微服务API安全方案

（可持续、可扩展的访问控制）

Securing Microservice APIs

（Sustainable and Scalable Access Control）

CA Technologies

Matt McLarty, Rob Wilson, and Scott Morrison

February 2018: First Edition

目录

[前言 - Preface 4](#_Toc79595976)

[Chapter1：微服务架构——Microservice Architecture 5](#_Toc79595977)

[1.1 微服务API概览—The Microservice API Landscape 6](#_Toc79595978)

[1.2 微服务API访问控制—API Access Control for Microservices 7](#_Toc79595979)

[1.3 微服务架构的质量—Microservice Architecture Qualities 8](#_Toc79595980)

[Chapter2：微服务访问控制——Access Control for Microservices 10](#_Toc79595981)

[2.1 构建信任—Establishing Trust 11](#_Toc79595982)

[2.2 网络级控制—Network-Level Controls 12](#_Toc79595983)

[2.2.1 本地主机隔离—Localhost Isolation 12](#_Toc79595984)

[2.2.2 网络分段—Network Segmentation 13](#_Toc79595985)

[2.2.3 基线—The Bottom Line for Microservices 15](#_Toc79595986)

[2.3 应用级控制—Application-Level Controls 15](#_Toc79595987)

[2.3.1 传统Web令牌的问题—The Problem with Traditional Web Tokens 15](#_Toc79595988)

[2.3.2 用于API的现代令牌—Modern Tokens For APIs 17](#_Toc79595989)

[2.3.3 基线—The Bottom Line for Microservices 24](#_Toc79595990)

[2.4 基础设施—Infrastructure 24](#_Toc79595991)

[2.4.1 代理/网关—Proxy/Gateway 25](#_Toc79595992)

[2.4.2 网络覆盖技术—Network Overlays 26](#_Toc79595993)

[2.5 新兴技术—Emerging Approaches 28](#_Toc79595994)

[2.5.1 服务网格—Service Mesh 29](#_Toc79595995)

[2.5.2 无服务器计算—Serverless Computing 30](#_Toc79595996)

[Chapter3：一个微服务API安全的通用方法——A General Approach to Microservice API Security 31](#_Toc79595997)

[3.1 API安全解决方案中的通用模式—Common Patterns in Microservice API Security Solutions 31](#_Toc79595998)

[3.2 DHARMA模型—Domain Hierarchy Access Regulation for Microservice Architecture (DHARMA) 32](#_Toc79595999)

[3.3 DHARMA模型设计方法—DHARMA Design Methodology 35](#_Toc79596000)

[3.4 与平台无关的DHARMA 模型实现—A Platform-Independent DHARMA Implementation 36](#_Toc79596001)

[3.4.1 域的层次—Domain Hierarchy 36](#_Toc79596002)

[3.4.2 信任和访问机制—Trust and Access Mechanisms 37](#_Toc79596003)

[3.4.3 实现过程中的考虑因素—Implementation Considerations 38](#_Toc79596004)

[3.4.4 DHARMA实现总结—Summary of the Platform-Independent DHARMA Implementation 41](#_Toc79596005)

[3.5 开发人员的经验—Developer Experience in DHARMA 43](#_Toc79596006)

[3.5.1 对服务/API启用访问控制—Enabling Access Control for a Service/API 44](#_Toc79596007)

[3.5.2 发布和发现API访问控制策略—Publishing and Discovering API Access Control Policies 44](#_Toc79596008)

[3.5.3 访问控制策略的变更—Access Control Policy Change Management 45](#_Toc79596009)

[Chapter4：结论：微服务安全边界—Conclusion: The Microservice API Security Frontier 46](#_Toc79596010)

[标准化微服务语言—Standardizing the Language of Microservices 46](#_Toc79596011)

[应用DHARMA模型—Applying DHARMA 46](#_Toc79596012)

[扩展DHAEMA模型—Extending DHARMA 47](#_Toc79596013)

# 前言 - Preface

在微服务体系结构中，有许多技术用于控制对WEB API的访问，包括网络控制、加密方法和基于平台的功能。本文提出了一种API访问控制模型，该模型可以在任何一个平台上实现，也可以跨多个平台实现，以便在微服务网络上提供整体的安全性。

There are a number of techniques for controlling access to web APIs in a microservice architecture, including network controls, cryptographic methods, and platform-based capabilities. This paper proposes an API access control model that can be implemented on any one platform or across multiple platforms in order to provide cohesive security over a network of microservices.

**谁应该阅读这份报告？**

本报告适用于参与构建和维护微服务系统的任何人，特别是负责整个系统安全的人。这包括许多可能的角色:架构师、产品经理、开发负责人、平台团队和运营经理。

**Who Should Read This Report**

This report is intended for anyone involved in building and maintaining a system of microservices, especially those responsible for the security of the overall system. This encompasses many possible roles: architects, product owners, development leaders, platform teams, and operational managers.

**本报告有哪些内容**

本报告由四部分组成。

1. 微服务综述，为安全模型设置背景

2. 对适用于微服务API现有安全技术和解决方案的调研

3. 一个推荐的微服务API安全模型

4. 总结，以及对微服务API安全未来发展方向的推测

**What’s in This Report**

This report consists of four sections:

1. An overview of the microservices landscape, to set the context for the security model

2. A survey of available security technologies and solutions that apply to microservice APIs

3. A proposed model for securing microservice APIs

4. A conclusion that includes speculation on the future direction of microservice API security

**本报告中没有的内容**

本报告明确聚焦于基于HTTP用于与微服务之间的通信的API，既不包括非HTTP传输协议的安全方法，也不包括一般容器的安全方法。

**What’s Not in This Report**

This report is explicity focused on HTTP-based APIs for communication with and between microservices. Neither security approaches for non-HTTP transport protocols nor security approaches for containers in general are included.

# Chapter1：微服务架构——Microservice Architecture

2014年初，詹姆斯·刘易斯（James Lewis）和马丁·福勒（Martin Fowler）发表了一篇博文，描述了一种新型的软件架构风格，该体系结构由小型互连组件组装成分布式应用程序，此后，“微服务”一词开始流行。微服务体系结构中的单个微服务通常显示以下特征：

* **面向服务**

单个微服务通常实现单一的功能性职责，并可能被系统的任何“层”（“layer” or “tier”）的其他软件组件使用。

* **独立的可部署性和可管理性**

一个单独的微服务应该能够独立部署、管理和扩展，而不需要与系统中的其他组件协调。

* **短暂性和弹性**

单个微服务实例的寿命通常很短，为了满足系统的动态性能需求，一个微服务的多个实例经常运行后关闭。

The term “microservices” gained popularity following a blog post from James Lewis and Martin Fowler published in early 2014 in which they described a new style of software architecture consisting of small, interconnected components assembled to form distributed applications. Individual microservices within a microservice architecture generally display the following characteristics:

**Service orientation**

An individual microservice typically implements a single functional responsibility and may be consumed by other software components at any “layer” or “tier” of the system.

**Independent deployability and manageability**

An individual microservice should be able to be deployed, managed, and scaled on its own without the need to coordinate with other components in the system.

**Ephemerality and elasticity**

Individual microservice instances are frequently short-lived, and multiple instances of a microservice are often run and then shut down in order to meet the dynamic performance needs of the system.

除了这些特性之外，微服务通常还使用以下标准技术：

* **Web API通信**

微服务经常通过基于HTTP的Web API发布其业务功能，这些API使用JSON或其他相关媒体类型进行编码。

* **基于容器的部署**

微服务通常使用Linux容器——通常是Docker容器——作为其部署单元，允许在一系列框架和平台中从研发平滑过渡到运营。

In addition to these characteristics, microservices often use the following standard technologies:

**Web API communication**

Microservices often publish their business functions through HTTP-based web APIs encoded using JSON or other related media types.

**Container-based deployments**

Microservices often use Linux containers—frequently Docker containers—as their unit of deployment, allowing for a smooth transition from development to operations in a range of frameworks and platforms.

总之，这些微服务特性和通用技术必须纳入任何微服务API访问控制的解决方案中。

Collectively, these microservice characteristics and common technologies must be factored into any solution for microservice API access control.

## 微服务API概览—The Microservice API Landscape

为了定义微服务API安全性的通用模型，需要一些关键概念。我们从服务（也称为微服务）开始，这是一个逻辑组件，通过接口向服务消费者提供功能。一个服务实例通过一个或多个运行时组件实现，这些组件通常是微服务架构中的一组容器。服务接口通常是一个WEB API，一个可通过HTTP(s)访问的编程接口。**服务的API是通过API端点来访问的，API端点是运行时环境中的一个可网络寻址的位置。一个服务的API可能有多个端点。**

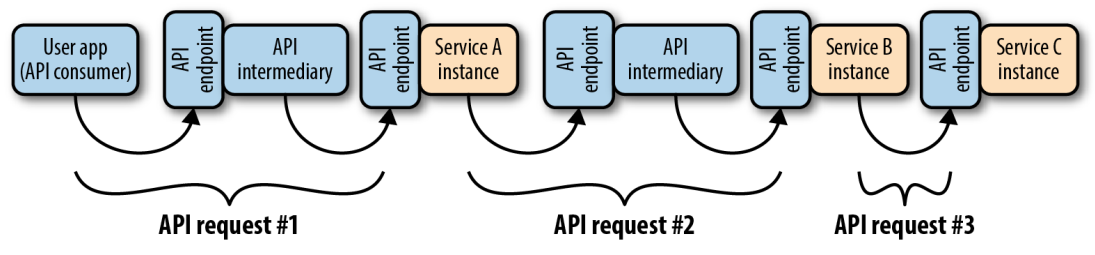
Some key concepts are needed in order to define a universal model for microservice API security. We start with the service (aka microservice), a logical component that provides functionality to service consumers through an interface. A service instance is implemented through one or more runtime components, often a set of containers in a microservice architecture. The service interface is often a web API, a programmatic interface accessible via HTTP(s). A service’s API is accessed through an API endpoint, a network-addressable location within the runtime environment.1 A service’s API may have more than one endpoint.

API请求，发送给API端点（API endpoint）【API endpoint注释：通常由Consul, Eureka，etcd，*zookeeper*这样的服务注册框架以列表形式给出】，该端点触发服务的执行；API响应是服务执行结果。发送API请求的组件扮演API消费者（API consumer）的角色，而接收API请求并向消费者发送API响应的服务扮演着API提供者（API provider）的角色。一个服务可能同时扮演API消费者和API提供者的角色，具体取决于消息的上下文。这两个角色均可以由服务以外的组件扮演。API中间人（API intermediary）是位于从API消费者到API提供者之前的API请求路径上的一个组件。API网关（API gateway）和服务代理（Service proxy）是常见的API中间人。API端点（API endpoint）可以在API中间人（API intermediary）上实现。

An API request is a message sent to an API endpoint that triggers the service’s execution, and an API response is a message sent in return to communicate the result of the service’s execution. A component that sends an API request takes the role of API consumer, while the service that receives the API request and sends the API response back to the consumer takes the role of API provider. A service may play the role of both API consumer and API provider, depending on the message context. Both roles may also be played by components other than services. An API intermediary is a component that sits in the API request path from API consumer to API provider. API gateways and service proxies are common API intermediaries. An API endpoint may be implemented on an API intermediary.

图1-1展示了一个微服务架构中各种概念协同工作的示例：

Figure 1-1 shows an example of these concepts working together in a microservice architecture:



## 微服务API访问控制—API Access Control for Microservices

交付速度通常是推动组织转向微服务体系结构的因素，安全性是次要考虑因素。本书解决了微服务体系结构中web API的访问控制问题。“IAAA”访问控制框架——身份标识、认证、授权和审计——为在微服务场景中描述WEB API访问控制提供了有用的基础。

Speed of delivery has typically been the motivating factor for organizations moving to a microservice architecture, security being a secondary consideration. This book addresses access control for web APIs within a microservice architecture. The “IAAA” access control framework—identification, authentication, authorization, and accountability (alternatively accounting, or auditing)—provides a useful basis for describing web API access control in the context of microservices.

IAAA访问控制框架：identification, authentication, authorization, accountability (alternatively accounting, or auditing)

**身份标识**

消息可能由终端用户的活动或自动化事件触发，并可能通过几个中间人的分发和转换。服务消费者和中间人必须能够发送包含多个身份标识信息（附带详细描述这些身份标识的可选属性）的API请求消息，并且也必须能够接收包含多个身份标识信息（附带属性）的API请求。

Identification

Messages may be triggered by end user activity or automated events and may be distributed and transformed through several intermediaries. Service consumers and intermediaries must be able to send API request messages that include multiple identities along with optional attributes that detail those identities, and they must be able to accept API requests that include multiple identities and their attributes.

**认证**

可以通过包含的凭据、断言的声明（例如，令牌）、信任关系或这些方法的组合对API请求进行身份验证。服务必须能够自行执行身份认证，或者将身份认证委托给受信任的组件来执行。

Authentication

API requests may be authenticated through included credentials, asserted claims (e.g., a token), trust relationships, or a combination of these methods. Services must be able to either perform the authentication themselves or delegate authentication to a trusted component.

**授权**

任何验证身份标识的应用程序组件——服务或中间人，都可以根据身份标识、其属性和请求上下文的组合做出授权决策。在一个分布式微服务架构中，单个请求在从组件到组件的传递过程中，可能要经过多个授权决策。

Authorization

Any application component—service or intermediary—that authenticates an identity may make an authorization decision based on the combination of the identity, its attributes, and the request context. In a distributed microservice architecture, a single request may go through multiple authorization decisions as it is passed from component to component.

