业务系统行为的数据为 API 调用关系树、服务名、操作名、HTTP POST 参数等。

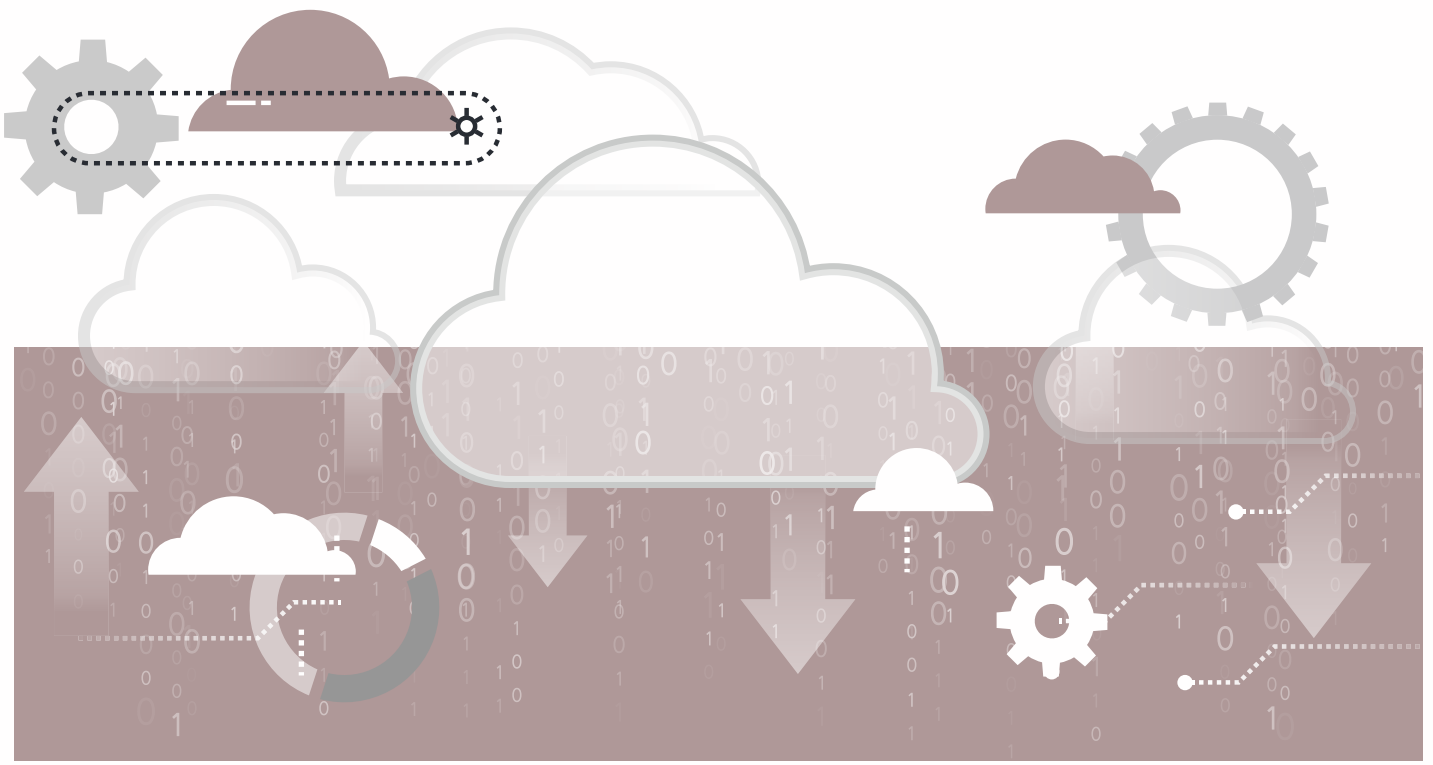
数据训练模块。将预处理后的历史数据利用机器学习或统计学的方法，训练出业务系统中的正常行为，并生成与业务系统正常行为匹配的特征数据。这里进行训练的先验知识为，我们认为业务系统中大量存在的行为是正常行为，而数量很少的行为是异常行为。在训练过程中，需要根据专家知识对训练结果的检验不断调整训练模型的参数。

检测引擎。将业务系统当前数据与特征数据库中数据进行检索匹配，并利用序列相似性计算等方法

找出特征数据库中与当前行为最为匹配的特征数据。检测引擎需要将特征数据与当前数据的相似性与基线进行比较，若比较结果显示当前行为与正常行为的差异在基线限制范围内，则为正常行为，若超出基线限制范围，则判定为异常行为。对于基线，首先需要根据专家知识设置合理的初始基线，并根据不同场景，或利用无监督模型自行调整基线，或由运维人员手动维护基线。

10

服务网格安全防护



在 3.4 节中我们已经提出了服务网格存在的安全风险，本节我们将以 Istio 举例，为大家介绍服务网格的安全防护部分。

10.1. Istio 和 API 网关协同的全面防护

Istio 的防护措施多是基于网格内部的东西流量，对于服务网格的南北流量而言，Istio 采取的解决方案为使用边缘代理 Ingress 与 Egress 分别接管用户或外界服务到服务网格内部的入 / 出站流量，Ingress 与 Egress 实则为 Istio 部署的两个 Pod，Pod 内部为一个 Envoy 代理，借助 Envoy 代理的安全 Filter 机制，可在一定程度上对恶意 Web 攻击进行防护，但现有的 Envoy 安全 Filter 种类相对较少，面对复杂变化场景下的 Web 攻击仍然无法应对，可行的解决方案为在服务网格之外部署云原生 API 网关，具体如下图所示：

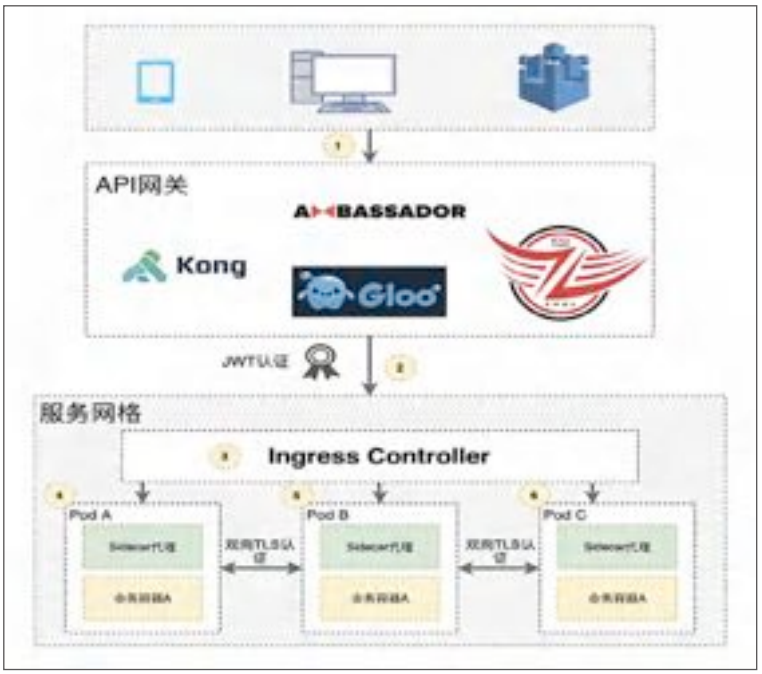


图 10.1　Istio 与 API 网关结合防护图

在安全功能上，云原生 API 网关可提供全方位的安全防护，例如访问控制、认证授权、证书管理、机器流量检测、数据丢失防护、黑白名单限制、常见 Web 攻击等等，在这些有效防护基础上服务网格的南北流量安全得到了控制。

