

LEARNING MADE EASY

Data Theorem Special Edition

API Security

for
dummies[®]
A Wiley Brand



Growth of the
API economy

—
Address API security
challenges

—
Automate protection
with DevSecOps

Brought to you
by

datatheorem

Emily Freeman


About Data Theorem

Data Theorem is a leading provider of modern application security, helping customers prevent AppSec data breaches. Its products focus on API security, cloud (serverless apps, CSPM, CWPP, CNAPP), mobile apps (iOS and Android), and web apps (single-page apps). Its core mission is to analyze and secure any modern application anytime, anywhere. The award-winning Data Theorem Analyzer Engine continuously analyzes APIs, Web, Mobile, and Cloud applications in search of security flaws and data privacy gaps. The company has detected more than 1 billion application incidents and currently secures more than 8,000 modern applications for its enterprise customers around the world. Learn more at www.datatheorem.com

Free API Attack Surface Calculation <https://apicalculator.datatheorem.com/>

API Attack Surface Calculator

1 Web — 2 Mobile — 3 APIs — 4 Clouds — 5 Cloud Services — 6 Framework — 7 Compliance



Do you have public web apps with APIs?

Select one

Yes

No

BACK NEXT



API Security

Data Theorem Special Edition

by Emily Freeman

for
dummies[®]
A Wiley Brand

API Security For Dummies®, Data Theorem Special Edition

Published by

John Wiley & Sons, Inc.

111 River St.

Hoboken, NJ 07030-5774

www.wiley.com

Copyright © 2020 by John Wiley & Sons, Inc., Hoboken, New Jersey

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Trademarks: Wiley, For Dummies, the Dummies Man logo, Dummies.com, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries, and may not be used without written permission. Data Theorem and the Data Theorem logo are trademarks or registered trademarks of Data Theorem, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, or how to create a custom *For Dummies* book for your business or organization, please contact our Business Development Department in the U.S. at 877-409-4177, contact info@dummies.biz, or visit www.wiley.com/go/custompub. For information about licensing the *For Dummies* brand for products or services, contact BrandedRights&Licenses@Wiley.com.

ISBN 978-1-119-63962-6 (pbk); ISBN 978-1-119-63963-3 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Publisher's Acknowledgments

We're proud of this book and of the people who worked on it. Some of the people who helped bring this book to market include the following:

Project Editor: Martin V. Minner

Editorial Manager: Rev Mengle

Acquisitions Editor: Ashley Coffey

Business Development

Representative: Karen Hattan

Production Editor: Siddique Shaik

Table of Contents

INTRODUCTION	1
About This Book	2
Icons Used in This Book.....	2
CHAPTER 1: Understanding APIs.....	3
Digging into API Architecture	4
Addressing HTTP	4
Executing actions	4
Managing APIs	5
Modernizing your systems	5
Communicating with protocols.....	5
Architecting Modern APIs.....	6
RESTing on your service layer	6
Separating concerns.....	6
Testing code.....	6
Documenting APIs	7
Keeping your code agnostic	8
Working with Legacy APIs.....	8
CHAPTER 2: Growing the API Economy	9
Competing in Your Market.....	10
Moving to the Cloud.....	10
Enabling Microservices	10
Building a Single-Page App (SPA).....	11
Utilizing Mobile Apps	11
Creating SDKs	12
Consuming Open Source Software	13
Scaling with Serverless	13
CHAPTER 3: Introducing DevOps.....	15
Transitioning to DevOps.....	15
Applying Security to DevOps.....	16
Understanding the Software Delivery Life Cycle	17
Shifting Security Left	18
Architecting for Security	18
Monitoring and Alerting	19