**审计**

针对受API请求影响的系统活动进行审计是很重要的，这可以对有意或无意的系统泄露提供取证细节。API消息的审计可以在请求路径或者响应路径的任何点实施。鉴于所涉及的身份、属性和处理组件的范围可能很广，尽可能多地捕获消息的上下文是很有价值的。

Accountability

It is important to audit system activity affected by API requests in order to provide forensic details for intentional or unintended system breaches. Accounting for an API message can happen at any point in the request’s or response’s path. It is valuable to capture as much of the message’s context as possible, given the potentially wide range of identities, attributes, and processing components involved.

## 微服务架构的质量—Microservice Architecture Qualities

除了对微服务API访问控制具体的功能要求外，非功能要求也很重要。功能性需求描述了解决方案需要做什么，而非功能性需求定义了解决方案应该如何实现和操作。这在微服务体系结构中尤其重要，因为有许多质量问题将决定一个解决方案是否适合采用了微服务的组织。

In addition to the specific functional requirements for microservice API access control, it is important to note the nonfunctional requirements. Whereas the functional requirements describe what the solution needs to do, the nonfunctional requirements define how the solution should be implemented and operated. This is especially important in a microservice architecture, since there are a number of qualities that will determine whether a solution will be amenable to organizations adopting microservices.

**可管理性/可操作性**

微服务架构的特点是所有功能的高度自动化。为了使访问控制解决方案在微服务架构中可行，它必须提供机器可访问接口以实现管理自动化。

Manageability/Operability

Microservice architectures typically feature a high degree of automation for all functions. In order for an access control solution to be viable in a microservice architecture, it must expose machine accessible interfaces for management automation.

**性能**

由于微服务体系结构的分布式特性，每个组件的处理延迟都有可能降低整个系统的性能。因此，微服务体系结构中的访问控制解决方案应尽可能避免增加延迟。

Performance

Due to the distributed nature of microservice architectures, the processing latency of each component has the potential to degrade the performance of the overall system. As such, an access control solution within a microservice architecture should avoid adding latency as much as possible.

**易用性**

微服务架构的流行是由开发者推动的。在微服务运动中受到欢迎的工具通常具有很强的易用性，其特点是减少了开发者体验中的摩擦。因此，对于微服务API访问控制解决方案来说，不妨碍开发者的任务是很重要的。

结合功能访问控制框架及其最佳特性，我们现在可以评估当前微服务架构中API访问控制的各种方法。

Usability

The rise in popularity of microservice architecture has been driven by developers. Tools that have gained popularity in the microservices movement have usually featured strong usability, marked by reduced friction in the developer experience. Therefore, it is important for a microservice API access control solution not to impede a developer’s tasks.

With the combination of the functional access control framework and its optimal characteristics, we may now evaluate the variety of current approaches to API access control in a microservice architecture.

# Chapter2：微服务访问控制——Access Control for Microservices

API使应用集成变得简单。一个网络浏览器或一个curl命令就是你搞出一个端点的全部需要。没有复杂的库，没有代码生成的SDK，甚至没有编译，只有WEB的基本架构和基础设施。这种消除障碍和摩擦的做法，比任何其他原因都要重要，这就是为什么开发人员喜欢API。

APIs make application integration simple. A web browser or a curl command is all you should need to try out an endpoint. No complex libraries, no code-generated SDKs, not even a compile—just the basic architecture and infrastructure of the web. This elimination of barriers and friction, more than any other reason, is why developers love APIs.

但是，您可能会将WEB模型看得太高，尤其是在安全性方面。API在信任和身份方面带来了一些复杂的挑战，需要比传统web提供的更复杂的方法以应对这一挑战。**OAuth和OpenID Connect等协议、服务节流等实践都是对API安全性这一独特挑战的响应。**

But you can take the web model too far, and this is especially true for security. APIs bring some complex challenges in trust and identity that demand a more sophisticated approach than the conventional web has to offer. Protocols like OAuth and OpenID Connect, practices such as service throttling—these were all responses to the unique challenge of API security.

微服务增加了另一层复杂性，使其具有独特的安全需求。容器、短暂的实例、运行时服务发现、对跨应用程序重复的关注——这些因素共同作用，使微服务的安全变得困难。**到目前为止，很少有指南描述如何保障现代微服务的安全。**

Microservices add another layer of complexity with unique security demands. Containers, ephemeral instances, runtime service discovery, the focus on re-use across many apps—these factors conspire to make microservices security hard. Until now, there have been few guides describing how to secure modern microservices.

本章的目的是帮助架构师和开发人员更好地了解他们在哪里投入信任。本章不涉及如何实现每项技术的细节，因为这超出了本书的范围，您最好使用最新的材料来实现。相反，它说明了一项技术存在的原因，以便你在自己的微服务架构中正确应用它。

The goal of this chapter is to help architects and developers better understand where they are investing their trust. This chapter does not go into the details of how to setup each technology, as this is beyond the scope of this book and better dealt with using the most up-to-date materials for your implementation. Instead, it illustrates why a technology exists so that you apply it correctly in your own microservices architecture.

## 构建信任—Establishing Trust

所有的安全都建立在信任的基础上。但是信任没有有效的措施，只有谨慎的努力才能对信任建立信心。我们对分布式系统的信任是我们为降低风险而做出的许多决策的而得来的。

All security is based on trust. But trust has no effective measure, only confidence that grows with careful diligence. Our trust in a distributed system is an accumulation of many decisions we make to mitigate risk.

做出这些决定是很重要的，因为只有这时，我们才能开始梳理隐藏在设计中的隐式信任。

It is important to call out these decisions, because only then can we begin to tease out the implicit trust that hides in our design. Too many modern platforms make security opaque. This might make them easy to use, but it masks assumptions and limitations. Good security architecture is transparent about where it invests the trust.

考虑一个简单的静态网站。从表面上看，它应该很容易做到安全。页面是开放的，每个人都可以阅读，因此不需要身份验证或用户管理。它只支持简单的HTTP GET，因此攻击者似乎没有机会利用它进行攻击。

Consider a simple, static website. On the surface, it should be easy to secure. The pages are open for everyone to read, so there is no need for authentication or user management. It only supports simple HTTP GETs, so it would appear there is little opportunity for an attacker to exploit.

但深入研究，我们会发现隐含的信任。我们相信我们的提供商能够处理DDoS缓解问题。我们希望他们有良好的物理安全。我们假设他们对CMS平台进行了加固，并及时了解最新的攻击向量，这些攻击向量可能针对我们简单HTML页面下面的基础设施。

But dig down, and we find the implicit trust. We trust our provider to handle DDoS mitigation. We hope they have decent physical security. We assume they harden their CMS platforms and keep up on the latest attack vectors that might target the infrastructure below our simple HTML pages.

在另一个极端，想象一个安全的政府部门的计算机设备。与互联网断开，它的系统被安置在没有窗户的设防建筑内的一个法拉第笼内。甚至连电源都是专用的。然而，尽管如此注重细节，安全性只能达到与使用它的内部人员的审查措施的安全性一样好的程度。

At the other extreme, imagine a secure government computing facility. Disconnected from the internet, its systems reside in Faraday cages inside a fortified building without windows. Even the power supply is private. Yet despite this attention to detail, the security is only as good as the vetting of the insiders using it.

关键是，信任是一种妥协，我们需要将一系列风险保持在可接受的水平。计算机安全没有绝对性；只有信任和接受风险。安全体系结构是一个调优练习，尝试在许多利益冲突之间优化信任。

The point is, trust is about compromise, and we need to be comfortable residing on a spectrum of risk. There are no absolutes in computer security; there is only trust and acceptance of risk. Security architecture is a tuning exercise, trying to optimize trust against many competing interests.

下面的章节涵盖了用于建立微服务安全访问的基本构件。有些体系结构单独使用这些技术，但它们也可以组合使用，以满足开发人员、操作人员和安全专业人员的相互竞争的需求。

The following sections cover the basic building blocks used to build a secure access to microservices. These approaches all create trust boundaries. Some architectures use these techniques in isolation, but they may also be combined to meet the competing needs of developers, operators, and security professionals.

## 网络级控制—Network-Level Controls

限制应用程序访问的最简单方法是控制对网络的访问。这是一个有吸引力的解决方案：通过与应用程序的解耦，我们使开发者的工作变得更加容易。但这是一种比较笨的方法，很难大规模维持，如果失陷，也会出现灾难性的失败。尽管如此，它在安全的微服务体系结构中仍有一席之地。

The simplest way to restrict access to an application is to control access to the network. It is an attractive solution: by decoupling from the application, we make the developer’s job much easier. But this is a blunt instrument that is difficult to maintain at scale and subject to catastrophic failure if compromised. Nevertheless, it has its place in a secure microservices architecture.

### 本地主机隔离—Localhost Isolation

本地主机隔离是一种常见的开发人员模式。我们已经在自己的开发机器上构建并测试了应用程序，相信防火墙可以保护我们免受恶意网络连接的伤害。它很简单，正因为如此，它是一个很有用的模型，可以用来说明任何网络分割方案的优缺点。因为它在容器部署中的广泛使用，特别是在诸如边车这样的常见模式中，它今天也非常重要。

Localhost isolation is a common developer pattern. We’ve all built and tested applications on our development machine, confident that the firewall is protecting us from malicious network connections. It is simple, and because of that it’s a useful model to illustrate the pros and cons of any network segmentation scheme. It is also very relevant today because of its widespread use in container deployments, especially in common patterns like the sidecar.

本地主机隔离简化了应用程序，因为应用可以信任所有发送者。它允许我们将服务与特定的端口相关联，类似于传统的TCP和UDP安全模型，将知名端口绑定到特定的应用程序。

Localhost isolation simplifies applications because they can trust all senders. It allows us to associate services with specific ports, approximating the traditional TCP and UDP security model binding wellknown ports to specific applications.

但是这个模型不能识别客户端应用程序(源端口是临时的，由网络堆栈分配)，并且该模型假定操作系统上的所有进程都是同样的可信。它没有告诉我们与客户端实体关联的用户的任何信息——因此我们需要在堆栈上向上移动。

But this model does nothing to identify client applications (source ports are ephemeral and assigned by the network stack) and assumes that all processes on the OS are equally trustworthy. And it tells us nothing about users associated with a client entity—for that we need to move up the stack.

### 网络分段—Network Segmentation

利用物理交换机、路由器和防火墙的巧妙组合进行网络分段，是计算机安全的基础要素之一。通过将受信任的实体组合到一个私有网段，开发人员可以专注于应用逻辑，而不是访问控制。但是，这种“免费搭车”是有代价的，因为分段方案的任何失败都会使每个实体面临风险。

Network segmentation, using clever combinations of physical switches, routers and firewalls, is one of the foundation elements of computer security. By combining trusted entities into a private segment, developers can focus on application logic, not access control. But this free ride comes at a cost, as any failure in the segmentation scheme puts every entity at risk.

为了降低这种风险，网段应该尽可能的小。将网络划分为多个区域，可以更容易地隔离边界内的入侵。跨越边界需要更高级别的检查，比如安全令牌验证。区域成员（zone membership）的划分应该平衡开发人员经验、运营效率和安全暴露。

To mitigate this risk, network segments should be kept as small as possible. Carving up the network into zones makes it easier to isolate breaches within the boundary. Crossing a boundary should require a higher level of scrutiny, such as security token validation. Zone membership should balance developer experience, operations efficiency, and security exposure.

虚拟世界（包括经典虚拟化和基于容器的网络）使用了一种与物理世界基本相同的分割模型，用软件定义的模拟物代替对应的硬件。像ACL这样的模型在文件系统中的使用让每个人都很熟悉，它使用简洁的访问控制规则简化了网络策略定义。

The virtual world (both classic virtualization and container-based networking) uses a segmentation model that is largely the same as the physical, substituting software-defined analogs for their hardware counterparts. Models like ACLs—which are familiar to everyone from their use in file systems—simplify network policy definition with succinct access control rules.

网络分割的真正问题在于规模。随着网络变得更加复杂，管理区域成员的规则变得难以维护。随着区域和主机数量的增加，攻击表面也会随之增加。

The real problem with network segmentation comes with size. As networks become more complex, the rules governing zone membership become difficult to maintain. And as the number of zones and hosts increase, so too does the attack surface.

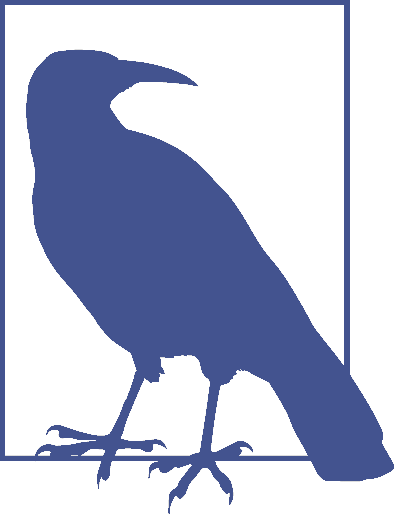
#### SSL/TLS

对不良参与者的机会进行限制一种方法是确保网络分段中的所有通信都使用SSL/TLS。这为传输中的数据提供了保密性和完整性保护，为客户端提供了服务器身份验证，并为服务器添加了重要的(尽管只是可选)客户端身份验证。

One way to limit the opportunities for bad actors is to ensure that all communications in a network segment use SSL/TLS. This provides confidentiality and integrity protection of data in flight, server authentication for clients, and adds important—though optional—client-side authentication for servers.