此外，该解决方案的好处还在于服务网格内部的东西流量无需通过外部网关层，这样可以从边缘到端点进行一站式防护。

10.2. Istio 与 WAF 结合的深度防护

WAF 作为一款抵御常见 WEB 攻击（包括前述的 API 威胁）的主流安全产品，可以有效对 Web 流量进行深度防护，并且随着云原生化概念的普及，国内外安全厂商的容器化 WAF 产品也在迅速落地，未来容器化 WAF 与服务网格的结合将会在很大程度上提升微服务安全。

根据近期市场调研，Signal Sciences[42]、Fortiweb[43]、Wallarm[44]、Radware[45] 这几家公司已有了各自的容器化 WAF 解决方案。值得注意的是 Signal Sciences 公司的解决方案支持 WAF 服务与 envoy 或 Istio 结合，其设计如图 10.5 所示，该方案主要运用到了 Envoy 的 Filter 机制，通过 External Authorization HTTP filter 可以将流经业务容器的东西 / 南北向流量引流至 WAF 容器，从而可阻断恶意请求，保护了微服务的安全。

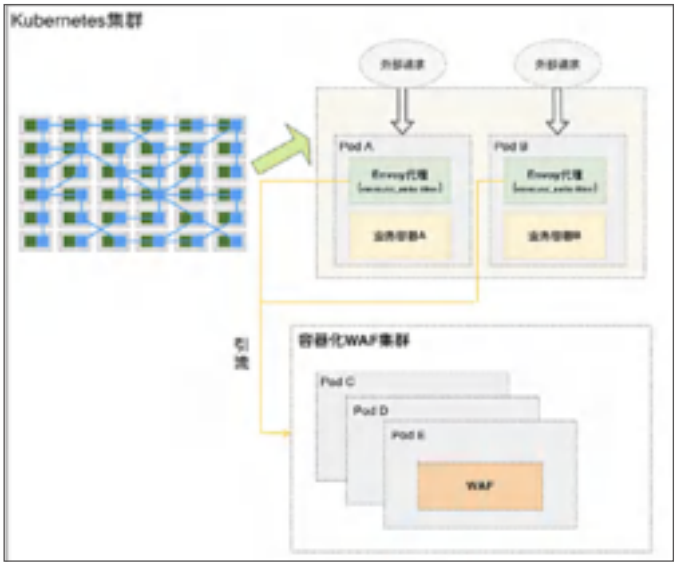


图 10.2 云原生 WAF 与 Istio 结合防护图

[42] https://www.signalsciences.com/
[43] https://www.fortinet.com/products/web-application-firewall/fortiweb
[44] https://wallarm.com/
[45] https://www.radware.com/

此方案带来的好处是对业务入侵较小，实现较为容易、且容器化 WAF 规模不会随用户业务更改而更改。但同时也有一些弊端，比如需要单独部署容器化 WAF、Envoy 引流模块的性能问题、引流方式对 WAF 处理的延迟等都是需要考虑的问题。

另一种解决方案是 Radware 提出的 Kubernetes WAF 方案，该方案基于 Istio 实现，其中 WAF 被拆分为 Agent 程序和后端服务两部分，Agent 程序作为 Sidecar 容器置于 Pod 的 Envoy 容器和业务容器间，该 Sidecar 的主要作用为启动一个反向代理，以便将外部请求代理至 Pod 外部的 WAF 后端服务中，如下图所示。该套方案带来的好处是无需关心外部请求如何路由至 Pod、与 Istio 结合的理念更接近云原生化、实现了以单个服务为粒度的防护。但同时也存在着一些不足，例如流量到达业务容器前经历了两跳，这在大规模并发场景下可能会影响效率。