CHAPTER 4:	Securing APIs with Traditional Approaches	21
	Two Types of APIs.....	22
	Productizing customer-facing APIs.....	22
	Internalizing APIs.....	22
	Shadow APIs.....	22
	Securing APIs with Legacy Approaches.....	23
	(Pen)testing manually.....	23
	Using an API gateway.....	24
	Creating web application firewalls.....	24
	Dealing with the Consequences of Traditional API Security.....	25
CHAPTER 5:	Modern Approaches to Securing APIs	27
	Catching Bugs Early via CI/CD.....	27
	Injecting checks.....	28
	Looking for dangerous code.....	28
	Testing in CI/CD.....	29
	Documenting APIs.....	29
	Validating the Security of Your APIs in Production.....	30
	Auto-Enforcing Policies.....	31
	Tracking Changes.....	32
	Closing the Loop.....	32
CHAPTER 6:	Benefitting from an API Security Framework	33
	Streamlining Security Processes.....	33
	Security at Key Stages of Development.....	34
	Releasing with CI/CD Pipelines.....	34
	Tracking Bugs.....	35
	Accelerating Incident Response.....	36
CHAPTER 7:	Automating Your API Security Framework: Introducing DevSecOps	37
	Evaluating Risk.....	37
	Securing Continuously.....	38
	Testing.....	39
	Monitoring and scanning.....	39
	Managing Security Incidents.....	40
	Alerting developers.....	40
	Creating playbooks.....	40
	Tooling Your DevSecOps Practice.....	41
CHAPTER 8:	Ten Takeaways	43

Introduction

Security is an afterthought in tech, like it or not. Engineers often say security should be a priority but don't take actionable steps in their everyday work.

At its best, security is a sentinel, invisible and continuous, watching over data and alerting you to threats, breaches, and incidents. When you create and enforce security policies, you begin to manage the risk that is already threatening your company's products, APIs, and services — whether you know it or not. It's become apparent that tech companies must secure every component of their systems.

Application programming interfaces (APIs) are an often overlooked attack vector, although this technology is how most modern applications transfer data to external users and internally between services. Most companies have adopted software architectures such as microservices, which emphasize distributed systems, and tools like containers, which make applications more ephemeral in nature. Serverless, a fully managed method of writing functionality and paying only for execution time, continues to increase in adoption and relies heavily on APIs.

The short-lived and dispersed nature of modern applications makes traditional security practices — much of which were manual — ineffective. Instead, organizations must adopt modern security practices and automation to secure APIs with appropriate techniques, catch security incidents before they become critical, and alert appropriate engineers in as close to real-time as possible.

Much of modern security moves away from the manual work of traditional processes and adopts automation for continuous testing, scanning, monitoring, and alerting. This doesn't mean humans are removed from security; instead, they have improved tools to discover and remedy vulnerabilities in near real-time.

About This Book

This book is a high-level introduction to the key concepts of API security and DevSecOps. It's meant to be an amuse-bouche in your security journey and empower you with the knowledge to make decisions as you determine security practices for your engineering organization.

As you read this book, you may find your interest piqued over specific security attack vectors or modern approaches to securing systems. That's great! I encourage you to seek out additional information from a plethora of resources on modern API security, DevSecOps, and application security.

In addition to a number of automation tools, Data Theorem offers regular webinars and resources on API security to empower you to make the best decisions for securing your applications and systems.

Icons Used in This Book

This book uses the following icons to call your attention to can't-miss information.



TECHNICAL
STUFF

I use this icon to introduce something that's particularly technical in nature.



TIP

Don't miss the information marked with the Tip icon — it can make your life easier.



WARNING

When I need to emphasize a point that can help you avoid potential pitfalls leading to bad consequences, you'll see this icon next to the paragraph.



REMEMBER

This icon identifies material that's worth committing to memory.

- » Understanding API basics
- » Designing modern APIs
- » Managing and documenting APIs

Chapter 1

Understanding APIs

An application programming interface (API) is a data transfer approach that enables services within your application to talk to other applications (or other services within your system). Essentially, it enables Service A to talk to Service B in a uniform way.

You benefit from APIs on a daily basis without even thinking about them. Every time you sign on to your favorite social media platform or shop on Amazon, the application is utilizing APIs to fetch and update the data with which you're interacting. APIs come in three forms:

- » **Private:** For you and you alone. These private APIs are often publicly accessible on the Internet, but not intended for public consumption. For example, Apple's API has hundreds of private endpoints for iOS, which are intended for internal use only.
- » **Partnered:** Exposed to specific external users. Two banks might expose APIs to each other, such as when communicating wire transfer status where a web UI is not needed, but rather a status code, such as "200."
- » **Public:** Open to anyone. Twitter is a good example. It has hundreds of APIs that developers can use in their own apps at any time.