那么，为什么我们不到处使用SSL/TLS呢?部分原因是惯性。在WEB的早期，SSL的加密要求很高，所以大多数网站将其限制在信用卡传输等关键操作中使用。使用现代cpu时，SSL/TLS的影响可以忽略不计，但过去人们不愿意在所有地方都使用它。

So why don’t we use SSL/TLS everywhere? Part of the reason is inertia. In the early days of the web, the cryptographic demands of SSL were high, so most websites restricted its use to critical operations like credit card transmissions. The impact of SSL/TLS is negligible using modern CPUs, but there is an historical reluctance to use it everywhere. This is a bad web practice we need to resist; all APIs should use SSL/TLS everywhere.

 SPIFFE- Secure Production Identity Framework For Everyone

确保所有流量的安全确实要以可管理性为代价。现代的微服务网络通常是根据12因素原则建立的，这些原则鼓励临时的、无状态的服务。在动态环境中，主机和容器是不断循环的，因此证书和密钥管理很有挑战性。传统的PKI系统在设计时没有考虑到这种工作量。

Securing all traffic does come at a manageability cost. Modern microservices networks are often built to the 12-factor principles, which call for ephemeral, stateless services. In a dynamic environment, where hosts and containers are cycling on a continuous basis, certificate and key management can be challenging. Traditional PKI systems were not designed with this kind of workload in mind.

SPIFFE模型试图简化微服务的身份验证和网络安全配置。SPIFFE为在微服务网络中处理基于X509证书的身份提供了一种开发人员友好的方法。SPIFFE指定了“SVID’s”(SPIFFE可验证身份文档)，用于唯一标识微服务基础设施中运行的组件的证书。

The Secure Production Identity Framework For Everyone (SPIFFE) attempts to simplify microservice authentication and secure network configuration. SPIFFE provides a developer-friendly means for dealing with X509 certificate-based identities in a microservice network. SPIFFE specifies “SVID’s” (SPIFFE Verifiable Identity Documents), certificates used to uniquely identify running components in a microservice infrastructure.

|  |
| --- |
| 何时使用网络分段  1. 当你信任服务器和网络基础设施的物理安全时  2. 当你信任基础设施隔离机制和过程是  3. 当你信任网络分段中的每个实体时  When to Use Network Segmentation 1. When you trust the physical security of the server and network infrastructure 2. When you trust the infrastructure isolation mechanism and process 3. When you trust every entity on the network segment |

### 基线—The Bottom Line for Microservices

网络分段可用于创建微服务分组。基于诸如依赖关系、相似服务之间的自然信任、性能需求、域成员关系（基于Eric Evans的同名书中所描述的领域驱动设计原则）等因素进行分组; 使它们满足开发人员、部署人员或操作人员的需求。在通信和评估框架(如SPIFFE)中使用SSL/TLS，以简化管理。使用具有应用程序级控制的中间人来限制对网段的访问。

Network segmentation can be used to create groupings of microservices. Make groupings based on factors such as dependencies, natural trust between like-services, performance needs, domain membership *(Based on the principles of Domain-driven Design as described in Eric Evans’ book with this same name)*; make them address the needs of developers, deployment, or operations. Use SSL/TLS in communications and evaluate frameworks such as SPIFFE to simplify management. Use an intermediary with application-level controls to restrict access into the network segment.

## 应用级控制—Application-Level Controls

应用实体通过交换安全令牌建立信任。可信的第三方发行令牌，并使用加密技术（跨越信道或在令牌本身内），这样实体可以在没有事先关系的情况下建立信任。令牌信任模型通常基于共享秘密或更常见的公开密钥加密实践。

Application entities establish trust by an exchange of security tokens. A trusted third-party issues tokens and uses cryptography (either across the communications channel or within the token itself) so that entities can establish trust with no prior relationship. Token trust models are usually based on either shared secrets or the more common practice of public-key cryptography.

### 2.3.1 传统Web令牌的问题—The Problem with Traditional Web Tokens

Web会话是开发人员认为理所当然的事情。应用服务器使状态持久化变得非常容易，以至于人们很容易忘记HTTP是一种无状态协议。这里有许多优秀的工程，如果不认识到支撑现代web服务器/浏览器交互所得来之不易的经验教训，那将是一个错误。你当然可以在一个集中式的微服务网络中使用基于cookie的会话，只要你有一个快速的会话存储机制，如Redis来服务每个实例。

Web sessions are something developers take for granted. Application servers make persisting state so effortless, it’s easy to forget that HTTP is a stateless protocol. There is a lot of good engineering here, and it would be a mistake not to recognize the hard-won lessons that underpin a modern web server/browser interaction. You can certainly use cookie-based sessions in a centralized, microservices network, as long as you have a fast session storage mechanism like Redis to serve each instance.

但传统的网络会话有局限性。会话标识符与认证行为绑定在一起，因此它可以作为用户主要认证因素的代理。这就是会话劫持是如此有效的攻击的原因。一旦攻击者获得会话ID，他们就可以执行有效凭据允许的任何操作。

But traditional web sessions have limitations. The Session Identifier binds back to an act of authentication, and so it acts as a proxy for a user’s primary authentication factors. This is why session hijacking is such an effective attack. Once an attacker acquires a session ID, they are able to do anything that valid credentials would permit.

另外一个问题是Web会话不能跨安全域；但是SAML可以用来突破这个限制。SAML不是一个会话机制，而是联邦技术，它允许在不同安全域之间交换认证信息，以及用户的权利和属性。它是一个常用的企业SSO技术。

Another issue is that web sessions don’t cross security domain; however, SAML came about to address that limitation. SAML isn’t a sessioning mechanism, but a federation technology that allows security domains to exchange information about acts of authentication, as well as a user’s entitlements and attributes. It is a common technology for enterprise single-sign on.

SAML向开发人员介绍了一些与微服务高度相关的重要访问控制模式。它将客户端、受保护的资源和身份提供者分开，并明确区分了策略决策点（PDPs—根据安全策略评估令牌的地方）和策略实施点（PEPs—决策被实施的地方）。

SAML did much to introduce developers to some important access control patterns that are very relevant to microservices.

It separated out clients, protected resources and identity providers, and made a clear distinction between Policy Decision Points (PDPs—where tokens are evaluated against a security policy) and Policy Enforcement Points (PEPs—where a decision is enacted).

PDPs可以是集中式的，也可以是高度分布式的

It acknowledged that PDPs could either be centralized or highly distributed (co-located with a PEP protecting a service) to meet security and performance requirements.

SAML引入了一个标准化安全、透明的token，它持有关于认证、授权和属性的相关声明

SAML also introduced a standardized secure, transparent token holding claims about authentication, authorization, and attributes.

它描述了如何安全地传输这些信息，并阐明了本地平和和集中评估间的权衡。

It described how to transmit these safely and articulated the tradeoffs between local and centralized evaluation.

这些想法以不同的形式在现代授权技术中（如OAuth、OpenID Connect和JWT）以不同的形式重复出现

Many of these ideas reappear—though in altered guise—in modern authorization technologies like OAuth, OpenID Connect, and JWT.

然而，SAML不是一个很好的微服务API解决方案。它是一个复杂的技术，过于依赖集中式、正式的信任管理和昂贵的、面向企业的基础设施。对于一个习惯于JSON为中心的APIs的开发者来说，这是一个噩梦。XML token很麻烦，并且endpoints 是SOAP。

SAML, however, is not a good solution for APIs or microservices.

It is a complicated technology, relying too much on centralized, formal trust administration and expensive, enterprise-oriented infrastructure.

To a developer accustomed to JSON-centric APIs, it’s a nightmare.

The XML tokens are cumbersome and the endpoints are SOAP.

但SAML最大的问题是它不能帮助用户在应用程序之间委托授权。

现代Web是建立在这样一个理念之上的：用户应该被授权在不同安全域中拥有的帐户之间建立连接。这代表着身份管理的权力从集中管理转向用户自身的巨大转变。

But biggest problem with SAML is that it doesn’t help users to delegate authorization between applications.

The modern web is built on the idea that a user should be empowered to make connections between the accounts they own in different security domains.

This represents a huge shift in power for identity management—away from central administrators, and toward the users themselves.

### 2.3.2 用于API的现代令牌—Modern Tokens For APIs

新一代以API为中心的安全令牌框架解决了旧web技术中的这些限制。令牌是基于JSON的，协议很容易实现为API endpoint。它们也解决了一个更深层次的问题，即用户对应用程序的隐含信任。

The new generation of API-centric security token frameworks address these limitations in the old web technologies.

Tokens are JSON-based, and protocols are simple to implement as API endpoints.

But they also address a deeper concern about the implict trust a user invests in applications.

新的令牌模型认为我们永远不应该信任一个使用像密码（或其他主要认证因子）一样强大的认证因子的client端或server端应用程序。浏览器可能受到威胁；本机应用程序可能包含误用凭据的恶意代码。

The new token model maintains that we should never trust a client or a server application with something as powerful as a password (or any primary authentication factor).

Browsers can be compromised; native apps might have malicious code to misuse credentials.

现代令牌方案通过将应用程序与身份验证分离来解决这一风险。他们发行具有受限功能的短期token，旨在限制不可信实体的安全暴露。

Modern token schemes address this risk by decoupling applications from authentication. They issue short-lived tokens with constrained capabilities, designed to limit the security exposure from entities that might not be trustworthy.

#### API Keys

API密钥是一种不透明的令牌，用于标识客户端应用程序。许多应用程序可能使用API，因此它对负责该API的产品经理了解流量来自哪里是有用的。API密钥由API的所有者或产品经理颁发给客户端应用程序的开发人员。

API keys are an opaque token intended to identify a client app. Many applications may use an API, so it is useful for a product manager responsible for the API to know where the traffic is coming from. API keys are issued to the developer of a client app by an API’s owner or product manager.

例如，手机上的本机游戏应用将有自己的API密钥。当应用程序调用API端点（API endpoint）时，它会带上此密钥，以便服务能够识别它。API密钥不能识别应用程序的唯一部署实例。它被编译到二进制映像中，因此API密钥在每个安装中都是相同的，“应用程序键（application key）”可能是一个更好的名字。

For example, a native gaming app on a mobile phone would have its own API key. When the app calls an API endpoint, it includes this key so the service can recognize it. An API key does not identify a unique, deployed instance of an app. It will be compiled into the binary image and so is identical across every installation. Application key might have been a better name.

问题就在这里：因为这是一个简单的嵌入式凭证，API密钥可以被的攻击者定位。出于这个原因，**您不应该考虑API密钥具有权威性**。它对于粗略的使用情况进行跟踪或对流量进行管理很有用，但请记住它可能被欺骗。

Herein lies the problem: because this is a simple, embedded credential, API keys can be located by a determined attacker. For this reason, you should never consider an API key authoritative. It is useful for rough usage tracking and traffic management, but always remember it could be spoofed.

#### OAuth 2.0

OAuth2.0是现代应用程序体系结构中安全授权的首选框架。它一开始是在Web站点之间进行授权的简单方式，现在是API授权的主要方式。

OAuth 2.0 is the preferred framework for secure authorization in modern application architectures. What begin as a simple way to delegate authorization between websites is now the primary means of API authorization.

但是很容易将OAuth误解为一种简单的身份认证和会话跟踪机制——基本上是一个类REST的，开发人员做了很多年的更新版

But it is easy to misinterpret OAuth as a simple authentication and session tracking mechanism—basically an updated, REST-like version of what web developers have done for years.

这不仅不准确，而且还忽略了这项技术的真正意义。OAuth框架解决了用户、应用程序和基础设施之间的信任问题，这些问题我们已经忽略多年了。

Not only is this inaccurate, but it misses the real point of this technology. The OAuth framework addresses trust issues between users, applications, and infrastructure we have overlooked for years.

OAuth允许用户在分布式应用程序之间委派访问权限。它不是一个身份认证协议，用来证明用户所声明的身份。

它是一种授权协议，允许用户（资源所有者）授予应用程序（客户端）访问一个API（资源）。该访问的时间有是限的，范围有限的。

OAuth allows users to delegate access between distributed applications. It is not an authentication protocol, which proves a user’s claim to an identity.

It is an authorization protocol that lets a user(the resource owner) grant an app (the client) access to an API (the resource) on their behalf. This access is for a limited time and with limited scope.

OAuth提出的重要一点是，我们永远不应该信任任何使用不受限认证因子（如密码）的应用程序。

这些是我们的钥匙，我们永远无法确定应用程序是会将这些钥匙用于我们的预期目的。

相反，我们应该只信任使用了功能有限、生命周期较短的令牌的应用程序。

The important point OAuth makes is that we should never trust any application with unrestricted authentication factors (such as a password).

These are the keys to our kingdom, and we can never be certain the application will use these keys for our intended purpose.

Instead, we should only trust applications with tokens having limited capability and a short lifespan.

OAuth流程之所以显得如此复杂，是因为它们解决了一个比简单的基于cookie的会话管理困难得多的问题。

The reason OAuth flows appear so complex is that they solve a much more difficult problem than simple cookie-based session management.