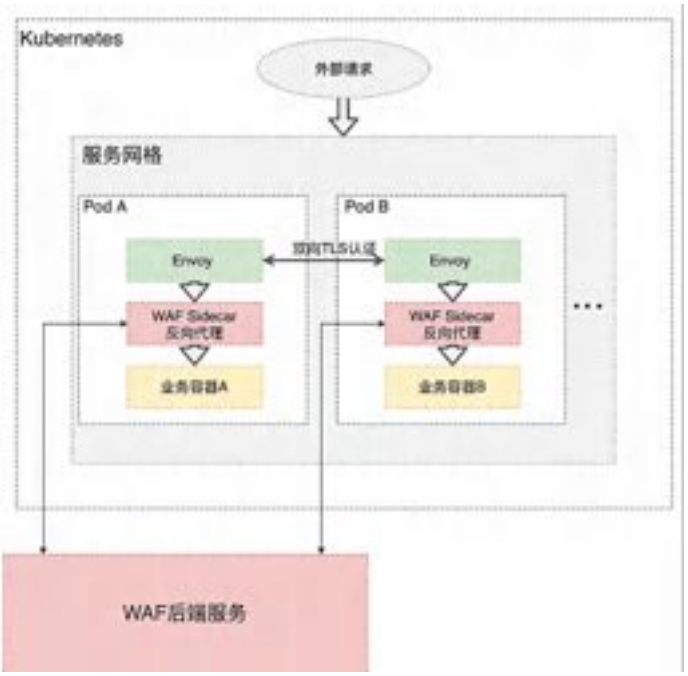


图 10.3　WAF Sidecar 化示意图

10.3.　小结

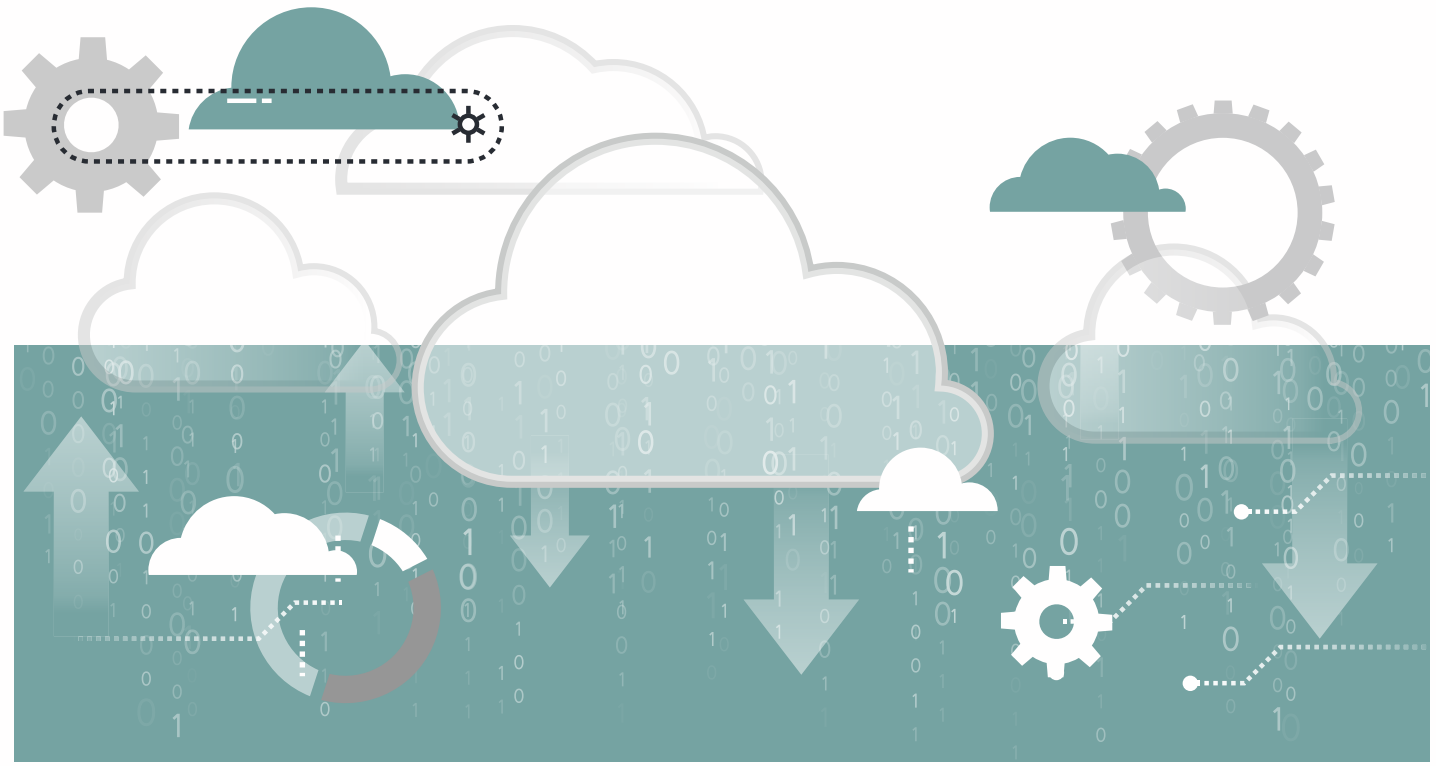
服务网格为微服务治理带来了优势，例如降低代码耦合度、将应用层功能下沉至基础设施层，将复杂代码从应用中抽离等，但一个新技术的诞生也必定伴随着安全风险，服务网格流量多为微服务的东西

向流量，如何防护服务间的通信安全、数据安全成为了我们需要关注的问题。

Istio 实现了面向东西向流量的基本的认证授权和安全通信功能，为了保证整个服务网格的应用安全，还应考虑 Istio 与 API 网关、WAF 等安全机制的联动。

11

Serverless安全防护



由 3.5 节内容可以看出，Serverless 面临的威胁主要包含函数的代码漏洞、依赖库漏洞、访问控制、数据安全和 Serverless 的平台账户这五方面，那么针对 Serverless 防护我们也应从这五点出发一一进行防护。

11.1. 应用程序代码漏洞防护

应用程序代码漏洞防护应当从两方面考虑，一是安全编码，二是使用自动化检测工具。

安全编码需要开发者具备安全编码的能力。首先，由于 Serverless 函数的执行为事件触发，因此针对不同的事件源，我们都应该视为不可信，采取事件的白名单机制可以在一定程度上缓解漏洞攻击。再者针对函数中可能存在隐含威胁的字符我们需要对其进行编码，例如用户名、密码、文件名、目录等。最后切记勿将敏感数据进行硬编码。OWASP 组织提供了 Serverless 的 Top 10 最佳实践 [46]，可供参考。

自动化检测工具是使用静态代码检测工具扫描函数的安全漏洞，业界比较主流的检测工具有 AppScan、Fortify、Burp 等。

11.2. 应用程序依赖库漏洞防护

针对依赖库漏洞的防护，最直接的方法是使用受信任的源，通常开发者可参考官方源进行下载。Serverless 平台中，云厂商提供的运行时环境相比于官方相对滞后，如果函数中包含的依赖库较少，可对库文件进行安全验证后再引入函数中；如果依赖库较多，则很容易会出现依赖库版本和运行时版本不匹配的场景，对于开发者而言，逐个去验证极其繁琐。

为了更好地解决 Serverless 函数引入第三方库漏洞的风险，业界通常采取软件组成分析（Software Composition Analysis，SCA）技术，其原理是通过对现有应用程序中使用的开源依赖项进行统计，并同时分析依赖项间的关系最后得出依赖项的开源许可证及其详细信息，详细信息包括依赖项是否存在安全漏洞、包含漏洞数量、漏洞严重程度等。最终 SCA 会根据这些前提条件判定应用程序是否可以继续运行。目前主流的 SCA 产品有 OWASP Dependency Check[47]、SonaType[48]、Snyk[49]、Bunder

[46] https://owasp.org/www-project-serverless-top-10/
[47] https://owasp.org/www-project-dependency-check/
[48] https://www.sonatype.com/
[49] https://snyk.io/

Audit[50]，其中 SonaType、Snyk、Bunder Audit 均为开源项目。

11.3. 应用程序访问控制

传统的访问控制防护方法在 Serverless 上也同样适用，我们可以从最小特权原则、隔离性这两方面出发。

最小特权原则——每个用户只能访问指定资源，粒度越细，攻击面暴露的越少。在 Serverless 中，运行单元为一个个函数，Serverless 中最小特权原则通过事先定义一组具有访问权限的角色，并赋予函数不同的角色从而实现函数层面的访问控制，以下是一个简单的 Lambda 函数部署文件代码片段：

```
service: new-service
provider:
    name: aws
functions:
    func0:
        role: myCustRole0
    func1:
        role: myCustRole1
```

上述代码片段声明了两个函数 func0 和 func1，每个函数都赋予了特定的角色，每个角色均包含对函数的具体访问权限。

函数隔离——函数间进行隔离可有效降低安全风险。Serverless 中，我们可通过部署多组函数构成一个完整的应用，由于其中每个函数都能提供大量的输入源，因此攻击者如果有权限访问某一个函数，且开发者没有对函数进行有效隔离，那么攻击者也可同时访问应用中的其它函数。由此可见，将函数作为安全隔离边界可以确保即使一组函数受到攻击也不会升级为整体应用的沦陷。现有的大多 FaaS 平台已提供了函数隔离机制，例如 AWS Lambda 采用 EC2 模型 [51] 和 Firecracker 模型 [52] 机制进行隔离。

[50] https://github.com/rubysec/bundler-audit
[51] https://docs.aws.amazon.com/lambda/latest/dg/services-ec2.html
[52] https://firecracker-microvm.github.io/

11.4. 应用程序数据安全防护

Serverless 中，我们认为应用程序的数据安全防护应当覆盖安全编码、 密钥管理、安全协议三方面。安全编码涉及敏感信息编码，密钥管理涉及密钥的存储与更换，安全协议涉及函数间数据的安全传输。

