O'REILLY®

Securing Microservice APIs

Sustainable and Scalable Access Control

Matt McLarty, Rob Wilson
& Scott Morrison

Building Your Secure Microservice Architecture in Practice

This eBook is brought to you by CA Technologies - an analyst-acclaimed leader in full lifecycle API and microservices management.

Try these tools from CA to help build your microservices architecture for speed, scale and safety:

CA Live API Creator

Create microservices with enterprise-class APIs in minutes

CA Microgateway

Optimize network traffic and build a consistent and secure service-mesh with SSL/TLS, OAuth and service discovery in each Docker™ container.

CA API Gateway

Aggregate microservice APIs into valuable business APIs for secure, scalable and developer-friendly external consumption

CA API Management

Enable developers to discover and register to use your business APIs with automated API docs and testing tools

CA API Management - CA Application Performance Management

Analyze API usage and performance from app to container to assess operations, business impact and to diagnose issues

Learn more and start a trial today at:

ca.com/microservices >

Securing Microservice APIs

Sustainable and Scalable Access Control

Matt McLarty, Rob Wilson, and Scott Morrison

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

Securing Microservice APIs
by Matt McLarty, Rob Wilson, and Scott Morrison

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Foster
Production Editor: Colleen Cole
Copyeditor: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

February 2018: First Edition

Revision History for the First Edition
2018-01-29: First Release

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and CA Technologies. See our *statement of editorial independence*.

978-1-492-02711-9

[LSI]

Table of Contents

Preface

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.

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.

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

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.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

The authors would like to thank Alan Marion, Tarun Khandelwal, Irakli Nadareishvili, Mike Sample, Sascha Preibisch, and Josh Chiang for their invaluable contributions to the report. Thanks also to Shiu Fun Poon, Kin Lane, Ronnie Mitra, and Daniel Bryant for their helpful feedback.

CHAPTER 1

Microservice Architecture

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.

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.

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

The Microservice API Landscape

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.

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.

[1] API endpoints are often listed in service registries like Consul, Eureka, or etcd.

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

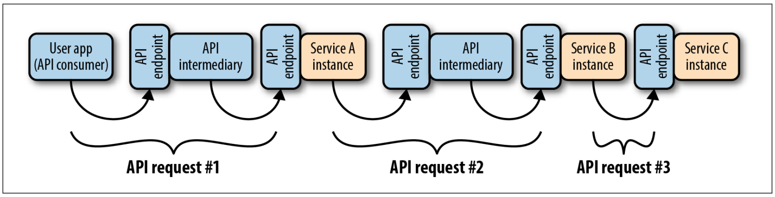

Figure 1-1. Sample API requests in a microservice architecture

API Access Control for Microservices

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.

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.

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.

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.

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.

Microservice Architecture Qualities

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.

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.

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.

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.

CHAPTER 2
Access Control for Microservices

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.

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.

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.

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.

Establishing Trust

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.

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.

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.

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.

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.

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.

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.

Network-Level Controls

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.

Localhost Isolation

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.

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 well-known ports to specific applications.

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.

Network Segmentation

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.

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.

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.

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

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.

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

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.

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.

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

The Bottom Line for Microservices

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;[1] make them address the needs of developers, deployment, or operations. Use SSL/TLS in communications and evaluate

[1] Based on the principles of *Domain-driven Design* as described in Eric Evans' book with this same name.

frameworks such as SPIFFE to simplify management. Use an intermediary with application-level controls to restrict access into the network segment.

Application-Level Controls

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.

The Problem with Traditional Web Tokens

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.

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.

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 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). 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 also introduced a standardized secure, transparent token holding claims about authentication, authorization, and attributes. It described how to transmit these safely and articulated the tradeoffs between local and centralized evaluation. Many of these ideas reappear—though in altered guise—in modern authorization technologies like OAuth, OpenID Connect, and JWT.

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.

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.

Modern Tokens For APIs

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.

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. 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 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.

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.

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

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. 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. 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 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.

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.

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

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 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?

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 is an authentication layer built on top of the OAuth framework. OAuth is concerned only with authorization, making no attempts to define how authentication takes place. OpenID Connect takes this on, providing flows to authenticate an end user and provide claims back to a relying party.

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

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 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. 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 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.

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

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.

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. But dereferencing a token can be expensive, and centralization always creates a bottleneck that could limit scalability and reliability. It is common for OAuth's access tokens and refresh tokens to be opaque and require validation from a centralized *authorization server*.

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 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.

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.

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 microservices, which have a need to restrict access but also promote re-use by many different (and continuously changing) sources.

JWT Sessions and JOSE Issues

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.[2] 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.[3] Efforts such as POST offer an alternative, though nonstandard, solution.[4]

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.

The Bottom Line for Microservices

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.

Infrastructure

Both network-level controls and application-level controls have a place in a well-thought-out security architecture. The approach we

[2] "Stop using JWT for sessions", last modified 13 Jun 2016.