有不同的流程以适应功能和限制极其不同的客户端，从浏览器中的JavaScript应用程序（没有安全的本地存储）到原生移动应用程序（功能更强，但受限于供应商应用程序模型），再到桌面应用程序（限制相对更少）。

Different flows exist to accommodate clients with wildly diverse capabilities and limitations, from JavaScript apps in a browser (where there is no secure local storage) to native mobile apps (more capable, but constrained to vendor app model), to desktop apps (where there are relatively few limitations).

我们大多数人认为OAuth是一种网络边缘技术，在组织边界将外部internet客户端和service endpoint连接起来。

但这样理解过于局限了，**Oauth也是管理微服务环境访问的重要技术。**

Most of us think of OAuth as a network edge technology, interfacing external internet clients with the service endpoints at an organizational boundary.

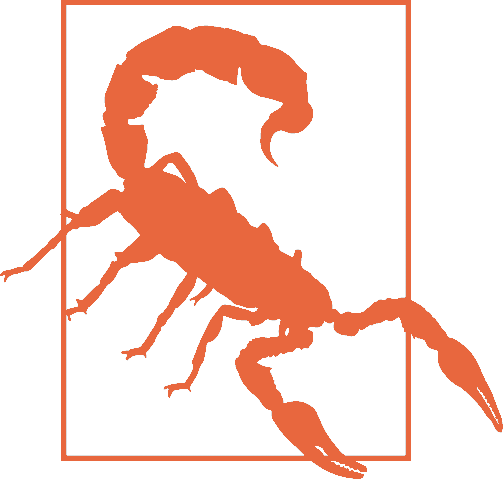
But this is too limiting. Oauth is also an important technology for managing access to microservice environments.

OAuth依赖于由资源所有者执行的同意仪式。

这在微服务环境中并不总是可行的，因为它具有复杂的相互依赖性和不断变化的服务实例环境。

OAuth relies on a consent ceremony performed by resource owners.

This is not always practical in a microservices environment, with its complex interdependencies and ever-changing landscape of service instances.

Should I Use API Keys or OAuth Access Tokens?

记住，API Key标识的是应用程序，而不是用户，这一点很重要。它们很容易被逆向工程，因此它们永远不应该取代用户身份验证。应始终使用OpenID Connect/OAuth对用户进行身份验证和授权。

It’s important to remember that API keys identify an application, not a user. They are easy to reverse-engineer, so they should never be a replacement for user authentication. Always use OpenID Connect/OAuth to authenticate and authorize users.

#### OpenID Connect

OpenID Connect是构建在OAuth框架之上的身份认证层。

OpenID Connect is an authentication layer built on top of the OAuth framework.

OAuth只关心授权，不试图定义如何进行身份认证。

OAuth is concerned only with authorization, making no attempts to define how authentication takes place.

OpenID Connect承担了这一任务，它提供流来验证终端用户，并反馈声明向依赖方。

OpenID Connect takes this on, providing flows to authenticate an end user and provide claims back to a relying party.

与OAuth一样，OpenID Connect也强调了一点，即我们使用的应用程序不可信

Like OAuth, OpenID Connect makes the important point that the apps we use may not be trustworthy.

如果你停下来想一想，这是完全有道理的。你的手机充满了由第三方编写的应用程序；您如何才能确信这些应用不会滥用您的凭证？答案是，当然，你不能！因此，我们需要一种方法将应用程序从身份验证业务中剥离出来。

If you stop and think about this, it makes perfect sense. Your phone is full of apps written by third parties; how can you be confident that these won’t misuse your credentials? The answer is, of course, you can’t—so we need a method to take apps out of the authentication business.

复杂的是应用程序，不仅是他们调用的服务，需要对用户的身份建立信心。应用程序无法从操作系统中派生，因为，一个用户的上下文可能与应用程序中的不同。

The complication is that apps—and not just the services they invoke—need to be confident in the identity of a user. An app can’t derive this from the operating system, as this user context is likely different from that of the app.

OpenID Connect通过使用可信通道对应用程序进行“端运行”以进行身份验证（例如在移动设备上利用本机浏览器）来实现这两个目标。浏览器与身份提供者交互时与应用程序隔离。

OpenID Connect achieves both these goals by doing an end-run around the app using a trusted channel to authenticate, such as leveraging the native browser on a mobile device. The browser interacts with an identity provider in isolation from the app.

身份供程序可以自由地使用任何认证因子的组合执行身份验证——OpenID Connect对此是开放的。反过来，授权服务器向应用程序发出一个ID令牌，该令牌包含主体的身份（用户）和令牌的预期受众的断言。

This provider is free to perform authentication using any combination of factors—OpenID Connect leaves this open. In return, the authorization server issues the app an ID token, which asserts the subject’s identity (the user) and the token’s intended audience.

OpenID Connect在范围和时间上对ID令牌进行限制，这限制了误用的可能性。令牌还为诸如姓名和电话号码等常见声明提供了方便的打包。

OpenID Connect constrains ID tokens in scope and time, which limits the potential for misuse. The tokens also provide a convenient packaging for common claims such as name and phone number.

这是一个包含了很多的简单端点，这就是它的吸引力。OpenID Connect以一种简单、可移植和强大的方式解决了建立信任的首要问题。

This is a lot packed into a few simple endpoints—and that’s its attraction. OpenID Connect solves a first order problem in establishing trust in a way that is simple, portable, and powerful.

**Opaque** tokens versus transparent tokens

构建授权体系结构的最大挑战之一是在集中控制和分散控制之间找到适当的平衡。这通常表现在我们的受保护资源如何处理令牌上。

One of the biggest challenges in building an authorization architecture is finding the right balance between centralized and decentralized control. This usually shows up in how our protected resources handle tokens.

**（1）Opaque Token**

许多OAuth实现使用不透明的访问令牌，有时称为by-reference令牌。通常，这些是随机生成的标识符，攻击者无法猜测集中服务器上的索引状态。资源必须与颁发者进行令牌验证。这具有集中控制的所有优点。它易于管理并提高响应能力，因为很容易使恶意客户端持有的任何活动令牌失效。

Many OAuth implementations make use of opaque access tokens, sometimes called a by-reference token. Usually these are randomly generated identifiers, infeasible for an attacker to guess, that index state on a centralized server. The resource must validate this with the issuer. This comes with all the advantages of centralized control.It is easy to administer and increases responsiveness, as it is easy to invalidate any active tokens held by a rogue client.

但取消对令牌的引用可能代价很高，并且集中化总是会造成瓶颈，从而限制可扩展性和可靠性。

But dereferencing a token can be expensive, and centralization always creates a bottleneck that could limit scalability and reliability.

OAuth的访问令牌和刷新令牌通常是不透明的，并且需要从集中式授权服务器获得验证。

It is common for OAuth’s access tokens and refresh tokens to be opaque and require validation from a centralized authorization server.

**（2）Transparent Token**

另一种方法是使用可在本地决策点解释的透明令牌，这称为by-value令牌。使用透明令牌的体系结构可以通过消除中心化验证的瓶颈得到良好扩展。

在解释令牌时，每个资源还可以应用本地管理的策略；这在跨越组织或地理边界时可能很有价值。OpenID Connect的ID Token是一个透明令牌。

The alternative is to use a transparent token that can be interpreted at a local decision point. This is called a by-value token. Architectures with transparent tokens scale well by removing the central validation bottleneck.

Each resource can also apply locally managed policy when interpreting a token; this can be valuable when crossing organizational or geographical boundaries. OpenID Connect’s ID Token is a transparent token.

#### JWT

JSON Web令牌（JWT）是一种简单的、基于JSON的用于交换声明的打包格式。声明可以是您可以用JSON表示的任何内容；JWT只添加了形式化的header和body、签名机制（JWS）、可选加密（JWE）和简单的web编码。来自OpenID Connect的ID Token是一个JWT。

JSON Web Token (JWT) is a simple, JSON-based packaging format for exchanging claims. The claims can be anything you can represent in JSON; JWT adds only a formalized header and body, a signing mechanism (JWS), optional encryption (JWE), and a simple web encoding. The ID Token from OpenID Connect is a JWT.

声明很重要，因为它让我们可以改进我们的访问控制决策。如果令牌完全不透明，则资源服务器不能对事务应用本地策略。但使用JWT这样的透明令牌，“application=iosTradingApp”这样的声明可以为微服务提供有价值的上下文，从而做出访问控制（或一般服务分发）的决策。

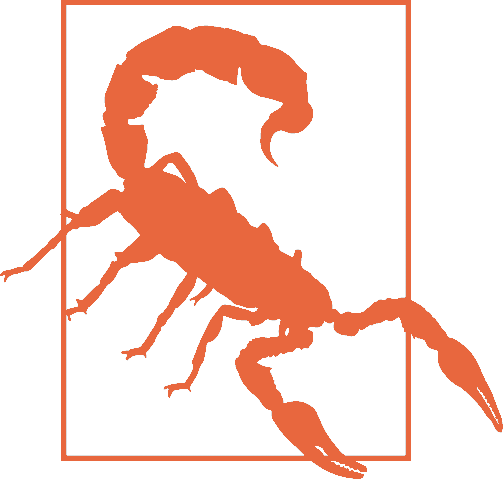
Claims are important because they let us refine our access control decisions. If a token is completely opaque, a resource server has no opportunity to apply local policy on a transaction. But with a transparent token like JWT, a claim such as application=iosTradingApp could provide valuable context to a microservice making access control (or general service-delivery) decisions.

权威声明是访问控制模型（如ABAC）的基础。ABAC模型将授权从以个人身份为中心的决策（允许Bob访问打印机）转变为以属性为中心的规则集（三楼的所有系统都可以访问三楼打印机）。

这是一种适用于微服务的强大技术，微服务须要限制访问，但也需要促进许多不同（持续变化）来源服务的重用。

Authoritative claims are the basis of access control models like attribute-based access control (ABAC). ABAC shifts authorization from individual identity-centric decisions (Bob is allowed access to the printer) to attribute-centric rule sets (All systems on the third floor can access the third-floor printer).

This is a powerful technique to apply to micro-services, which have a need to restrict access but also promote re-use by many different (and continuously changing) sources.

JWT Sessions and JOSE Issues

使用JWT时要小心——它仍然是一种新兴技术。我们建议将它用于授权，但避免使用它来管理用户会话。研究人员还发现JavaScript对象签名和加密（JOSE）堆栈存在问题。早期的实现让攻击者可以伪造令牌，而缺乏版本控制机制意味着规范无法进化到排除弱加密算法。而PAST（Platform-Agnostic Security Tokens）尝试提供一种非标准的替代解决方案。

Be careful if you use JWT—it is still an emerging technology. We recommend using it for authorization, but avoid using it to manage user sessions. Researchers have also identified problems with the JavaScript Object Signing and Encryption (JOSE) stack. Early implementations let attackers forge tokens, and the lack of a versioning mechanism means that the specification cannot evolve to exclude weak encryption algorithms. Efforts such as PAST offer an alternative, though nonstandard, solution.

|  |
| --- |
| 何时使用token  1.您需要对用户和应用程序进行身份验证和授权。  2.您的信任需要跨越边界，可能是跨组织的、地理的、应用程序或虚拟的边界。  3.您可以容忍应用程序和用户之间的仪式。使用token的繁琐过程  4.您拥有完成令牌交换的基础设施。  When to Use Tokens  1. You need to authenticate and authorize users and applications.  2. Your trust needs to cross boundaries, which might be organizational, geographical, application, or virtual.  3. You can tolerate ceremony between applications and users.  4. You have the infrastructure to facilitate the token exchange. |

### 2.3.3 基线—The Bottom Line for Microservices

当授权外部的客户端访问网络边界的端点时，始终使用OAuth 2.0。这种情况下Token可以是不透明的。对于内部跃点（hops），使用带透明JWT令牌的OAuth，以跨越网络区域边界。使用能够对令牌应用本地策略解释的中介（intermediary）来实施跨边界的访问控制。

Always use OAuth 2.0 when authorizing an external client to edge-of-network endpoints. Tokens can be opaque in this use case. For internal hops, use OAuth with transparent JWT tokens to cross boundaries between network zones. Use an intermediary capable of applying local policy interpretation on tokens to enforce access control across the boundary.

## 基础设施—Infrastructure

网络级控制和应用级控制在深思熟虑的安全架构中都有一定的地位。我们在这里提倡的方法是同时利用这两种方法，使我们的信任模型清晰，具有合理的安全性（不妨碍开发、部署和运营）。

实现这种平衡可能是一个挑战，但我们可以依靠一些常见的基础设施元素来帮助我们。

Both network-level controls and application-level controls have a place in a well-thought-out security architecture. The approach we advocate here is to leverage both approaches so that our trust model is clear, with reasonable security that doesn’t hinder development, deployment, and operations.

Achieving this balance can be a challenge, but we can rely on some common infrastructure elements to help us out.

### 2.4.1 代理/网关—Proxy/Gateway

近年来，代理和网关之间的区别已经模糊。两者都是位于HTTP客户端和服务器之间的反向代理。

传统的代理是一个轻量级的网络实体，提供一些可预测的功能，如内容过滤或负载分配。

网关也有同样的功能，但在更高的层次上运作，通过在逐个交易上翻译应用协议来执行复杂的政策。它们是可编程的，通常负责认证、授权、威胁检测和复杂的流量管理。

The distinction between proxies and gateways has blurred in recent years. Both are reverse proxies that stand between an HTTP client and server.