安全编码

在开发环境中，开发者常常为方便调试将一些敏感信息写在日志中，随着业务需求地不断增多，开发者容易忘记将调式信息进行删除，从而引发敏感信息泄露的风险。更为严重的是这种现象在生产环境中也频频出现，例如 python 的 oauthlib 依赖库曾被通用缺陷列表（Common Weakness Enumeration CWE）指出含有脆弱性风险 [53]，原因是其日志文件中写入了敏感信息，以下为此依赖库对应含有风险的代码：

```
if not request.grant_type == 'password':
    raise errors.UnsupportedGrantTypeError(request=request)
    log.debug('Validating username %s and password %s.', request.username, request.password)
 if not self.request_validator.validate_user(request.username,request.password, request.client, request):
raise errors.InvalidGrantError('Invalid credentials given.',request=request)
```

以上可以看出开发者将用户名密码记录在了 Debug 日志中，这是非常危险的。

为避免安全编码导致数据泄露的风险，我们应禁止将敏感信息存储至源码、日志及函数部署的配置文件中。

密钥管理

公有云厂商默认提供相应的防护方案，例如 AWS 的 KMS[54] 方案，相比于使用手动进行密钥管理，在密钥数量较多时可能会导致频繁出错，使用 KMS 可自行创建并进行加密密钥管理，操作起来更为便捷。

安全协议

[53] https://cwe.mitre.org/data/definitions/532.html

[54] https://aws.amazon.com/cn/kms/

为避免中间人攻击，函数间通信应使用 TLS 协议进行加密。

11.5. Serverless 平台账户安全防护

针对 DoW 攻击，公有云厂商可通过提供账单告警机制 [55]，如 AWS 开发者可通过在 Lambda 控制台为函数调用频度和单次调用费用设定阈值进行告警；或者也可提供资源限额的配置，即函数到达一定副本数就不再进行扩展并向开发者下发告警通知。

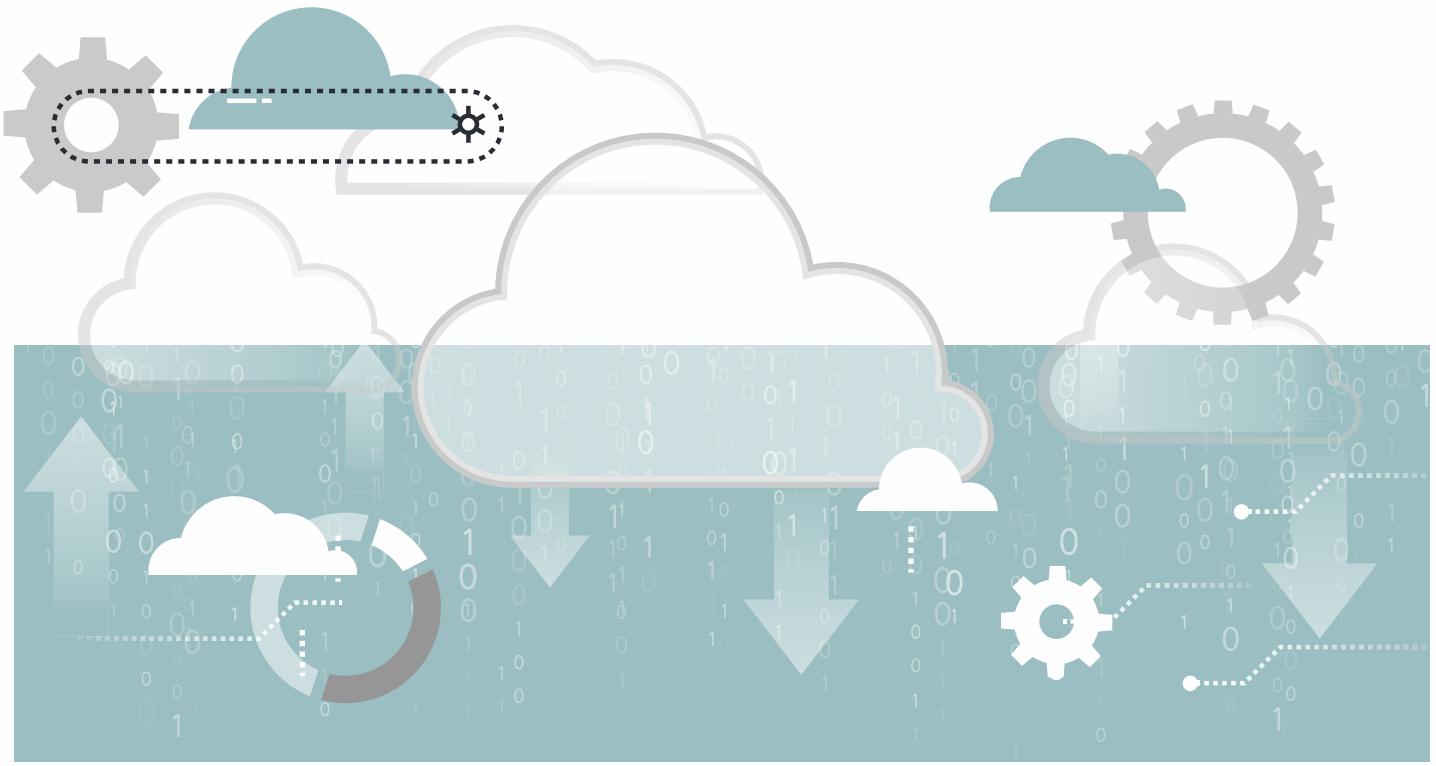
11.6. 小结

Serverless 的责任划分原则使其安全防护主要聚焦于客户端，经笔者调研，与传统的应用安全相同的是，其防护手段也适用于 Serverless 架构，不同的是 Serverless 架构层面带来的新型云原生下的应用安全场景，这就需要我们适应云计算模式的变化，不断学习总结新场景下的防护手法。

[55] https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html

12

面向新基建的云原生安全



新基建，即新型基础设施建设，目标是以新发展理念为引领、以技术创新为驱动、以信息网络为基础，面向高质量发展的需要，打造产业的升级、融合、创新的基础设施体系。主要包括 5G 基站建设、特高压、城际高速铁路和城市轨道交通、新能源汽车充电桩、大数据中心、人工智能、工业互联网七大领域。

需要说明的是，新基建中则是物理意义上的设备和系统的基础设施，而本文中提到的基础设施主要是延续云计算中的 IaaS 的“设施”概念，即 IT 基础设施。

12.1. 云原生助力信息基础设施落地

新基建包括融合基础设施、信息基础设施，以及创新基础设施 [56]。其中，信息基础设施主要指基于新一代信息技术演化生成的基础设施，比如，以 5G、物联网、工业互联网、卫星互联网为代表的通信网络基础设施，以人工智能、云计算、区块链等为代表的新技术基础设施，以数据中心、智能计算中心为代表的算力基础设施等。

在当前信息基础设施中，无疑以 5G 为代表的下一代通信网络融合了通信基础设施、新技术基础设施和算力基础设施，通过 5G 网络连接云端服务和边缘计算服务，实现了从无处不在的网络转向无处不在的计算。

一个很有意思的现象是在数年前业界讨论如何将新型网络技术应用于 5G 网络时，当时主流技术是虚拟化（Virtualization）技术，所以最流行的是基于 NFV 的编排技术和切片技术，而近年容器等技术已经成熟，5G 相关系统开始转向云原生技术。