[3] "No Way, JOSE! Javascript Object Signing and Encryption is a Bad Standard That Everyone Should Avoid", last modified March 14, 2017.

[4] "PAST: Platform-Agnostic Security Tokens".

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.

Proxy/Gateway

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.

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.

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 lightweight, 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.

Network Overlays

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.

PaaS

Popular microservices platforms offer a variety of access control abstractions to reduce complexity for operators and developers. This requires trust in the PaaS—which is a consideration you must never take lightly. However, used in combination with elements like proxy/gateways, PaaS offers a very powerful microservices platform for little invested effort.

This section explains the API access control mechanisms in available in Kubernetes, Cloud Foundry, and AWS.

Kubernetes

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 webhooks for integrating with other infrastructure.

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 container-to-container, pod-to-pod, and external-entity-to-service communication.

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 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 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.

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 third-party API gateway to add more specific web API access control policies.

Amazon Web Services (AWS)

Amazon Web Services is the most popular infrastructure-as-a-service (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 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.

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.

Emerging Approaches

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.

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.

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.

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.

Service Mesh

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 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.

Serverless Computing

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.

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.

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.

CHAPTER 3

A General Approach to Microservice API Security

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 cross-platform 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.

Common Patterns in Microservice API Security Solutions

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?

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.

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.

Domain Hierarchy Access Regulation for Microservice Architecture (DHARMA)

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.

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.

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).

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.

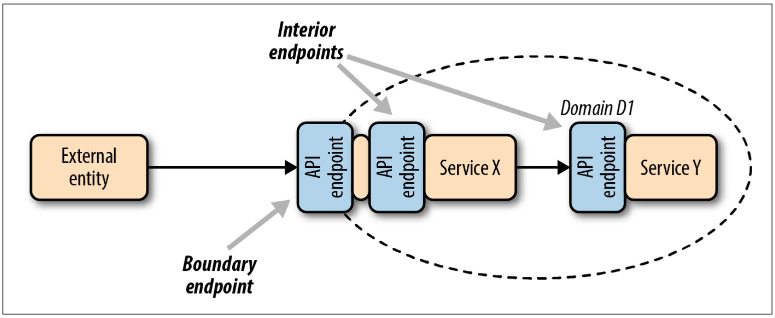

Figure 3-1. Domain example with boundary and interior endpoints

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.

Wthin 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.

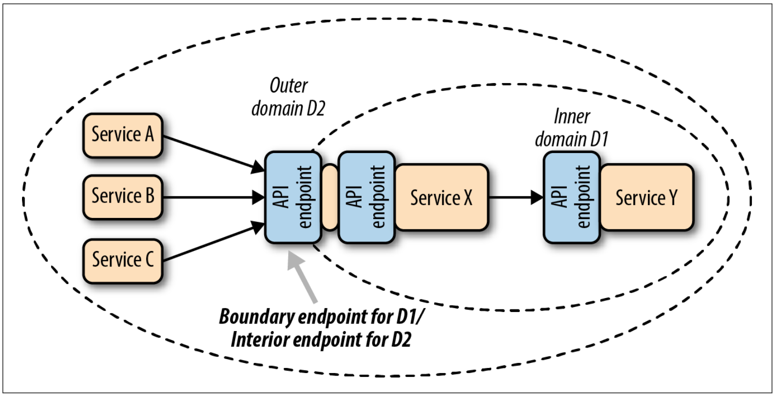

Figure 3-2. Domain hierarchy

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

DHARMA Design Methodology

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:

An API Access Control Design Methodology Using DHARMA

1. **Identify trust domains.** For the system under consideration, figure out what domains are signficant 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.** 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 capabili-

ties. 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.** 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.** 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.

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

A Platform-Independent DHARMA Implementation

The purpose of this book as stated at the outset is to define a cross-platform approach to API access control in a microservice 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.

Domain Hierarchy

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

Trust and Access Mechanisms

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.

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.

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.

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.

Implementation Considerations

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.

Certificate management

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 certficate issued to each inner domain.

Token management

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.

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.

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.

Component provisioning

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.

Service and endpoint deployment

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.

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.

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.

Microservice API accountability

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.

Summary of the Platform-Independent DHARMA Implementation

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

Interaction	Identification	Authentication	Authorization
External client request	External client obtains access token from authorization server, sends on API request to outer domain boundary endpoint	Receiving API gateway sends access token to authorization server for validation	Authorization server validates access token, exchanges for JWT, which is sent back to API gateway, which forwards request to service's interior endpoint
Outer domain service-to-service request OR outer-domain-to-inner-domain request	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	Receiving service proxy validates token signature and certificate chain	Service checks JWT claims and processes accordingly
Inner domain service-to-service request	Service consumer either sends previously obtained JWT or obtains new JWT from local secure token service and sends on API request	Trusted based on network isolation	Service checks JWT claims and processes accordingly