The traditional proxy is a lightweight network entity offering a few predictable functions, such as content filtering or load distribution.

Gateways do the same, but operate at a higher level, enforcing sophisticated policies by interpreting application protocols on a transaction-by-transaction basis. They are programmable and usually responsible for authentication, authorization, threat detection, and sophisticated traffic management.

代理（Proxy）是所有微服务架构的一个重要组成部分。一个微服务的实例可能是短暂的，因此，在任何时间点找到所有可用的实例（称为运行时服务发现）在任何架构中都是必要的。长期以来，这是Web代理的领域，如HAProxy和NGINX。

**现在出现了专门用于微服务的代理，如Envoy、Linkerd和Traefik。**这些代理支持双向（mutual）TLS，并在某些情况下支持简单的令牌验证。

Proxies are an important component of all microservices architectures. An instance of a microservice may be ephemeral, so finding all the available instances at any point in time—called runtime service discovery—is necessary in any architecture. For a long time, this was the domain of web proxies such as HAProxy and NGINX.

Specialized proxies for microservices are appearing, such as Envoy, Linkerd, and Traefik. These allow for mutual TLS, and in some cases simple token validation.

API网关（API Gateway）擅长实施安全策略和应对不寻常的网络挑战，但它们的二进制映像往往很大，而且网关部署起来也很复杂。

然而，新一代轻量级、以微服务为中心的网关正在出现，它可以支持12因素的微服务体系结构。这是一个理想的解决方案，因为可编程网关有助于将应用程序与该领域的快速变化隔离开来。

At the other end of the spectrum are API gateways. These excel at enforcing security policies and accommodating unusual networking challenges, but their binary images tend to be large, and the gateways are complex to deploy.

However, a new generation of light‐weight, microservice-centric gateways are appearing that can underpin a 12-factor, microservices architecture. This is an ideal solution, as programmable gateways help to insulate applications from the rapid change in this space.

### 2.4.2 网络覆盖技术—Network Overlays

许多供应商已经在云或基于容器的网络平台上引入了网络覆盖解决方案，旨在简化微服务网络的安全配置。

OpenTracil和Romana为云基础设施提供网络覆盖解决方案。Calico项目具备对Kubernetes、Docker和Mesos的原生支持。Cilium将新技术引入Linux内核，以修改网络功能。

A number of vendors have introduced network overlay solutions on popular cloud or container-based networking platforms. These are intended to simplify the configuration of secure microservices networks.

OpenContrail and Romana offer network overlay solutions for cloud infrastructures. Project Calico includes native support for Kubernetes, Docker, and Mesos. Cilium introduces new technology to the Linux kernel in order to modify networking capabilities.

#### Kubernetes

Kubernetes是一个用于运行时容器管理的平台。它被广泛用于管理微服务实例的生命周期。每个Kubernetes安装实例由一个或多个Cluster组成，这些Cluster由多个Node组成，每个Node运行着一个或多个容器。POD可以通过定义服务进一步被抽象。

Kubernetes使用服务帐户来标识系统中的组件或组件组。它使用基本身份验证、X.509证书以及多种类型的令牌来认证使用其控制平面API的用户和服务帐户进。以插件方式提供丰富的授权模型，包括RBAC、ABAC以及与其他基础设施集成的web挂钩（web-hook）。

Kubernetes is a platform for run time container management. It is widely used to manage the lifecycle of microservice instances. A Kubernetes installation consists of one or more clusters made up of nodes that run a collection of pods consisting of one or more containers. Pods can be further abstracted by defining services.

Kubernetes uses service accounts to identify components or groups of components within the system. It uses basic authentication, X.509 certificates, as well as multiple token types to authenticate users and service accounts using its control plane API. It offers a rich set of authorization models as plugins, including RBAC, ABAC, and web‐hooks for integrating with other infrastructure.

Kubernetes集群中容器、Pod和外部应用程序之间的数据平面通信实施网络控制。容器只能与同一Node上的容器通信，但“服务”允许容器或Pod使用专用IP地址跨Node通信。网络策略可以充当容器到容器、Pod到Pod以及外部实体到服务通信的ACL。

Data plane communication between containers, pods, and external applications in a Kubernetes cluster uses network controls. Containers can only talk to containers on the same node, but “service” abstraction allows containers or pods to talk across nodes using private IP addresses.

Network policies can act as an ACL for containerto-container, pod-to-pod, and external-entity-to-service communication.

尽管有这些功能，运行在Kubernetes中的微服务所暴露的web API的访问控制通常由微服务自行负责，或者委托给API网关这样的代理负责。

Despite these capabilities, responsibility for access control of web APIs exposed by microservices running in a Kubernetes cluster is generally left to the microservice itself, or to a delegated intermediary such as an API gateway.

#### Cloud Foundry

Cloud Foundry是一个开源的PaaS平台，旨在对基础设施问题进行抽象，并为开发人员提供一个多语言的应用环境。Cloud Foundry 通过路由器组件（称为 Gorouter）将所有入站消息分发给对应应用程序组件，尽管也可以将容器到容器的网络配置为绕过Gorouter。

Cloud Foundry is an open source platform-as-a-service (PaaS) intended to abstract away infrastructure concerns and provide a polyglot application environment for developers.

Cloud Foundry directs all inbound messages to application components through the router component (called the Gorouter), although it is possible to configure container-to-container networking that bypasses this default component.

Cloud Foundry 有一个集中式身份服务器 UAA（用户账户和身份验证），它充当 OAuth2 的授权服务器。UAA 发行签名的 JWT，用于访问平台上运行的组件。

**Cloud Foundry 包括两种类型的应用程序级 ACL：应用安全组用于限制容器可以路由到的网络地址，而容器到容器的网络策略用于限制入站请求。**

Cloud Foundry has a centralized identity server, UAA (User Account and Authentication), that acts as an OAuth2 authorization server. UAA issues signed JSON Web Tokens for accessing components running on the platform.

Cloud Foundry includes two types of application level ACL’s: ***Application Security Groups*** to restrict network addresses containers can route to, and***Container-to-Container Network Policies*** to restrict inbound requests.

在基于 Cloud Foundry 的微服务部署中，Web API 访问最常通过 UAA 签发的访问令牌和网络限制的组合来控制。

In Cloud Foundry-based microservice deployments, web API access is most often controlled through a combination of UAA-issued access tokens and network restrictions. But you can add a thirdparty API gateway to add more specific web API access control policies.

#### Amazon Web Services (AWS)

AWS是世界上最流行的IaaS平台，它由不断增加的数个核心服务组成，包括计算服务EC2，存储服务S3，关系数据库RDS。Amazon引入了ECS（EC2 Container Service）来支持基于容器的应用程序。

Amazon Web Services is the most popular infrastructure-as-aservice (IaaS) platform in the world. AWS consists of an ever-increasing number of core services, including EC2 for compute resources, S3 for storage, and RDS for relational databases. Amazon introduced its EC2 Container Service (ECS) to support Docker based applications.

AWS包括一个内置的身份和访问管理服务,AWS IAM，用于管理用户身份验证和授权。AWS IAM不使用OAuth 2.0、JWT或OpenID Connect。相反，它使用专有机制来传递身份和权限。AWS有一个内置的证书管理服务,AWS certificate Manager，主要用于支持SSL/TLS。

AWS includes a built-in identity and access management service, AWS IAM, for administering user authentication and authorization. AWS IAM does not use OAuth 2.0, JWT, or OpenID Connect.

Instead, it employs proprietary mechanisms for communicating identities and permissions. AWS has a built-in certificate management service, AWS Certificate Manager, mostly used to support SSL/TLS.

在AWS上部署微服务的组织通常使用自己的自部署工具来控制对web API的访问。

AWS有一个API网关；但是，该服务不是为我们在这里描述的微服务安全拓扑而设计的。

It is common for organizations deploying microservices on AWS to use their own self-deployed tools to control access to web APIs.

AWS has an API gateway; however, this service is not designed for microservice security topologies such as we describe here.

## 新兴技术—Emerging Approaches

技术从来不会停滞不前。我们追随时尚，重新包装旧思想。有时我们甚至会想出一些真正新鲜的东西。它使这个行业保持新鲜和令人兴奋，但它使预测未来变得非常困难。一次又一次，我们看到好的想法和优秀的实现仅仅因为缺乏思想共享而输给了较弱的替代方案。

Technology never stands still. We follow fads and re-package old ideas. Sometimes we even come up with things that are genuinely new. It keeps the industry fresh and exciting, but it makes predicting the future very difficult. Time and again, we see good ideas and great implementations lose out to weaker alternatives simply for lack of mindshare.

也就是说，有一些作用于微服务领域的力量是比较可预测的。随着微服务架构的复杂性增加，对基础设施元素抽象画的需求，以及协调所有微服务的指挥和控制系统的抽象化的需求，都会跟着增加。这是在PaaS上的经验，它将我们编排服务生命周期所需的所有组件集合到一个共同的平台。

That said, there are some forces acting on the microservices space that are more predictable.

As microservices architectures increase in complexity, so too will the need to abstract both infrastructure elements and the command-and-control system that coordinates all the underlying pieces. This is our experience with PaaS, which assembles all the components we need to orchestrate service lifecycle into a common platform.

安全性将遵循这个相同的模式。访问控制将包含在平台本身，使用与运行时状态解耦的高级策略来表示。只要平台在如何将策略映射到实现方面是透明的，这就是一件好事。它使我们能够关注信任、威胁和缓解的大局，而这正是我们应该关注的地方。

Security will follow this same model. Access control will be subsumed into the platform itself, expressed using high-level policies decoupled from the runtime state. As long as the platforms are transparent about how they map policy to implementation, this is a good thing. It allows us to focus on the big picture of trust, threats and mitigation, which is where our attention should be.

以策略为中心的方法的一个例子是开放策略代理（OPA），它允许用户定义策略，并使用sidecar容器或嵌入式库在本地强制执行这些策略。OPA的设计支持广泛的策略，并且有一些特定于HTTP API访问控制的示例。

One example of a policy-focused approach is the Open Policy Agent(OPA) allows users to define policies and enforce them locally using a sidecar container or an embedded library. The design of OPA supports a broad range of policies, and there are examples specific to HTTP API access control.

### 2.5.1 服务网格—Service Mesh

服务网格是一种新兴技术，帮助我们管理服务之间的互连。大多数由控制轻量级代理的命令和控制背板组成，作为中间人主动管理服务之间的通信。

其目标是将安全和管理与单个服务解耦，并通过应用于整个平台的通用策略来表达这一点。

Service mesh in an emerging technology that helps to manage interconnections between services.

Most consist of a command-and-control backplane in control of lightweight proxies, acting as intermediaries that actively manage the communication between services.

The goal is to decouple security and management from individual services and express this through generalized policies applied to the platform as a whole.

Istio是一个用于微服务的服务网格平台。它专注于流量管理、安全策略执行和遥测。Istio是一个由谷歌、IBM和Lyft领导的开源项目。

Istio使用特使服务代理提供服务之间的连接。Istio Auth使用来自SPIFFE的SVID来标识和验证服务，以及识别和验证使用了双向认证TLS的服务网格。

目前，Istio Auth依赖于Kubernetes中特定的平台的功能，这可能使其难以支持其他平台。

Istio is a service mesh platform for microservices. It focuses on traffic management, security policy enforcement, and telemetry. Istio is an open-source effort led by Google, IBM, and Lyft.

Istio uses the Envoy service proxy to provide connectivity between services. Istio-Auth uses SVID’s from SPIFFE to identify and authenticate services and a service mesh using **mutually authenticated TLS**.

Currently, Istio-Auth relies on platform-specific capabilities in Kubernetes, which may make it challenging to support other platforms.

### 2.5.2 无服务器计算—Serverless Computing

无服务器计算将抽象化的理念进一步延伸。这里的重要思想是，开发人员的时间最好用于解决业务问题，而不是与基础设施纠缠。无服务器为开发人员提供了一个与部署和生命周期细节隔离的简单代码容器。

Serverless computing extends the idea of abstraction even further. The big idea here is that a developer’s time is best spent solving business problems, not fighting with infrastructure. Serverless gives the developer a simple code-container insulated from the details of deployment and lifecycle.

平台上的事件触发服务的激活。一个对资源的HTTP调用可能是个事件，但重要的是要超越这种明显的联系。达到特定阈值的计数器可能是一个事件，数据库中发生更改的字段也可能是一个事件。对于开发人员来说，这是一个解放思想的想法，他现在可以专注于数据和工作流，将可用性、可扩展性、安全性和计量留给平台。

Events on the platform trigger activation of a service. An HTTP call to a resource might be an event, but it is important to think beyond such obvious connections.

A counter reaching a particular threshold might be an event, or a field changing in a database.

It is a liberating idea for a developer, who can now focus on data and workflows, leaving availability, scaling, security, and metering to the platform.

AWS的Lambda服务是无服务器计算最突出的例子，尽管替代者正在出现。开发人员可以用各种语言（如Java和C#）编写Lambda函数，并将其函数与一组丰富的触发器联系起来。这些触发器可以响应广泛的AWS资源中的事件，从DynamoDB表中的变化到Cloudwatch中的预定事件。AWS Lambda正开始被开发微服务的组织广泛采用。

The AWS Lambda services is the most prominent example of serverless computing, though alternatives are appearing. Developers can write Lambda functions in various languages, such as Java and C#, and associate their functions with a rich set of triggers. These triggers can fire in response to events across a broad range of AWS resources, from changes in a DynamoDB table to scheduled events in Cloudwatch. AWS Lambda is beginning to see widespread adoption by organizations developing microservices.