例如在很多边缘计算平台都采用了容器技术和编排系统的技术路线。开源边缘计算平台 KubeEdge、OpenNess 和 StarlingX 就是基于容器和 Kubernetes，针对边缘计算的场景做了组件和功能改动。在国内面向 5G MEC 的边缘计算场景中，主流的商用边缘计算平台，如华为 FushionStage 就是面向容器环境的 CaaS 平台；开源的边缘计算平台，如由华为、信通院等单位发起的开源项目 EdgeGallery，不仅是容器边缘计算平台，还提供了面向第三方应用的生态系统。可见，云原生技术已经全面应用于 5G 边缘计算的 IT 基础设施。

5G 核心网的设计借鉴了大量 IT 业界成熟的技术，如切片技术就是基于 SDN 和 NFV 技术对资源和

[56] http://www.bjnews.com.cn/news/2020/04/20/718855.html

流量进行隔离和服务优化。在控制平面，核心网网元采用了 SOA、微服务架构等理念 [57]。目前，我们已经观察到有些开源和商业的网元采用了容器技术交付和部署，此外，网元之间的交互是基于 RESTful API，因而每个网元就可以被视为一个微服务。这些技术容器、编排和微服务技术的 5G 核心网网元，就适用于本文的前述威胁和风险分析，以及安全防护机制。

除了 5G 网络之外，云原生技术也同样适用于工业互联网。如 Rancher[58] 等 CaaS 厂商积极构建面向工业互联网 的 PaaS 平台，并与部署在工业现场附近的边缘计算平台结合，形成统一、基于云原生的海量数据采集、实时计算能力。

12.2. 5G 核心网安全

在运营商通信网络中，顾名思义，核心网是其核心所在，也是标准的通信技术（CT）网络。在 5G 之前的通信网络中，核心网基本上都是封闭的，运行的通信协议、网元业务与 IT 环境有很大的差异，因而网络安全厂商很难有效地对核心网网元进行全面安全防护。

然而，5G 时代的核心网发生了巨大的变化，独立架构（SA）的 5G 核心网（即 5GC）普遍使用了网络功能虚拟化（NFV）和软件定义网络（SDN）等技术。从 IT 基础设施的角度看，5G 核心网可以视为 IaaS 虚拟化系统或 CaaS 容器平台，其中 5G 核心网网元则多为虚拟机或容器的形态。特别是切片技术的引入后，控制面和数据面的网元都需要根据业务快速、按需部署和调整，因而笔者预计容器化的网元可能会越来越普遍；而且从设计上，每个 5G 核心网网元的功能独立，例如 NEF 负责网络功能开放、SMF 负责会话管理等，因而未来 5G 网元承载的服务可能在编排平台的支撑下，以微服务的模式提供 5G 控制面的业务服务。

我们研究了两个开源的 5G 核心网项目，free5gc[59] 和 open5gs[60]，接下来以这两个项目代表分析 5G 核心网安全防护机制。

[57] https://www.zte.com.cn/china/about/magazine/zte-technologies/2019/11-cn/4/2.html
[58] https://www.infoq.cn/article/FuKmSWDKmvKTX5QCxQeG
[59] https://www.free5gc.org/
[60] https://open5gs.org

在资源层方面，两个 5G 核心网项目都支持容器编排方式部署[61][62]。本文做了一些尝试，可发现这些 5G 核心网系统均可以部署，每个网元可以单个容器的方式管理。因而，原理上看，5G 核心网如果以容器和编排技术部署网元，则本文前述云原生安全防护机制，均可适用于资源层面的安全防护。

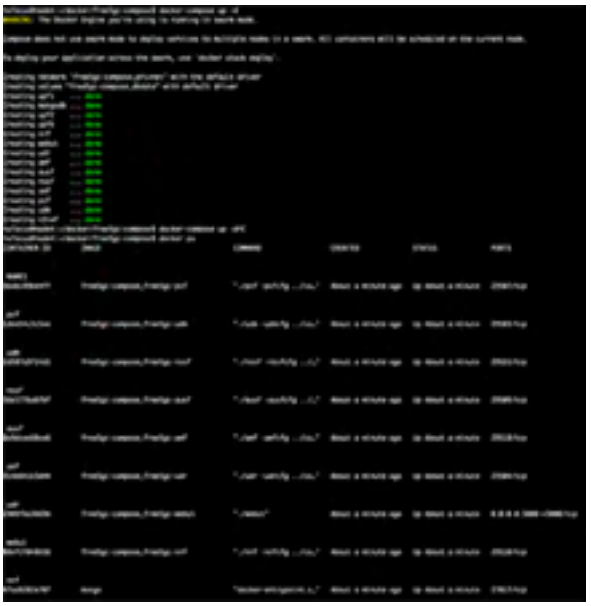


图 12.1　free5gc 的容器编排部署方式

在服务层方面，上述两个开源的 5G 核心网中，每类网元均以独立的容器部署，对外提供开放服务。如下图展示了 open5gs 的网元列表，每个网元都提供了各自微服务，图 12.3 展示了管理的前端微服务。

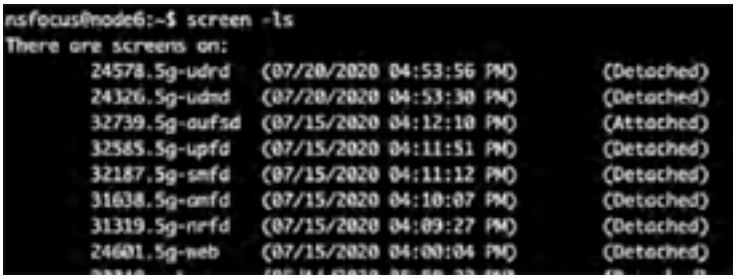


图 12.2　open5gs 的网元列表

[61] https://github.com/free5gc/free5gc#d-deploy-within-containers

[62] https://github.com/open5gs/open5gs/tree/master/docker

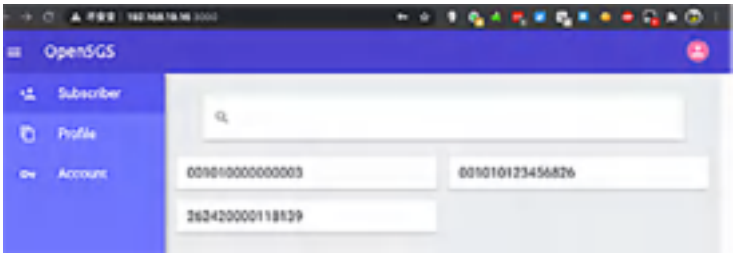


图 12.3　管理前端微服务

我们尝试在容器网络中抓包，解析后发现网元服务之间的交互均是标准的RESTful/HTTP 协议，因而，第 9 章中相关的云原生应用的业务安全检测方法同样适用于 5G 核心网的业务分析。我们可以借助 API 调用参数和序列分析对微服务业务进行基线画像，从而在运行时可以持续监控 5G 核心网网元的业务交互，及时发现异常的业务请求。