APIs are a powerful way to enhance your application and expose services to external parties. APIs are becoming products on their own because the appetite and economies for data continue to expand across every industry.

Digging into API Architecture

APIs expose back end data to specific services and users. Architecting an API depends heavily on the audience. Is the API private and designed for services within your system? Or is it public-facing and exposed to anyone?

Most public APIs are designed to be consumed by developers or other apps. For example, Amazon uses APIs to connect services within the company with others. The user service may use an API to communicate with the payment processing service. However, tech companies also expose APIs to developers to monetize their applications through in-app advertising and purchasing (in addition to other use cases).

Addressing HTTP

APIs follow the standards of hypertext transfer protocol (HTTP) — the protocol that determines how messages are formatted and sent over the Internet — to pass information back and forth. HTTP usually uses four actions (sometimes referred to as methods or verbs): GET, POST, PUT, and DELETE. An additional method, called PATCH, mimics PUT but applies only the delta (or difference) of a resource.

HTTP is stateless, which means every instruction is executed in isolation. In other words, a command has no knowledge of other commands.

Executing actions

The HTTP methods (or actions) listed in the preceding section are dependent on the scope you provide. For example, if you ask a service to return all users, you implement an API to call GET and supply a scope of all users. If you want to retrieve a user, you call GET and supply a scope of a single resource, in this case the user's database ID or other identifier.

Managing APIs

Just like any code in your system, APIs require constant management. The interface itself unlocks a massive amount of value for your organization in terms of performant access to data as well as opportunities for monetization. Managing APIs goes beyond maintaining the code in your system. Securing APIs includes controlling access, monitoring use, securing transactions, and analyzing opportunities for improvement.



REMEMBER

APIs should be designed with your customer in mind and allow for efficient transfer of data, taking advantage of the protocols that govern the Internet, like HTTP.

Modernizing your systems

Cloud native applications — a term for applications designed to run in the cloud — are driven by microservices and APIs. As cloud adoption continues to increase and companies take advantage of the infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) options provided by cloud providers, the need for well-designed and secure APIs will increase exponentially.



TIP

The separation of logic provided by a decoupled architecture supported by APIs is currently unmatched. Modernizing your entire system should include a strategy for writing, maintaining, and improving APIs.

Communicating with protocols

APIs use specific URLs to tell applications which action to use and include any required information. Your user service likely will have dozens of APIs, but focus on these:

- » The API `/users` would GET all users, returning a list of every user in the database.
- » The API `/users/10` would GET the user with an ID of 10, returning the data of that specific user.
- » The API to retrieve a user by another identifier might look like `/users?name='emily'`. That API might return any user with a name including “emily.”

Architecting Modern APIs

APIs allow different engineering teams to select diverse languages, technologies, and tools. Additionally, APIs create a communication path without direct linking, shared memory, or shared data. If architected well, APIs offer the following benefits.

RESTing on your service layer

The modern standard for APIs is REST — Representational State Transfer. The RESTful approach takes advantage of the HTTP actions of GET, PUT (or PATCH), POST, and DELETE. REST emphasizes that all actions should be condensed to those methods.

A new API and querying language, GraphQL, is growing in popularity because of its flexible approach. Unlike other solutions, GraphQL allows the client to determine how data is structured and returned, limiting excessive data transfers.

Separating concerns

One of the benefits of APIs is the separation of concerns between your application's logic and the external communication layer. Users don't have direct access to your business logic. Instead, they access information by requesting it via the API and receiving a response from your application.

Testing code

Testing APIs, especially their security, can be challenging. I suggest you start with the following questions:

- » Does the function work as expected?
- » What happens when a user inputs unexpected parameters?
- » Can the API endpoints tolerate intense user load?
- » Does the API work with any kind of client?