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

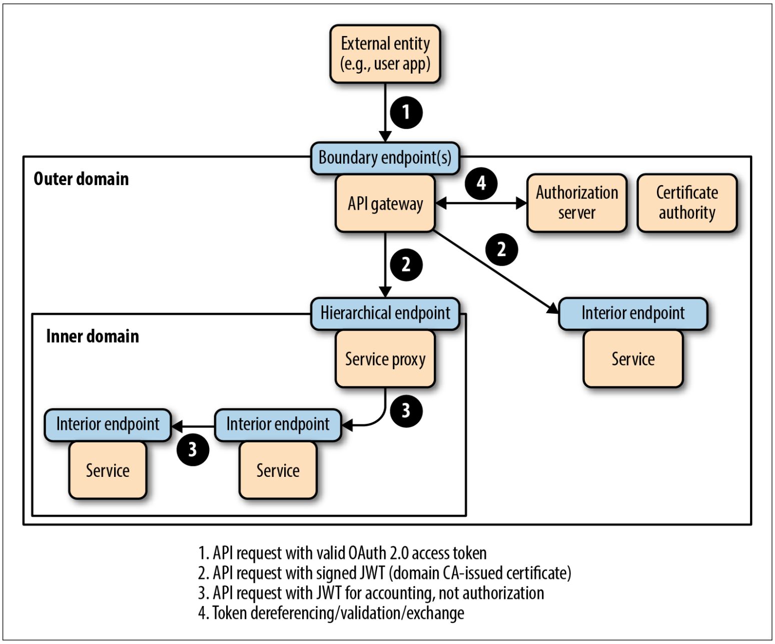
External entity (e.g., user app)
1
Boundary endpoint(s)
Outer domain
API gateway
4
Authorization server
Certificate authority
2
2
Hierarchical endpoint
Interior endpoint
Inner domain
Service proxy
Service
3
Interior endpoint
Interior endpoint
3
Service
Service
1. API request with valid OAuth 2.0 access token
2. API request with signed JWT (domain CA-issued certificate)
3. API request with JWT for accounting, not authorization
4. Token dereferencing/validation/exchange

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

Developer Experience in DHARMA

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 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.

Enabling Access Control for a Service/API

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.

Publishing and Discovering API Access Control Policies

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.

Access Control Policy Change Management

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".

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.

CHAPTER 4

Conclusion: The Microservice API Security Frontier

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.

Standardizing the Language of Microservices

This book proposes a conceptual vocabulary for API security in a microservice 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.

Applying DHARMA

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.

Extending DHARMA

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. 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.

APPENDIX A

Helpful Resources

For more information on the concepts and technologies introduced in this report, please visit the following links.

API and Microservices Practices

- 12-factor application principles
- "API Design for Microservices" lesson
- Ben Christensen's talk "Don't Build a Distributed Monolith"
- Domain Driven Design (DDD) community
- Developer Experience (DX) for APIs
- James Lewis and Martin Fowler, "Microservices"
- Irakli Nadareishvili, Ronnie Mitra, Mike Amundsen, and Matt McLarty, *Microservice Architecture* (O'Reilly)
- Sam Newman, *Building Microservices* (O'Reilly)

Emerging Microservice Technologies

- Apache Kafka
- AWS Lambda
- CA Microgateway (service proxy)
- Cilium network security
- Cloud Foundry concepts

- Consul service discovery
- Envoy service proxy
- gRPC
- Istio policy management
- Kubernetes concepts
- Linkerd service proxy
- OpenContrail network virtualization
- Open Policy Agent
- Project Calico network security
- Romana network virtualization
- Secure Production Identity Framework for Everyone (SPIFFE)
- SPIFFE Verifiable Identity Documents

About the Authors

Matt McLarty is vice president of the API Academy at CA Technologies. The API Academy helps companies thrive in the digital economy by providing expert guidance on strategy, architecture, and design for APIs. He is an experienced instructor, speaker, and one of the authors of the acclaimed book, *Microservice Architecture* (O'Reilly).

Rob Wilson has been working in the field of information technology for over 20 years. He enjoys working on complex and diverse issues where the analysis of situations requires an in-depth evaluation of numerous factors, as well as ingenuity and originality to solve. Today much of his time is spent working with clients on their API and microservices strategies. Rob holds a bachelor's degree in technology management from Memorial University, and master's in information technology from the University of Liverpool. When not working with clients Rob enjoys outdoor activities with family, gaming, and having lively and engaging conversations.

Scott Morrison is a senior vice president and Distinguished Engineer at CA Technologies. He joined CA as part of its acquisition of Layer 7 Technologies, where he served as CTO. Scott is a passionate, entertaining, and highly sought-after keynote speaker. His quotes appear regularly across media, including the *New York Times*, the *Wall Street Journal* and CNN. He has co-authored academic papers in medical, physics, and engineering journals, and holds 8 US patents. Scott lives with his family in Vancouver, BC.