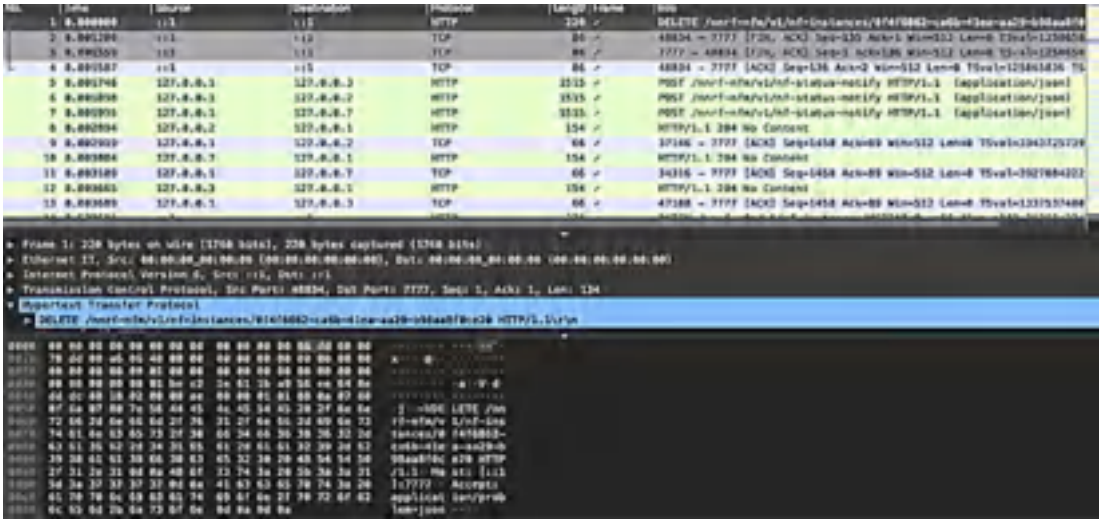


图 12.4　核心网网络抓包分析示例

12.3.　边缘计算安全

边缘计算是在靠近物或数据源头的网络边缘侧，融合网络、计算、存储、应用核心能力的分布式开放平台。随着现代工业及 5G 通信的发展，边缘计算技术将得到广泛应用。

前述 KubeEdge、OpenNESS 及 StarlingX 在内的主流开源边缘计算平台均采用了云原生技术路线，

基于容器和 Kubernetes 为各种云边需求提供解决方案。一方面，这体现了云原生技术广泛的应用场景，边缘计算具有云原生灵活、高效、稳定的特性；另一方面，这意味着云原生面临的的风险也会存在于边缘计算环境，边缘计算自身特点也将带来新的安全挑战。

综合来看，边缘计算面临的安全挑战主要有以下几点：

资源受限：与传统云计算环境不同的是，边缘计算环境下算力、存储资源通常较为有限，传统安全防护软硬件的部署可能会受到限制。

云边平台自身安全性：边缘与云端共同作为云计算环境的组成部分，各自平台系统自身安全性是整个云计算环境安全的基础，传统主机、网络安全威胁仍然存在。

边缘应用的时间约束：边缘应用自身具有复杂多样性，加上容器技术的应用，边缘计算应用将会越来越体现出高频次、短周期的特点，攻防进而也会发生变化。

数据隐私与保护：在边缘计算概念中，边缘不再单单是一个个传感器，而是具备一定计算、存储能力的分布式节点，确保边缘计算环境下的数据隐私得到合理应用和保护变得愈加重要。