Ideally, all your reading and writing functionality will pass through an API, thus limiting the paths through which data can be accessed and changed. This single interface then enables you to secure it appropriately. This approach is significantly more beneficial if you have regulatory or compliance concerns.

Testing APIs, just as with other code, should take place in a variety of environments that enable developers to slowly increase resources and data to isolate potential issues in their code. Developers should write automated tests as they write their APIs and store the test code in the same source code repository that your entire engineering team utilizes.



TIP

When all code is in one place, every engineer can be empowered to contribute to its maintenance and improvement.

Finally, consider abnormal user behavior in your testing strategy. A test that confirms expected functionality given expected inputs is called a *happy path* test. A test that looks at behavior when unexpected parameters are used is called a *sad path* test. Sad path tests are critical to ensuring that users are limited in their ability to interact with your API, thus preventing hackers from taking advantage of vulnerabilities.

Documenting APIs

APIs should be as consistent as possible to decrease confusion. They should also be well-documented — especially if exposed to external users. Software development kits (SDKs) are sets of tools — typically accompanying public APIs — released by organizations with public APIs to help developers get started using an API.

Ideally, your APIs will be documented with information pertinent to the user but also future developers. API specifications like OpenAPI and Swagger are crucial. The questions you want to answer in documentation are:

- »» What is the resource it accesses? For example, users are a resource.
- »» What are the endpoints and methods?
- »» What parameters does the endpoint expect?
- »» What's an example of a successful request for this API? Include a schema in your example.
- »» What response is expected? This should include details of the schema and parameters that the user will receive accompanying a successful response.

Keeping your code agnostic

Perhaps the most beneficial aspect of APIs is that they are technology agnostic. APIs are a communication layer between two services. This approach does not require the services to have anything in common. For example, one service may be implemented in the language Go and another in Python. With a well-designed API, the services communicate flawlessly.



REMEMBER

The power this capability unlocks for engineering teams is incredible. It means you can utilize different languages and technologies to take advantage of specific advantages without taking into account the impact to your entire system. Engineering teams can scale and evolve to specialize in different tools and still interact with other services internal to the application. This layer, when secured properly, protects the business logic of your application from meddling.

Working with Legacy APIs

The term *legacy API* can refer to two things: legacy (essentially old or unmaintained) code that is wrapped in an API or an API that is, well, mature. In the case of the former, an API serves as a way of exposing the legacy logic to users in a modern way. This can be a great way of allowing users (or other services) to access old or deprecated features. The latter is often an API that utilizes an older method of API design like SOAP or formats the data in a traditional structure like Extensible Markup Language (XML) rather than the modern JavaScript Object Notation (JSON).



TIP

Many codebases have an ancient piece of logic that drives a key piece of functionality. This logic typically falls into two types: logic that works but no one knows why, and logic written in a language no longer supported by the team. Wrapping that legacy code in an API is a great way to manage access to the functionality without opening the code to problematic changes.

- » Utilizing the competitive advantage of APIs
- » Taking advantage of modern infrastructure
- » Digging into microservices, serverless, and OSS

Chapter 2

Growing the API Economy

The rapid iteration of modern software delivery life cycles requires engineering teams to innovate quickly and bring new ideas to market before their competitors.

APIs are a way for you to gain market share from competitors and monetize your applications in new and exciting ways. Imagine you have the best data on restaurant reviews. You can create a website and a mobile app and try to get thousands of users to use your products while spending millions of dollars in advertising, or you can create an API and sell access to the data to Yelp, Google, Facebook, Trip Advisor, and so on.

Although the latter approach will not make you a household name, your data will be consumed by millions of people and you'll have only one product (an API) to maintain. This is an oversimplification of the real world, but the power of APIs as it pertains to growing business needs outweighs web apps and possibly even mobile.



TIP

The term *API economy* is a way of describing the market developing from an emphasis of APIs and API design in tech.

Competing in Your Market

An API enables you to connect users with logic in a quick and usually an insecure fashion. The API layer that sits between your customer and your application is a set of tools and protocols that pass information.

APIs have become ubiquitous and the new way for companies to take advantage of previously untapped business and monetization strategies (like charging for API calls). One of the most exciting competitive advantages of an API is that it is the best way to mobilize, process, and monetize large datasets.