# Chapter3：一个微服务API安全的通用方法——A General Approach to Microservice API Security

目前在微服务背景下的API访问控制的各种方法强调了其中的复杂性。尽管第2章中提出的解决方案是有用的，但目前还没有一种跨平台的方法能涵盖第1章中的所有要求。

本章提出了一种通用的微服务API访问控制方法--微服务架构的域层次访问规范（DHARMA）--它包含并说明了各种解决方案的情况。

The variety of current approaches to API access control in a microservices context underlines the complexity involved. Although the solutions outlined in Chapter 2 are useful, there is not yet a crossplatform approach that covers all of the requirements from Chapter 1.

This chapter proposes a generalized approach to microservice API access control—Domain Hierarchy Access Regulation for Microservice Architecture (DHARMA)—that incorporates and accounts for the variety of solutions.

## API安全解决方案中的通用模式—Common Patterns in Microservice API Security Solutions

理论上，可以采取单一的方法来保护微服务架构中的每个API端点，用 "零信任 "的思想来实现最大安全性。然而，在实践中，我们已经看到网络、密码学、凭证、令牌和平台都被用来提供不同程度的访问控制。这是为什么呢？

In theory, a singular approach could be taken to protecting every API endpoint in a microservice architecture, with maximum security using a “zero trust” mentality. However, in practice, we have already seen how networks, cryptography, credentials, tokens, and platforms are all being used to provide varying degrees of access control. Why is this?

如前所述，微服务架构的采用是为了帮助组织在扩展的同时优化其软件交付速度和系统稳定性。分布式或去中心化的组织已经认识到，在管理API安全策略并在运行时有效执行这些策略时，不能一刀切。

这种持续的优化需求导致了微服务API安全解决方案的多样性。然而，在异质性中仍存在通用的模式。

As discussed earlier, microservice architecture is employed to help organizations optimize their software delivery speed and system stability while scaling up. Distributed or decentralized organizations have recognized that a one size does not fit all when it comes to administering API security policies and enforcing those policies efficiently at runtime. This continuous need for optimization has led to the diversity of microservice API security solutions. Yet there are

still common patterns in the heterogeneity.

第二章中的每一个解决方案都考虑了API请求或其来源是否可信，以此作为其逻辑的基础。例如，网络隔离假设所有的流量都是可信的，基于证书的访问控制验证了信任链，而基于平台的解决方案依靠平台所在的位置的证明来对API请求进行授权。

验证信任在运行时通常比认证不受信任的消息源更有效。因此，我们可以利用这种受信任/不受信任的API请求的二元性（duality）来优化一般的微服务API安全解决方案。

Each of the solutions in Chapter 2 considers whether or not the API request or its source are trusted as a basis for its logic. For example, network isolation assumes all traffic is trusted, certificate-based access control verifies the trust chain, and platform-based solutions rely on proof of platform residency in order to authorize API requests.

Trust verification is typically more efficient at runtime than authenticating untrusted message sources. As a result, we can use this trusted/untrusted API request duality to optimize a general microservice API security solution.

## DHARMA模型—Domain Hierarchy Access Regulation for Microservice Architecture (DHARMA)

在分布式软件系统中，存在具有各种信任关联的多个实体。为了定义微服务架构中的API安全模型，我们考虑了各个服务之间的信任关系。由于单个服务可能与其他不同的服务具有不同的信任级别，因此提出的方法必须适用于所有级别的系统。

In a distributed software system, there are multiple entities with a variety of trust associations. In order to define a model for API security in a microservice architecture, we consider the trust relationship between individual services. Since an individual service may have different levels of trust with various groups of other services, the proposed approach must be applicable at all levels of system magnification.

为了阐明所提出的微服务API安全性方法，我们首先介绍其基本概念。信任域（或简称域）是一组以特权方式相互通信的服务。域关系是将域中服务分组在一起的原因。域内的服务使用信任机制来验证API请求是否来自可信源。

一个简单的示例，可能存在一个域，服务X和Y部署到特定ECS实例（域关系）中，这些服务通过双向验证的TLS连接（信任机制）进行通信。我们把这个域称为D1。

To articulate the proposed approach to microservice API security, we first introduce its foundational concepts. A trust domain (or simply domain) is a set of services that communicate with each other in a privileged way. The domain relation is the reason for the domain’s services to be grouped together. The trust mechanism is the method used by services within the domain to verify that an API request is coming from a trusted source.

In a simple example, there could be a domain of services X and Y deployed to a specific ECS instance (the domain relation) that communicate over mutually authenticated TLS connections (the trust mechanism). Let’s call that domain D1.

域内的服务可能会从域外的服务或域外其他实体接收API请求。在这种情况下，需要为域定义一个访问控制机制，对这些外部API请求进行认证和授权。

扩展这个例子，服务X可以接受来自外部实体的API请求，其中包括有效的OAuth访问令牌（用于D1的访问控制机制）。

It is possible that services inside a domain may receive API requests from services or other entities outside. In this case, there needs to be a defined access mechanism for the domain that allows these external API requests to be authenticated and authorized.

Extending the simple example, service X may accept API requests from outside entities that include a valid OAuth Access Token (the access mechanism for D1).

正如第1章中所讨论的，一个服务可能有多个API，而一个服务的API可能有多个端点。

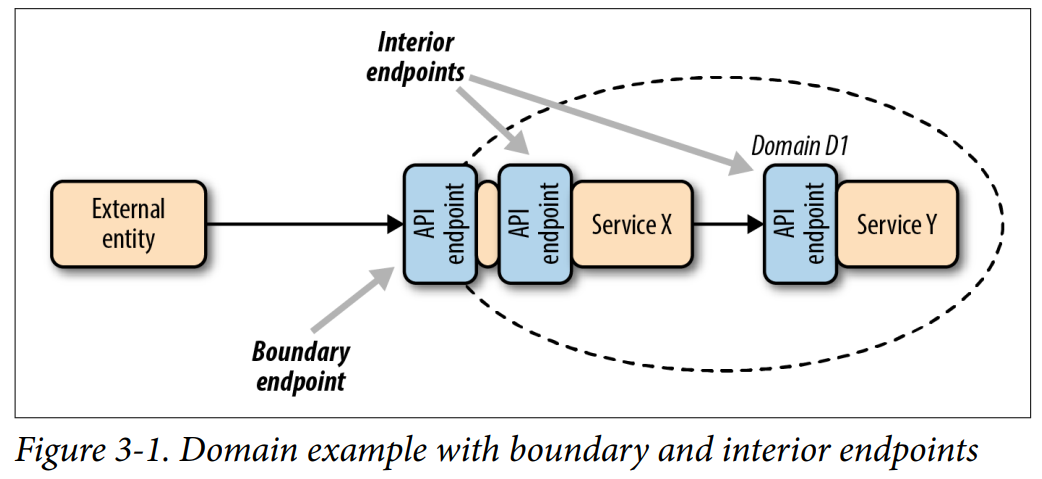
在提出的微服务API安全模型中，内部端点是域内其他服务可以访问的API endpoint。通过域的信任机制对内部端点的访问进行授权。边界端点是一个可以被域外服访问的API endpoint。过通过域的访问控制机制进行授权。

按照我们的简单例子，**服务X同时提供边界端点和内部端点，而服务Y只提供内部端点**。

As discussed in Chapter 1, a single service may have multiple APIs, and a service’s APIs may have multiple endpoints.

In the proposed model for microservice API security, an interior endpoint is an API endpoint that is accessible to other services within the domain. Access to an interior endpoint is authorized through the domain’s trust mechanism. A boundary endpoint is an API endpoint that is accessible to services outside the domain, authorized through the domain’s access mechanism.

Following our simple example, service X offers both a boundary endpoint and an interior endpoint, while service Y only offers an interior endpoint.



现在考虑由服务A、B和C组成的单独信任域-D2，但也包括服务X。D2的信任机制可以是使用有效的OAuth访问令牌，这与D1的访问机制相同。

因此，服务X的API endpoint在D1中是边界端点，而在D2中可以被视为是内部端点。

在我们的模型中，这创造了一个域的层次结构，因为它在D1域内和D2域外之间建立了一个层次的信任关系。

Now consider a separate trust domain—D2—made up of services A,B, and C but that also includes service X. The trust mechanism for D2 could be the use of a valid OAuth Access Token, which is the same as the access mechanism for D1.

Therefore, the API endpoint for service X that is a boundary endpoint in D1 can also be considered an interior endpoint in D2.

In our model, this creates a ***domain hierarchy***, since it creates a hierarchical trust relationship between the ***inner domain D1*** and the ***outer domain D2***.

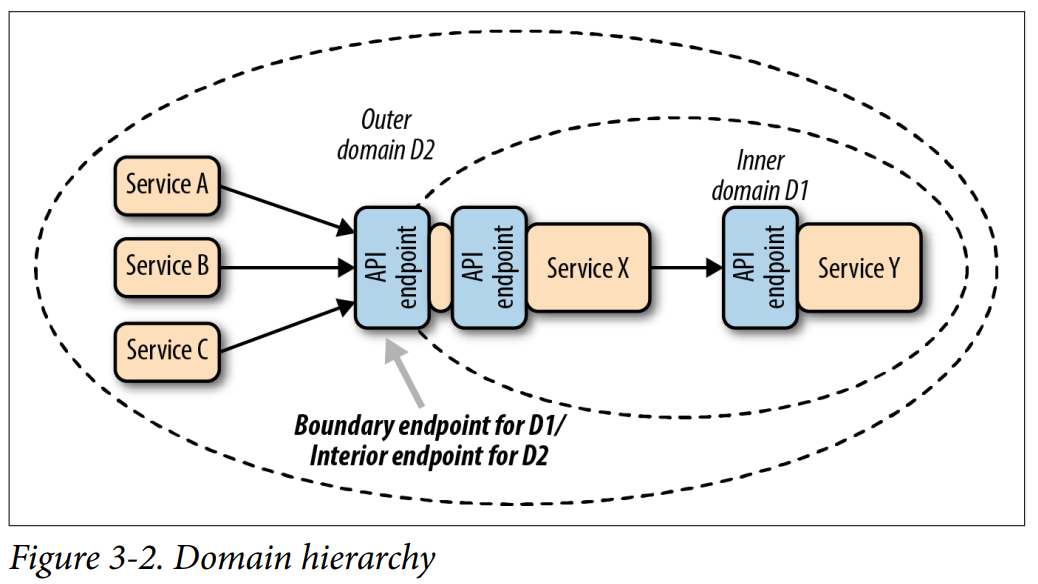
在域的层次结构中，存在着基于已发布 API 端点的服务间通信的隐式规则。在图3-2中，存在于D2，同时不存在于D1中的服务，只能向D1域中服务对外暴露的边界端点发送API请求。

相反，D1中的服务可以向D2中的服务（该服务在D1中不存在）发送API请求，但前提是D1域中的服务能够遵守D2域的信任机制。

Within a domain hierarchy, there are implicit rules about interservice communication based on published API endpoints.

In the Figure 3-2, a service in D2 that is not also in D1 can only send API requests to services in D1 that expose a D1 boundary endpoint.

Conversely, a service in D1 can send API requests to services in D2 (that are not in D1), but only if the D1 service is able to comply with D2’s trust mechanism.



总的来说，我们把本节所描述的模型称为微服务架构的领域层次访问规范，简称DHARMA。

Collectively, we call the model described in this section Domain Hierarchy Access Regulation for Microservice Architecture, or DHARMA for short.

## DHARMA模型设计方法—DHARMA Design Methodology

DHARMA为分析复杂的微服务系统中的信任提供了一个有用的手段。它也可以用来设计此类系统的访问控制方法。设计者可以使用以下DHARMA方法论来为他们微服务架构建立访问控制机制。

DHARMA provides a useful means of analyzing trust in complex systems of microservices. It can also be used to design the access control approach for such systems. Designers can use the following DHARMA methodology to set up access control for their microservice architectures:

（1）Identify trust domains. 识别信任域

对于正在考虑的系统，弄清楚哪些域对于访问控制是重要的。它可以帮助考虑何种域间关系是重要的。

例如，您可能希望对运行在同一平台上的服务、特定网段内的服务或具有特定业务关联性的服务进行分组。还要考虑域的层次结构：如何将一个域内的服务集合进一步细分为内部域？

For the system under consideration, figure out what domains are significant for access control purposes. It may help to consider what domain relations are important.

For example, you may want to group services that run on the same platform, within a specific network segment, or that have a particular business affinity. Also consider the domain hierarchy: how might the collection of services within a domain be further subdivided into inner domains?

（2）Define trust and access mechanisms. 定义信任和访问控制机制。

对于每个领域，定义什么信任机制将被用来保护域内服务之间的API通信。信任机制可能是网络隔离、基于证书的信任方案或基于某些平台的特定功能来实现。

另外，定义将使用什么访问控制机制将被用于允许外部API请求。可以是基于凭证的、基于令牌的或其他一些认证方案。

For each domain, define what trust mechanism will be used to secure API communication between its services. This could be network isolation, certificate-based trust schemes, or platform-specific capabilities.

Also, define what access mechanism will be used to permit external API requests. This could be credential-based, token-based, or some other authentication scheme.