安全体系云边融合：边缘计算绝不仅仅是“边缘的计算”，在业务上，边缘与云端并非割裂，而是协同、融合的，因此，与业务相适应的安全体系也必须做到协调联动、云边融合。

在主流开源的边缘计算平台中，KubeEdge、OpenNESS、StarlingX 的基本架构如图 12.5 所示：

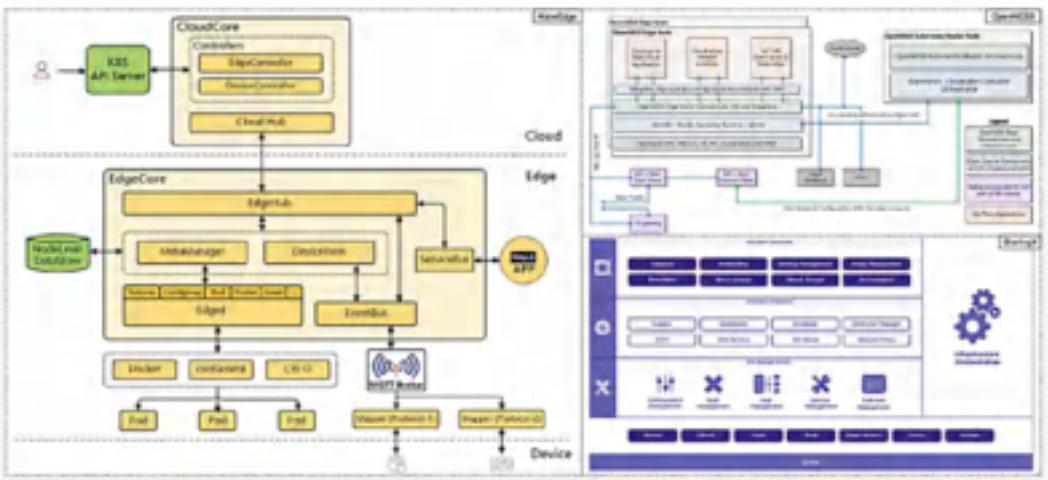


图 12.5　KubeEdge、OpenNESS 和 StarlingX 的基本架构

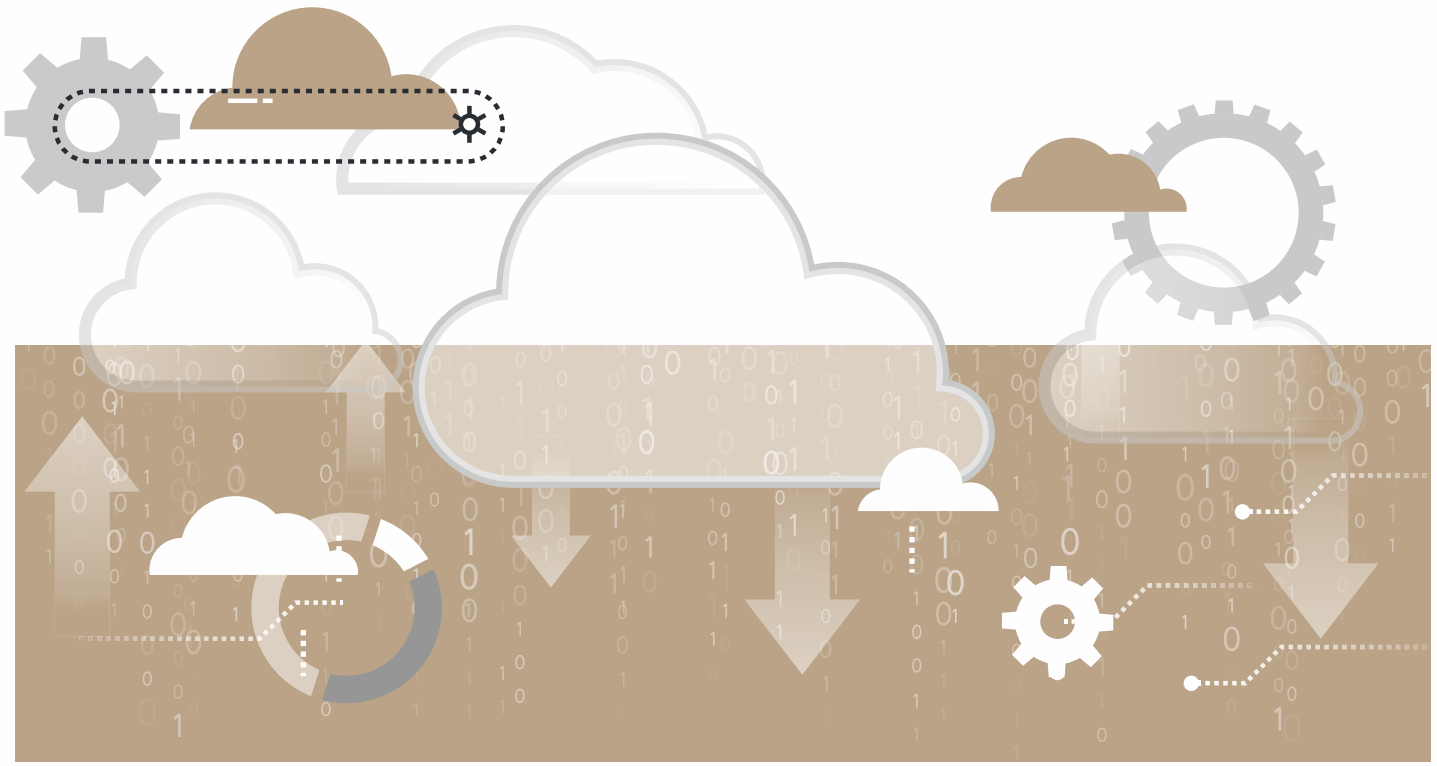
其中，KubeEdge 基于 Kubernetes 开发而成，由位于云端的 CloudCore 和位于边缘的 EdgeCore 两部分组成，对原 Kubelet 进行了精简，使其轻量化，充分考虑边缘节点资源受限的情况；OpenNESS 则直接使用了 Kubernetes 作为它的编排管理系统，它由 Kubernetes 控制平面节点和边缘节点组成； StarlingX 同时基于 OpenStack 和 Kubernetes，借助 OpenStack 对虚拟机资源进行管理，借助 Kubernetes 对容器资源进行管理。三者都与 Kubernetes 存在联系。

对于基于 Kubernetes 的平台，一个很自然的思路是采用云原生的方式去做安全防护。我们持续探索边缘计算的安全防护。基于容器防护思路和技术积累，我们已经实现上述三个开源平台的安全防护机制验证（图 12.6），可为边缘计算平台提供符合需求的安全服务。这再一次证明， 以云原生安全的方式为边缘计算提供安全保障是可行的。

图 12.6　针对不同边缘计算平台的安全实践

13

总结



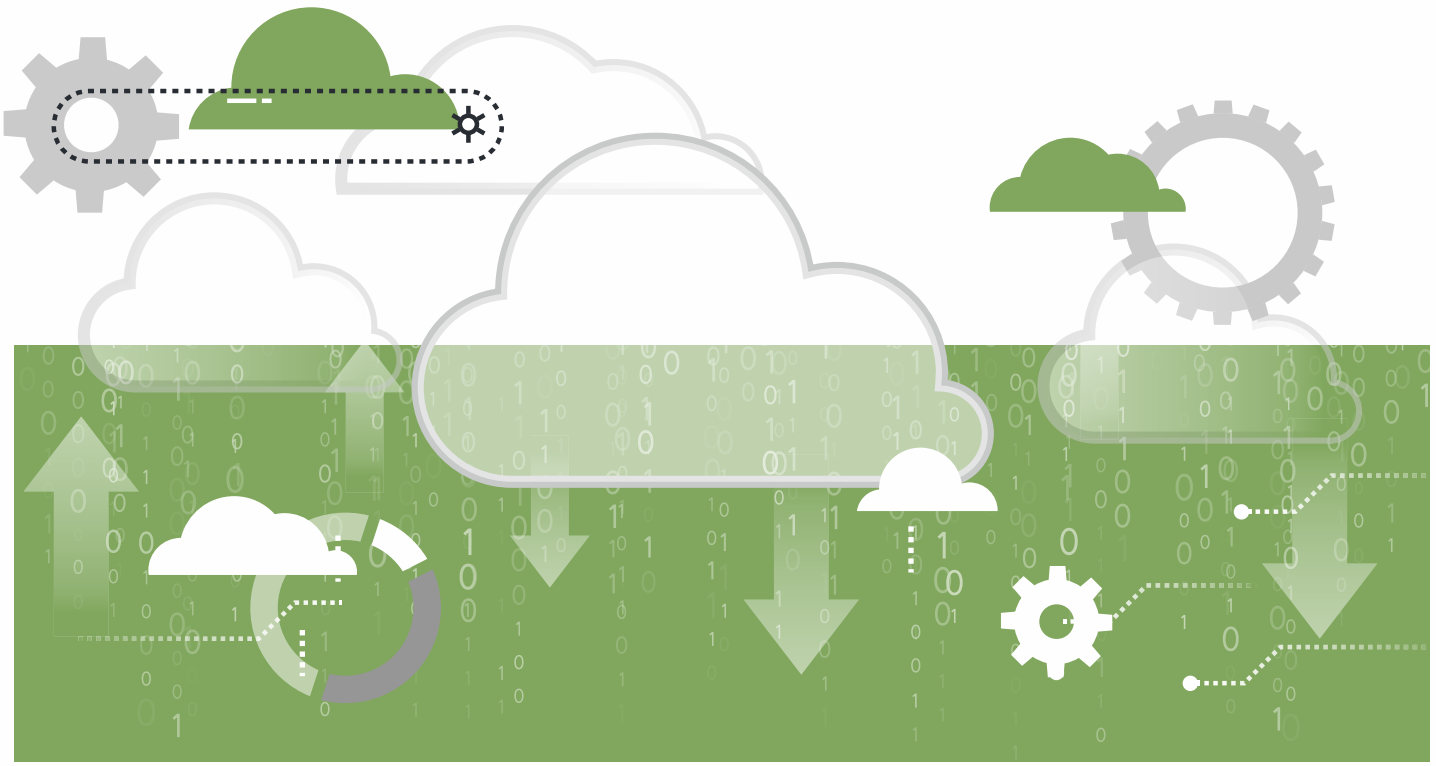
总结

云原生是云计算的下半场，云原生安全将是未来几年云安全的主要发展方向。一方面，云原生的内生安全需要深入研究，本文分析了云原生环境中的安全风险和威胁，进而提出并阐述了云原生安全防护体系；另一方面，随着新基建的快速推进，云原生技术与信息基础设施的融合也成为一个明显趋势，服务提供商在设计、部署和运营这些基础设施和应用系统时应充分考虑与云化技术相关的风险、威胁和安全防护手段。