Moving to the Cloud

Cloud providers like Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform (GCP) have played a key role in driving the API economy forward. For cloud platforms, high-quality, stable and secure APIs have become a way for users to have better management of their systems as well as quickly turn on new services or instantiate new resources.

Enabling Microservices

Microservices have eclipsed monolithic applications as the industry-recommended application architecture. Monoliths are typically designed in a three-layered stack with a user interface (UI), business logic, and data access layer sitting above the database. The layers are stacked upon each other. The UI only communicates with the business logic, which asks the data access layer for information and passes it back up the stack to the user.

In a *microservice* architecture, logic is organized into small, decoupled components. Each component can freely communicate with any other component — through an API. This layer is essential for ensuring services can be written in different languages and take advantage of tools incompatible with other components. Because the API design is standardized, the logic in each component is essentially tech agnostic.



REMEMBER

Although microservices can increase the general complexity of a system, they enable developers to take more ownership over specific services and maintain a codebase with cleaner separation of logic and responsibility.

Building a Single-Page App (SPA)

A single page-application (SPA) is a way of making a web app appear more responsive to the user by loading the application once. In the past, every event, or click, that the user initiated kicked off a new request to the server, which resulted in loading a new page.



WARNING

SPAs require you to make key architecture decisions at the start of your project. At no point in a SPA does the entire page get wiped away only to be reloaded. Instead, new components are loaded as required by accessing APIs within your system.

You use SPAs all the time when you go on Twitter or Facebook, as well as any time you check Gmail or get directions on Google Maps. Notice how you've never had to refresh the entire page on any of these pages. New information is loaded automatically without any end-user involvement.

Compartmentalized API calls consume fewer resources than rendering the entire page on the server and thus fuel the quick response time of a SPA. Requests made within the server are faster than the time it takes for the browser to request and receive from the server.

Utilizing Mobile Apps

Your APIs act as a glue and connect web apps, desktop apps, cloud resources, data, and mobile applications. Instead of building unique data access for each of these mediums, APIs can be called from anywhere in your system and access the same data securely. Thanks to sound API design, your mobile app acts as a single vehicle with which users may access your application logic.

Outside of data and user experience (UX) consistency, some of the most pressing concerns regarding mobile apps pertain to

performance. Users are unforgiving and expect quick response times on mobile apps.

The three main performance indicators to pay attention to are:



WARNING

» **Latency:** The time required for a request to make it from the API to the server.

For mobile, network connection and physical geography can have a massive impact on latency.

» **Response time:** The latency of your system added to the time required for the server to process a request.

» **Crashes:** If your mobile app crashes completely, users can't interact with your business, and you can't make money. Be sure to consider the device, browser, and operating system (OS) the client was using when the app crashed.



TIP

Having insight into your latency is critical to locating and improving performance issues in the back end of your application.

Not only will APIs improve your speed performance, RESTful APIs are cacheable. The client can store responses in a cache, which reduces the number of server requests required. Asking the server to do the bulk of the logic and rendering is more efficient than client-side execution with mobile clients. Finally, scalability is critical to mobile app development. Applications must evolve to sustain heavy user load at the same response rates.

Creating SDKs

Software development kits (SDKs) are collections of code, tools, and information required to build an application. SDKs take many forms but are often built for APIs so that organizations can showcase the power of their APIs through interesting use cases.

SDKs are key for API adoption and integration. If developers are your customers, APIs and SDKs should be central in your business planning. An SDK should include the following:

» **Documentation:** Instructions for how developers should implement the API. Include common "gotchas" that developers might run into and any other relevant information.

- » **Libraries and tools:** Libraries often include prebuilt classes or logic that will help developers as they build their apps.
- » **License:** Can the user build a proprietary application using the SDK? Be clear about your licensing expectations in the SDK.
- » **Sample code:** Ideally, your SDKs will target several languages and include sample code to enable adoption from developers with a wide variety of backgrounds and experience.