（3）Determine interior and boundary endpoints. 确定内部端点和边界端点

对于每个域，确定其服务的API，并枚举内部和边界的API endpoint。如果可能，此时识别服务之间和跨域边界的已知通信路径非常有用。

For each domain, determine the APIs for its services and enumerate the interior and boundary API endpoints. If possible, it is useful at this point to identify known communication paths between services and across domain boundaries.

（4）Select domain implementation platforms. 选择实现域的平台

对于每个域，选择哪个或哪些平台以及哪些组件将被用于实现API endpoint。API中间人通常被用来实现扩展了内部端点的边界端点。

For each domain, select which platform or platforms and which components will be used for implementing the API endpoints. API intermediaries are often used to implement boundary endpoints that extend interior endpoints.

带着这些实践方法，我们现在可以开发出一个可以在任何平台进行设置的DHARMA应用。

With this practical approach in mind, we can now explore a specific application of DHARMA that is implementable in any platform setting.

## 与平台无关的DHARMA 模型实现—A Platform-Independent DHARMA Implementation

本书的目的正如一开始所述，是在微服务体系结构中定义API访问控制的跨平台方法。DHARMA在普适层面上实现了这一目标。

然而，为了使这本书更实用，本节定义了一个组织可以在任何平台环境中实现的特定DHARMA实例。

The purpose of this book as stated at the outset is to define a cross-platform approach to API access control in a micro-service architecture. DHARMA achieves that purpose on a universal level.

However, to make the book more tangible, this section defines a specific instantiation of DHARMA that an organization can implement in any platform context.

### 域的层次—Domain Hierarchy

此DHARMA的安装实例有3层组成：

1. 内部域是组织中最细粒度服务的分组

2. 外部域由组织中粗粒度的服务组成，这些服务最有可能在整个组织和外部实体中被重复使用。

3. 组织控制之外的区域，可能包括外部实体，他们将向组织对外发布的服务提出API请求。

The domain hierarchy for this instantiation of DHARMA consists of three tiers:

1. Inner domains that are groupings of the organization’s most granular services

2. An outer domain made up of the organization’s coarse-grained services that are most likely to be re-used across the organization, and by external entities.

3. A region outside the organization’s control that may include external entities who will make API requests to the organization’s externally published services.

### 信任和访问机制—Trust and Access Mechanisms

按照DHARMA的设计方法论，现在为识别出的域定义信任和访问控制机制是有意义的。

完成这个目标，显然需要三种身份验证机制：内部域的信任机制、内部域的访问机制（也将是外部域的信任机制和外部域的访问机制）。我们将按相反的顺序确定这些内容。

Following the DHARMA design methodology, it now makes sense to define the trust and access mechanisms for the identified domains.

In doing this, it is clear that three authentication mechanisms are needed: the trust mechanism for the inner domains, the access mechanism for the inner domains which will also be the trust mechanism for the outer domain, and the access mechanism for the outer domain. We will determine these in reverse order.

由于跨外部域边界发出API请求的外部实体不受实施组织的控制，因此外部域访问机制必须灵活且严格。

在开放web的引领下，OAuth 2.0在这里很有意义，特别是结合不透明的访问令牌格式，不能被外部攻击者推导出来。

Since the external entities making API requests across the outer domain boundary are outside the implementing organization’s control, the outer domain access mechanism must be flexible and strict.

Following the lead of the open web, OAuth 2.0 makes sense here, especially in conjunction with an opaque access token format that cannot be derived by external attackers.

对于外部域信任机制（也是内部域的访问机制），我们可以依赖一定程度的组织控制。基于数字证书的信任是一种经验证的、可扩展的建立信任的选项。

事实上，使用组织签发的数字证书签名的JWT可以用来保护终端用户的身份，也可以用来断言发出API请求的服务的身份。

For the outer domain trust mechanism (also the inner domains’ access mechanism), we can rely on a degree of organizational control. Digital certificate-based trust is a proven, scalable option for establishing trust.

In fact, JSON Web Tokens signed using an organization-issued certificate can be used to preserve end-user identity as well as to assert the identity of the service making the API request.

最后，与严格的身份认证相比，内部域信任机制更倾向于对运行时性能的优化。

在这个层面上，以VPC或主机搭配的形式进行网络隔离是可行的。尽管如此，JWT仍然可以在API请求上传递，以保持系统的可问责性和可观察性。

Lastly, the inner-domain trust mechanism will have the strongest optimization bias toward runtime performance as opposed to strictness of authentication.

Network isolation in the form of VPC or host collocation is feasible at this level. Nonetheless, JWT’s may still be passed on API requests in order to maintain system accountability and observability.

### 实现过程中的考虑因素—Implementation Considerations

任何组织在实施这种独立于平台的DHARMA方法时，都有一些考虑因素。访问和信任机制的选择需要一些基本的实践和基础设施，以便使这种方法以一种高性能、可扩展和安全的方式运作。

There are a number of considerations for any organization implementing this platform-independent approach to DHARMA. The access and trust mechanism choices necessitate some foundational practices and infrastructure in order to make this approach work in a performant, scalable, and secure way.

#### 证书管理—Certificate management

由于外部域信任机制依赖于数字证书，因此实施组织必须具备证书颁发机构，向可信服务客户端、服务中间人、平台组件以及服务本身颁发数字证书。

证书撤销是一种有用的功能，但不是必需的。证书粒度是一个关键考虑因素。可以想象，可以向每个服务和每个服务中间人颁发证书，但不是必需的。应至少向每个内部域颁发一个证书。

Since the outer domain trust mechanism relies on digital certificates, the implementing organization must have a certificate authority capable of issuing digital certificates to trusted service clients, service intermediaries, platform components, as well as the services themselves.

Certificate revocation is a useful capability, but not essential. Certificate granularity is a key consideration. It is conceivable that a certificate could be issued to each service and each service intermediary, but not required. There should be at least one certificate issued to each inner domain.

#### 令牌管理—Token management

Token是与平台无关的DHARMA各级别实例的一个基本组成部分。因此，全面的令牌管理——验证、签发、交换和遵从令牌的能力——对实现至关重要。

理论上，一个组织可以为其整个服务域使用一个令牌管理服务器，但建议使用一些分布式的安全令牌服务，以最小化跳数，从而减少与API请求相关的事务延迟。然后，这些分布式令牌服务器可以通过基于证书的信任进行联合。

Tokens are a fundamental component of the platform-independent DHARMA instantiation at all levels. Therefore, comprehensive token management—the ability to validate, issue, exchange, and deference tokens—is essential to the implementation.

Theoretically, an organization could use one token management server for their entire service domain, but it is recommended that some secure token services be distributed to minimize the number of hops and thus the transactional latency associated with API requests. These distributed token servers may then be federated through certificate based trust.

在我们独立于平台的DHARMA实现中，OAuth 2.0被作为外部域的访问机制。这意味着该组织必须实现一个符合OAuth标准的授权服务器。OAuth授权类型将取决于提出API访问请求的外部客户端的类型。对于授权代码和资源所有者密码授权类型，需要终端用户（end user）身份验证，以便外部客户端获得访问令牌。

因此，每个与外部API关联的授权服务器必须能够自行（或通过IAM服务）验证终端用户凭证。

In our platform-independent DHARMA implementation, OAuth 2.0 is used as the access mechanism for the outer domain. This means that the organization must implement an OAuth-compliant authorization server. The OAuth grant type will depend on the type of external client requesting API access. For the Authorization Code and Resource Owner Password grant types, end user authentication is required in order for the external client to obtain an access token.

Therefore, the authorization server associated with each external API must be able to validate end user credentials, either on its own or by accessing the appropriate identity and access management (IAM) services that act as the authority for such credentials.

尽管在DHARMA的实施过程中对使用的令牌没有严格的规定，但这里有一些指南。

推荐在域内使用的JWTs仅有很短的过期时间（少于一个小时）。根据规模和敏感度的不同，它们可能会被签发为一次性使用。

推荐OAuth的范围和JWT的声明将用于携带对授权决策有用的信息。如何使用这些属性将由用户自行决定，但在令牌交换的情况下，OAuth范围可能映射到JWT声明。

最后，所提出的方法没有明确促进OpenID Connect的使用，但推荐使用OIDC令牌应用此平台无关的模型（DHARMA）。

Although there are no strict rules about the tokens used within this DHARMA implementation, here are some guidelines.

It is expected that the JWTs used inside the domains will have a short expiry time (less than an hour). Depending on scale and sensitivity, they may be issued for single use.

It is also expected that OAuth scopes and JWT claims will be used to carry information useful to authorization decisions. It will be at the discretion of how these properties are used, but it is likely that OAuth scopes will be mapped to JWT claims in the case of token exchange. Lastly, the proposed approach does not explicitly promote the use of OpenID Connect, but it is expected that this platform-independent model could be applied using OIDC tokens.

#### 组件指配—Component provisioning

必须安全地设置服务中间人和服务实例。这意味着部署活动必须由经过认证的管理员或具有适当授权的用户代理来执行，并且所有管理活动都要被审计。特别重要的是，必须以最大限度地减少暴露和违反信任的方式将证书提供给服务域内的组件。

Service intermediaries and service instances must be provisioned securely. This means that deployment activities must be performed by authenticated administrators or user agents with appropriate authorization and that all administrative activity be audited. Of particular importance, certificates must be provisioned to components within the service domain in a way that minimizes exposure and violation of trust.

#### 服务和端点部署—Service and endpoint deployment

内部域边界端点的访问机制不需要与授权服务器交互，但API请求确实需要一种到达域内专用网络的方式。例如，在这个与平台无关的DHARMA实现中，内部域的服务必须部署在一个隔离的网络上。除此之外，我们将重点关注API端点的实现位置。

The access mechanism for the inner domains’ boundary endpoints does not require interaction with the authorization server, but API requests do need a way of reaching the inner domain’s private network. For example, in this platform-independent DHARMA implementation, inner domain services must be deployed on an isolated network. Aside from that example, we will focus on the implementation location of the API endpoints.

为了实施OAuth 2.0访问机制，外部域边界端点必须在支持OAuth的组件上实现。为了保持一致性，API中间人在这里就有用了。具体而言，API网关可用于发布边界端点、连接授权服务器进行令牌验证和交换，并将API请求转发到外部域的内部端点执行。

In order to enforce the OAuth 2.0 access mechanism, the outer domain boundary endpoints must be implemented on an OAuth capable component. For consistency, an API intermediary makes sense here. Specifically, an API gateway can be used to publish the boundary endpoints, connect with the authorization server for token validation and exchange, and forward API requests to the outer domain’s interior endpoints for execution.

内部域边界端点的访问机制不需要与授权服务器交互，但API请求确实需要一种到达域内专用网络的方式。将这些边界端点部署到API中间人是有意义的，API中间人能够遍历该网段并处理内部域访问机制（JWT验证）所需的令牌验证。在这种情况下，中间人可以是内部域的本地API网关，也可以是轻量级的服务代理，即现在在服务网格语境中通常理解的组件。边车服务代理可以为外部域的本地服务实例执行类似的角色。

The access mechanism for the inner domains’ boundary endpoints does not require interaction with the authorization server, but API requests do need a way of reaching the inner domain’s private network. It makes sense to deploy these boundary endpoints to an API intermediary capable of traversing that network segment and dealing with the token authentication necessitated by the inner domain’s access mechanism (JWT validation). In this case, the intermediary could either be a local API gateway for the inner domain or a lighter weight service proxy, a component now commonly understood in the context of a service mesh. A sidecar service proxy could perform a similar role for service instances local to the outer domain.

#### 微服务API的可审计性—Microservice API accountability

考虑到微服务体系结构的分布式特性，单个用户请求可能会触发多个审计记录。在这个独立于平台的DHARMA实现中，必须审核所有令牌活动(发布、交换和传播)以及所有授权决策。**API中间人（API网关或服务代理）需要记录这些活动，这也是将它们包含在解决方案中的另一个原因。**

Given the distributed nature of microservice architecture, it is expected that a single user request may trigger multiple audit records. In this platform-independent implementation of DHARMA, all token activities (issuance, exchange, and propagation) must be audited, along with all authorization decisions. API intermediaries (API gateway or service proxy) are expected to log these activities, another reason for their inclusion in the solution.

### DHARMA实现总结—Summary of the Platform-Independent DHARMA Implementation

下表总结了独立于平台的DHARMA实现中处理API请求所涉及的步骤：

The steps involved in handling API requests within the platform independent DHARMA implementation outlined are summarized in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| 交互-Interaction | 身份-Identification | 认证-Authentication | 授权-Authorization |
| 外部客户端请求  External client request | 外部客户端从授权服务器获取访问令牌，通过API请求发送到外部域边界端点  External client obtains access token from authorization server, sends on API request to outer domain boundary endpoint | 接收端的API网关将访问令牌发送到授权服务器进行验证  Receiving API gateway sends access token to authorization server for validation | 授权服务器验证访问令牌、完成JWT的交换，并将其发送回API网关，后者将请求转发到服务的内部端点  Authorization server validates access token, exchanges for JWT, which is sent back to API gateway, which forwards request to service’s interior endpoint |
| 外部域服务到服务的请求，或者外部域到内部域的请求  Outer domain service-to-service request OR outer domain-to-inner-domain request | 服务消费者要么发送先前获得的JWT，要么从授权服务器获取新的JWT，并通过API请求时发送到外部域的内部端点/内部域的边界端点  Service consumer either sends previously obtained JWT or obtains new JWT from authorization server and sends on API request to outer domain interior endpoint/inner domain boundary endpoint | 接收端服务代理验证令牌签名和数字证书链  Receiving service proxy validates token signature and certificate chain | 服务检查JWT声明做相应处理  Service checks JWT claims and processes accordingly |
| 内部域服务到服务请求  Inner domain service-to-service request | 服务消费者要么发送先前获得的JWT，要么从本地安全令牌服务获取新的JWT，并通过API请求发送  Service consumer either sends previously obtained JWT or obtains new JWT from local secure token service and sends on API request | 基于网络隔离的信任  Trusted based on network isolation | 服务检查JWT声明做相应处理  Service checks JWT claims and processes accordingly |

图3-3说明了与平台无关的DHARMA实现中API请求流程。

Figure 3-3 illustrates the platform-independent implementation of DHARMA showing sample API request flows.