虽然云原生技术现在还处于早期发展的阶段，但其必然会与各种信息系统、各类计算场景融合，弹性敏捷、按需编排都会成为系统原生具备的能力。那么安全厂商和服务商应充分研究云原生相关的安全技术，提供新型云化信息基础设施的防护、检测和响应能力，并将云原生技术赋能于这些安全产品、平台和解决方案，最终提供云原生内生的安全能力。

14

参考文献



参考文献

[1] https://www.cncf.io/about/faq

[2] https://www.nsfocus.com.cn/html/2018/92_1112/70.html

[3] https://aws.amazon.com/cn/products/storage/object-storage-for-cloud-native-applications

[4] https://businessinsights.bitdefender.com/worst-amazon-breaches

[5] https://github.com/nagwww/s3-leaks

[6] https://web.archive.org/web/20180222103919/https://blog.redlock.io/cryptojacking-tesla

[7] https://azure.microsoft.com/en-us/blog/detect-largescale-cryptocurrency-mining-attack-against-kubernetes-clusters

[8] https://www.microsoft.com/security/blog/2020/06/10/misconfigured-kubeflow-workloads-are-a-security-risk

[9] https://unit42.paloaltonetworks.com/graboid-first-ever-cryptojacking-worm-found-in-images-on-docker-hub

[10] https://nvd.nist.gov/vuln/detail/CVE-2019-5021

[11] https://mackeeper.com/blog/post/cryptojacking-invades-cloud-how-modern-containerization-trend-is-exploited-by-attackers/

[12] https://github.com/docker/hub-feedback/issues/1121#issuecomment-326664651

[13] https://www.docker.com/blog/docker-can-now-run-within-docker

[14] https://mp.weixin.qq.com/s/UZ7VdGSUGSvoo-6GVI53qg

[15] https://i.blackhat.com/USA-20/Thursday/us-20-Avrahami-Escaping-Virtualized-Containers.pdf

[16] https://kubernetes.io/docs/concepts/policy/limit-range/

[17] https://owasp.org/www-project-top-ten

[18] https://nvd.nist.gov/vuln/detail/cve-2018-1002105

[19] https://nvd.nist.gov/vuln/detail/cve-2017-1002101

[20] https://en.wikipedia.org/wiki/Symlink_race

[21] https://istio.io/latest/docs/ops/deployment/architecture

[22] https://martinfowler.com/bliki/Serverless.html

[23] https://www.owasp.org/index.php/OWASP_Serverless_Top_10_Project

[24] https://snyk.io/opensourcesecurity-2019

[25] https://www.owasp.org/index.php/OWASP_Serverless_Top_10_Project

[26] https://www.npmjs.com/advisories

[27] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=java

[28] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=python

[29] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11756

[30] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7560

[31] https://copyconstruct.medium.com/monitoring-and-observability-8417d1952e1c

[32] https://landing.google.com/sre/

[33] https://mp.weixin.qq.com/s?__biz=MzAxNTA1OTY4OQ==&mid=2457362109&idx=1&sn=b90b15ded62d1f6f52545d025ef80014&scene=21#wechat_redirect

[34] https://csrc.nist.gov/publications/detail/sp/800-207/final

[35] https://istio.io/latest/docs/concepts/security/arch-sec.svg

[36] https://istio.io/latest/docs/concepts/security/authz.svg

[37] https://cilium.readthedocs.io/en/stable/concepts/overview

[38] https://www.cloudvector.com/owaspwhitepaper/#:~:text=Gartner%20predicts%20that%20%E2%80%9Cby%202022,breaches%20for%20enterprise%20web%20applications%E2%80%9D.&text=Risks%20due%20to%20uninspected%20API,to%20uncontrolled%20third%20party%20APIs

[39] https://owasp.org/www-project-api-security/

[40] https://www.nsfocus.com.cn/html/2019/209_1009/66.html

[41] https://www.nsfocus.com.cn/html/2019/206_0911/8.html

[42] https://www.signalsciences.com/

[43] https://www.fortinet.com/products/web-application-firewall/fortiweb

[44] https://wallarm.com/

[45] https://www.radware.com/

[46] https://owasp.org/www-project-serverless-top-10

[47] https://owasp.org/www-project-dependency-check/

[48] https://www.sonatype.com/

[49] https://snyk.io/

[50] https://github.com/rubysec/bundler-audit

[51] https://docs.aws.amazon.com/lambda/latest/dg/services-ec2.html

[52] https://firecracker-microvm.github.io/

参考文献

[53] https://cwe.mitre.org/data/definitions/532.html

[54] https://aws.amazon.com/cn/kms/

[55] https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html

[56] http://www.bjnews.com.cn/news/2020/04/20/718855.html

[57] https://www.zte.com.cn/china/about/magazine/zte-technologies/2019/11-cn/4/2.html

[58] https://www.infoq.cn/article/FuKmSWDKmvKTX5QCxQeG

[59] https://www.free5gc.org/

[60] https://open5gs.org

[61] https://github.com/free5gc/free5gc#d-deploy-within-containers

[62] https://github.com/open5gs/open5gs/tree/master/docker

绿盟威胁情报中心

绿盟威胁情报中心（NSFOCUS Threat Intelligence center, NTI）是绿盟科技为落实智慧安全 2.0 战略，促进网络空间安全生态建设和威胁情报应用，增强客户攻防对抗能力而组建的专业性安全研究组织。其依托公司专业的安全团队和强大的安全研究能力，对全球网络安全威胁和态势进行持续观察和分析，以威胁情报的生产、运营、应用等能力及关键技术作为核心研究内容，推出了绿盟威胁情报平台以及一系列集成威胁情报的新一代安全产品，为用户提供可操作的情报数据、专业的情报服务和高效的威胁防护能力，帮助用户更好地了解和应对各类网络威胁。

网址：https://nti.nsfocus.com/

星云实验室

专注于云计算安全、解决方案研究与虚拟化网络安全问题研究。 基于 IaaS 环境的安全防护，利用 SDN/NFV 等新技术和新理念，提出了软件定义安全的云安全防护体系。承担并完成多个国家、省、市以及行业重点单位创新研究课题，已成功孵化落地绿盟科技云安全解决方案。

绿盟创新中心

绿盟科技创新中心是绿盟科技的前沿技术研究部门。关注云安全、容器安全、威胁情报、数据驱动安全、物联网安全和区块链等领域。作为“中关村科技园区海淀园博士后工作站分站”的重要培养单位之一，与清华大学进行博士后联合培养，科研成果已涵盖各类国家课题项目、国家专利、国家标准、高水平学术论文、出版专业书籍等。我们持续探索信息安全领域的前沿学术方向，从实践出发，结合公司资源和先进技术，实现概念级的原型系统，进而交付产品线孵化产品并创造巨大的经济价值。

中国移动云能力中心

中国移动云能力中心是中国移动通信有限公司的全资子公司，公司定位为中国移动云设施构建者、云服务提供者、云生态汇聚者。公司以移动云运营为中心，产品和服务在政务、金融、制造、交通、医疗等行业得到广泛应用。

NSFOCUS

THE EXPERT
BEHIND GIANTS
巨人背后的专家

多年以来，绿盟科技致力于安全攻防的研究，
为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供
具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。
在这些巨人的背后，他们是备受信赖的专家。

www.nsfocus.com

欢迎关注
绿盟科技官方微信