Consuming Open Source Software

Open source software (OSS) is typically free-to-use software developed by a community of engineers who contribute to projects about which they're passionate. OSS is a wonderful and powerful way to integrate prebuilt tools into your system without dedicating massive resources to building a custom tool. Incorporating new software into your system isn't free, however. Integration takes engineering hours, not to mention the resources required to maintain the software.



WARNING

OSS is available under dozens of licenses with various requirements and restrictions. Be sure to read the fine print if you plan to use OSS in a proprietary product or service.

One of the key components of OSS is that you can see the source code. Everything is open to the public and you can fork (copy) an OSS and customize it.

OSS provides tools to better manage and secure your APIs but also includes projects that have APIs available for developers to consume out of the box. For example, Swagger, an API management tool, offers many open source tools.

Scaling with Serverless

Scaling is a key concern for most tech companies and can be difficult to solve. How do you scale to meet demand without exhausting your resources? *Serverless* is a short way to describe a cloud computing service to manage resources and pay only for the resources you consume while running the functions of an application.

Serverless is a relatively new and powerful tool often included in offerings by cloud providers. Serverless is still executed by servers,

but you pay only for execution time. At times when your application experiences less load, you pay less. You supply a function and receive the response. Though you have no control over the execution environment, your code is automatically containerized and auto-scaled for you, so you pay only when your code is run. This is true power of cloud computing, versus virtual machines in the cloud.

Serverless is a uniquely useful for scaling applications because it simplifies capacity planning and maintenance is managed by the cloud provider.



TIP

Serverless can be used in conjunction with code released and run through traditional methods. Alternatively, applications can be written to be exclusively serverless applications — eschewing any need to provision servers.

Serverless is closely associated with function as a service (FaaS), a cloud service that executes a method without storing any data. A related concept is the serverless database, which applies the same pay-as-you-go model to data storage.

The main benefits of serverless is the ease of implementation and the cost — it's quick for engineers to deploy and you pay only for the resources you use. Serverless is extremely useful for autoscaling based on user load. In addition, serverless abstracts complex development because it limits the requirement to develop software using a specific framework to the serverless ecosystem your team selects. Developers do not need to consider things like multithreading. Serverless is fully managed. At its most basic implementation, all the developer does is upload the code for a function to the cloud provider. The function is then run when triggered by a predetermined event.



TECHNICAL
STUFF

In Amazon Web Service (AWS), serverless is referred to as AWS Lambda. Other cloud providers refer to the comparable service as Azure Functions, Google Cloud Functions, or IBM Cloud Functions.

As with any technology, serverless has its own security concerns, which simply need to be managed. Legacy tools, such as vulnerability scanning tools, aren't designed for serverless, making it more difficult to detect problems in functions. Furthermore, serverless is a prime platform for a denial of service (DOS) attack that triggers countless serverless events that use a constrained resource. This type of DOS has been referred to as a denial of wallet (DOW) attack because of the economic harm it imposes by forcing an organization to incur greater cloud costs.

- » Making the move to DevOps
- » Designing your system for security
- » Notifying engineers of security concerns

Chapter 3

Introducing DevOps

DevOps is a concept that evolved from Agile and seeks to eliminate the friction between developers and operations folks in traditional engineering teams. The philosophy emphasizes three priorities: people, process, and technology.

The priorities of DevOps occur in that order for a reason. The first priority, *people*, refers to the cultural change of your engineering team to one that encourages engineers to collaborate, trust their colleagues, learn from failure, and take ownership over their work.

The next priority of DevOps, *process*, focuses on creating systems in which your engineers can thrive. If an engineer makes a mistake, it's because the processes you utilize need to be improved and refined to reduce the possibility of human error as a contributing factor to failure.

Finally, *technology* ensures that your team uses tools that optimize productivity and accelerate the team's velocity.

Transitioning to DevOps

Transitioning to DevOps is not an overnight process. Applying the methodology requires concerted effort to unify people and undo some of the patterns formed in the past. One of the biggest

challenges in a transition to DevOps entails convincing waterfall engineers that the initial investment of time will pay dividends in faster development, as well as creating a groundswell of excitement and adoption at the engineering level.