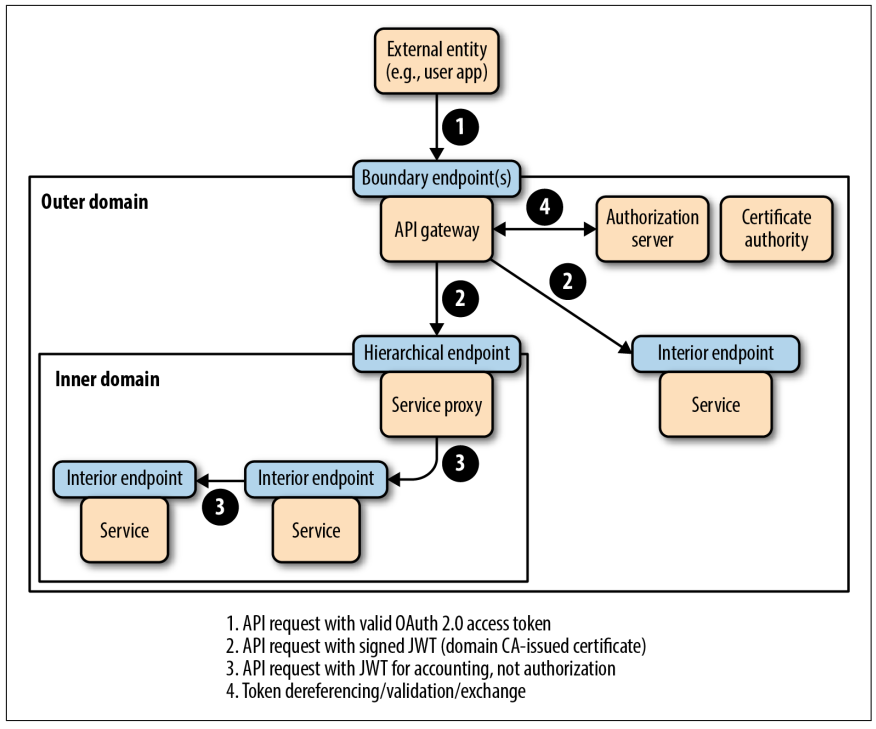


Figure 3-3. A three-tired, platform-independent DHARMA implementation

## 开发人员的经验—Developer Experience in DHARMA

微服务体系结构风格的快速采用是由开发人员推动的，特别是那些在单体应用程序体系结构中被代码协调和部署活动困扰的开发人员。

在转向微服务的过程中，安全功能有可能被视为发布软件的障碍，尽管这些开发人员知道它的重要性。

The rapid adoption of the microservice architectural style has been driven by developers, especially those developers who felt bogged down by the code coordination and deployment activities typical in a monolithic application architecture.

In moving to microservices, security functionality has the potential to be perceived as a **similar** impediment to releasing software, even though these developers know its importance.

DHARMA为开发人员提供了一个全面的方法来解决微服务架构中的API访问控制问题。然而，为了完全满足第1章中提出的需求，从开发人员的角度明确地检查模型是很重要的。

具体来说，当开发人员引入需要满足API安全的微服务，来构建使用安全API的应用时要考虑DHARMA。

假设身份和访问管理基础设施以及其他应用基础设施（如平台和生命周期工具）的职责由集中化团队承担，单个微服务的开发由跨职能开发团队执行。

DHARMA provides a comprehensive method for developers to address API access control in their microservice architectures. However, in order to address the requirements outlined in Chapter 1 completely, it is important to examine the model explicitly from the developer’s perspective.

Specifically, the DHARMA developer experience should be considered when the developer is introducing a new microservice that requires API security, building an application that consumes a secured API, or dealing with a change to the general access control policy of their organization.

It is assumed that the responsibility for identity and access management infrastructure, as well as other application infrastructure (e.g., platforms and lifecycle tooling) lie with centralized teams and that the development of individual microservices is carried out by cross-functional development teams.

### 对服务/API启用访问控制—Enabling Access Control for a Service/API

微服务架构的好处之一是，开发人员可以自由选择语言、框架和平台来开发和运行他们的服务，而DHARMA促进了这一点。与平台无关的DHARMA实现将证书管理、令牌管理和认证策略执行委托给中间人。因此，在设计、开发和部署服务时，开发人员必须首先考虑与API访问控制相关的三件事。首先，他们必须知道服务将部署到哪个域中。其次，他们必须考虑在对入站API请求授权时如何使用JWT声明。最后，他们必须确定如何审计服务中的API请求。除了这三个关键领域之外，开发人员还应该确定JWT信息是否将用于进一步的下游处理。

One of the stated benefits of a microservice architecture is that developers are free to choose the language, framework, and platform to use for developing and running their services, and DHARMA facilitates this. The platform-independent DHARMA implementation delegates certificate management, token management, and authentication policy enforcement to intermediaries. Therefore, there are primarily three things developers must consider related to API access control when designing, developing, and deploying their service. First, they must know into which domain the service will be deployed. Secondly, they must consider how JWT claims will be used in authorizing inbound API requests. Lastly, they must determine how API request access is audited within the service. In addition to these three critical areas, developers should also determine whether the JWT information will be used for further downstream processing.

### 发布和发现API访问控制策略—Publishing and Discovering API Access Control Policies

对于消费微服务API的开发者，应提供正确的信息以允许无缝的访问。这意味着API的访问控制策略应该清晰地表达出来，并且让这些消费API的开发人员容易访问到。OpenAPI——采用最广泛的API描述格式，它使用一个访问控制词汇表来促进此类文档描述（documentation）。

访问控制策略所遵循的方法会有所不同，取决于使用了何种服务消费者。但是，提供服务的组织可能希望提供帮助库或其它工具，使消费服务的开发者有尽可能的无障碍体验。

For developers consuming microservice APIs, providing the right information to permit access should be as seamless as possible. This means that an API’s access control policies should be clearly articulated and easily accessible to these consuming developers. OpenAPI—the most widely adopted API description format—includes an access control vocabulary to promotes such documentation. The method used to abide by the access control policy will vary, depending on what type of service consumer is being used, but the service providing organization may want to offer helper libraries or other tools to make the consuming developer’s experience as frictionless as it can be.

### 访问控制策略的变更—Access Control Policy Change Management

如何处理通用功能的变更是任何软件潜在的复杂性之一，例如组织级别的安全策略。对于微服务体系结构中的API访问，更改访问控制策略可能会影响服务域中的所有涉众，包括微服务开发人员。

为了解决这个问题，与平台无关的DHARMA实现隔离了大部分策略执行，另外将策略逻辑关联到由中心化团队控制的服务中间人。

这与在共享库中提供通用功能不同，当策略发生变化时，这种方法对服务开发人员的影响要大得多。Ben Christensen在他的演讲“不要建立一个分布式的单体应用”中详细阐述了共享库的危险。

One of the essential complexities of any software system is how to deal with change to universal capabilities, such as organization-wide security policies. For API access in a microservice architecture, there is a risk that changing the access control policy would impact all stakeholders in the service domain, including microservice developers.

To address this, the platform-indepenent DHARMA implementation isolates much of the policy enforcement, and—by association—policy logic into service intermediaries controlled by centralized teams.

This contrasts with offering common functionality in shared libraries, an approach has a much larger impact on service developers when policies change. Ben Christensen elaborated on the dangers of shared libraries in his talk “Don’t Build a Distributed Monolith”.

本章介绍了DHARMA方法，一种在微服务系统中定义API访问控制的通用方法。

然后我们介绍了一种应用DHARMA的设计方法，以及详细介绍了与平台无关的DHARMA实现。最后，我们从开发者体验的角度对DHARMA进行了检查。下一章将探讨更多可以应用DHARMA的领域。

This chapter introduced Domain Hierarchy Access Regulation for Microservice Architecture (DHARMA), a universal approach to defining API access control in a system of microservices. We then introduced a design methodology for applying DHARMA, as well as detailing a platform-independent implementation of DHARMA. Lastly, we examined DHARMA from a developer experience perspective. The next chapter examines more areas where DHARMA can be applied.

# Chapter4：结论：微服务安全边界—Conclusion: The Microservice API Security Frontier

本书的前三章提供了一个实用的目的：综述了微服务API安全总览及其需求，回顾当前业界可用的解决方案，最重要的是定义一个与平台无关的方法来用来保护微服务体系结构中的WEB APIs。然而，作为次要目的，我们希望这里介绍的概念和方法可以帮助弥补现有的差距，并对微服务体系结构和API安全的新领域进行探索。

The first three chapters of this book serve a practical purpose: to outline the microservice API security landscape and its requirements, to review the current solution options available in the industry, and most importantly to define a platform independent approach to securing web APIs in a microservice architecture. However, as a secondary purpose, we hope that the concepts and approaches introduced here can help to cover existing gaps and explore new areas of microservice architecture and API security.

## 标准化微服务语言—Standardizing the Language of Microservices

本书通过第2页的“微服务API概览”和图3-2中对DHARMA基本概念的定义，提出了一个关于微服务架构中API安全的概念性词汇表。考虑到微服务方法的范围和流行程度的增长，我们希望这个词汇表的使用可以超越API安全这个范围，并帮助软件架构师在处理复杂的微服务系统时开发一致的语言。

This book proposes a conceptual vocabulary for API security in amicroservice architecture, through “The Microservice API Landscape” on page 2 and the definition of DHARMA’s foundational concepts in Figure 3-2. Given the growth in scope and popularity of the microservices approach, we hope this vocabulary can be used beyond the API security scope and help software architects develop consistent language when working with complex systems of microservices.

## 应用DHARMA模型—Applying DHARMA

第3章详细描述了如何使用与平台无关的访问和信任机制来实现DHARMA。尽管如此，仍然有可能使用特定于平台的机制来实现DHARMA，比如在第2章中列出的那些机制。用于服务发现和动态配置的服务注册服务如Consul和etcd也可以在安全领域扮演一定角色。我们希望DHARMA能被用来阐明和澄清现有的微服务API安全方法，并能被用来探索和开发新的方法。

Chapter 3 includes a detailed description of how DHARMA can be implemented using platform-independent access and trust mechanisms. Still, it is quite possible to implement DHARMA using platform-specific mechanisms such as those listed in Chapter 2. It is expected that the service registries such as Consul and etcd that are used for service discovery and dynamic configuration could play a role in the security landscape as well. We hope that DHARMA can be used to articulate and clarify existing microservice API security approaches, and that it can be used to discover and develop new ones.

## 扩展DHAEMA模型—Extending DHARMA

除了WebAPI之外，在控制微服务的数据平面访问方面还有更多的工作要做。随着微服务实现中响应式、基于事件的体系结构的日益流行，用于通信的新协议正在出现，特别是微服务之间的通信。gPRC——最初由谷歌开发现在由云原生基金会（CNCF）管理——提供原生HTTP2支持和比JSON更紧凑的二进制序列化格式(协议缓冲区，或protobuf/ **protocol buffers, or protobuf**)。Apache Thrift在优化消息大小方面与protobuf类似。多个异步消息传递协议——RabbitMQ、Apache Kafka、NATS——在事件分发和流媒体中使用。尽管如此，在互操作访问控制机制方面，这些协议中没有一个接近WEB API的成熟度。

There is much more ground to cover in controlling data plane access for microservices beyond web APIs. With the increasing popularity of reactive, event-based architecture in microservice implementations, new protocols are emerging for communication, particularly between microservices. gRPC—originally developed by Google but now under the stewardship of the Cloud Native Computing Foundation—offers native HTTP2 support and a binary serialization format (protocol buffers, or protobuf) that is more compact than JSON.Apache Thrift is similar to protobuf in optimizing for message size. Multiple asynchronous messaging protocols—RabbitMQ, Apache Kafka, NATS—are being used in event distribution and streaming. Still, none of these protocols are anywhere near the maturity of web APIs when it comes to interoperable access control mechanisms.

由于其抽象的特点，DHARMA有潜力被用作包括所有协议的通用数据平面访问控制方法。同时，本书应该帮助正在实施微服务的组织，特别是那些使用多个平台进行部署和托管的组织，定义一种安全且可扩展的方法来控制对微服务API的访问。

With its abstract beginnings, DHARMA has the potential to be used as a generalized data plane access control approach that includes all protocols.In the meantime, this book should help organizations that are implementing microservices—especially those using multiple platforms for deployment and hosting—define a secure and scalable approach for controlling access to the microservices’ APIs.