WARNING

If engineers don't understand the benefits of DevOps for them and their daily work, a transition to a DevOps culture will fail.

If applied well, the concepts and principles of DevOps will energize your team and create a shared learning culture that embraces a growth mindset. Instead of looking at mistakes as failures, it seeks to learn from error rather than avoid it at all costs.



REMEMBER

This attitude is particularly important in security. If engineers feel they might be fired or otherwise punished for a security lapse, they may be more likely to avoid being forthright about the factors that led to the event. A DevOps environment recognizes that security is a component of the process. Mistakes will be made and vulnerabilities will be released. DevSecOps seeks to introduce security early in the development process and rapidly iterate to reduce incidents in the future.



TIP

Though it's called DevOps and focuses on developers and operations, the methodology applies to any team involved in the software development life cycle. The purpose is to create an environment where every engineer can thrive — regardless of their background or expertise. By inviting everyone to the table during the planning phase, you provide space for each engineer to highlight particular concerns and develop shared approaches toward solving those problems, long before code is written.

Applying Security to DevOps

Security has a key place in DevOps and is often referred to as DevSecOps. In traditional engineering, security was an afterthought. However, as security became more important, the industry overcorrected and development teams felt stalled to the point that they'd rather avoid security than embrace it. This approach put security in the impossible situation of preventing developers from shipping their insecure code. You can imagine the frustration and animosity that emerged between teams.

DevSecOps seeks to shift security left in the process and infuses security into the development process in a manner similar to performance testing, quality assurance (QA) testing, and user acceptance testing. When security is invisible but planned for and managed early in the process, security engineers are less likely to become gatekeepers who prevent developers from releasing their software. DevSecOps emphasizes preventative process over manual or reactive work (which requires more security engineers). Many security teams struggle with this issue. However, “fewer people, more processes” is an approach developers can embrace.



TIP

The worst thing you can do in DevSecOps is force developers to act like security people. Features always take priority, but DevSecOps identifies solutions in the process that developers can embrace quickly and efficiently. (No, security training is not one of them.)

Rather than being referees, security professionals should integrate security software into the system and ensure that appropriate monitoring is implemented so developers are immediately alerted to issues and provided key information needed to remedy security concerns.

Understanding the Software Delivery Life Cycle

In traditional engineering organizations, the software delivery life cycle was linear, with one phase occurring after the previous one until software was released to customers. The traditional phases were typically:

- » **Planning:** Decide which features should be included.
- » **Designing:** Make key architecture decisions for the project.
- » **Developing:** Write the code.
- » **Testing:** Ensure the code works as expected.
- » **Deploying:** Release the new functionality.

Security used to fit somewhere between testing and deploying. Thus, defects would have to be kicked all the way back to the developing phase. Large defects might even be returned to the design phase. This method slowed development significantly and created plenty of frustration.



TIP

DevOps seeks to connect the ends of the software development life cycle and create a continuous cycle of feedback and improvement. By treating the process as a circular and never-ending practice of software development, you can focus on small (and secure) changes deployed frequently. If a security tool has identified a top priority one (P1) security issue that would otherwise leak confidential data to unauthorized sources, the developer can treat the experience as an iteration in the development cycle rather than a devastating failure.

Shifting Security Left

DevSecOps seeks to move security considerations to the development phase. This approach connects the ends of the traditional software development life cycle to make a continuous circuit of rapid iteration.

In a DevOps culture, continuous iteration is an important aspect of a healthy engineering team. That continuous improvement includes encouraging engineers to push code — without fear of negative repercussions — and trust the process. Security is a prime area in which to provide this type of opportunity.



REMEMBER

Shifting security left in your delivery of software entails baking security into the right spots, but not every spot.

Architecting for Security

Security used to be a mostly manual process. It involved stagnant security processes and long checklists. As your systems become more distributed and you take advantage of decoupled microservices, continuous integration and delivery (CI/CD), and ephemeral containers, your security practices must evolve as well. Modern security takes advantage of automation to track components across your entire system.

Automating security covers nearly every area of your system. Here are a few areas to pay special attention to:

- » **Authentication:** Provide the appropriate amount of privilege to reduce opportunities for unauthorized access.

- » **Authorization:** Ensure authorized users have the appropriate rights and privileges for a particular function, assets, command, or data source.
- » **Encryption:** As information flows within your system, ensure everything is encrypted appropriately with cryptography suites that meet regulatory compliance standards.
- » **Availability:** Apps, systems, and assets always need to be up and running reliably.
- » **Auditing:** Ensure logs are always on and accessible because the identification of a data breach often relies on a few audit categories.

Monitoring and Alerting

You hear the term *continuous improvement* often in DevOps circles. The DevOps methodology doesn't seek to avoid failure at all costs but instead prepare for (and learn from) failure when it occurs.

Apply continuous improvement to your security practices, monitoring, and alerting. This approach helps you incrementally improve from wherever you are in your DevSecOps adoption.



TIP

It's better to start at nothing and slowly add security checks and DevOps practices than to avoid adopting modern approaches all together. If you're just getting started, you may feel overwhelmed by the number of security aspects to consider. Accept where you are currently and begin by improving a single component. Then approach the next, and the next, until you feel adequately protected against common vulnerabilities and attacks.

I encourage you to adopt a monitoring approach that tracks your security across all externally facing applications and infrastructure.

You need automation tools to tell you when security and privacy gaps occur and when (in)security becomes a bottleneck in your pipeline. Metrics to track include:

- » Number of sensitive data assets publicly exposed on the Internet
- » Number of data assets accessible to authenticated but unauthorized entities

- » Percentage of apps and packets running without any in-transit encryption
- » Amount of downtime for critical apps and systems



Implement security across the key parts of your process to alert you as soon as security issues appear.

WARNING



TIP

When alerting engineers to potential attacks, avoid “alert fatigue” by ensuring that engineers are alerted only about immediate and fixable issues affecting data. For more standard discoveries of vulnerabilities, rely on telemetry, monitoring, and visualizations to drive home the importance of security fixes.

- » Diving into traditional security practices
- » Understanding two types of APIs
- » Evaluating the risks

Chapter 4

Securing APIs with Traditional Approaches

Securing application programming interfaces (APIs) is a never-ending and often uphill battle. APIs, for all their benefits, open new opportunities for unauthorized sources to attack your data via vulnerabilities and exposures. In its report, “API Security: What You Need to Do to Protect Your APIs,” Gartner estimates that “by 2021, 90% of web-enabled applications will have more surface area for attack in the form of exposed APIs rather than the UI, up from 40% in 2019.”

Traditionally, APIs were protected by API gateways and content delivery networks (CDNs). Although these approaches hedge against most brute-force attacks, a more nuanced and sophisticated attack may be able to breach the protection.

The traditional approaches to API security are littered with limitations. This chapter introduces the common traditional API security strategies companies used to take and the tradeoffs of legacy approaches.

Two Types of APIs

Companies use two types of APIs: customer-facing APIs and internal APIs. The external type is exposed to all Internet traffic. External developers can make requests and receive responses to the customer-facing API and take advantage of the functionality and data of an application while bypassing the user interface. Internal APIs, however, are exposed only to intra-application traffic. For example, the `Cart` service may call the `User` service via an internal API during the checkout process of an e-commerce application.

Productizing customer-facing APIs

Customer-facing APIs are stable, visible, and targeted at a developer audience. This open and accessible API is designed to be utilized by developers outside the publisher's organization. The goal is to leverage a community of technical audiences and increase usage of the organization's data and services. Occasionally API publishers encourage developers to create applications that expand on the core business.

The largest challenge of customer-facing APIs is ensuring that the layer is secure every second of every day.

Internalizing APIs

While internal developers may call external-facing APIs, the more common API calls between services within an application are for internal use only. These APIs are not accessible by developers outside the publishing organization.



REMEMBER

Changes are made constantly — as fast as the company's DevOps and CI/CD practices allow. The speed of change makes mistakes, errors, and security oversights more likely. These APIs and changes are less visible, which leaves companies at risk of discovering vulnerabilities only after a hacker has breached private data.

Shadow APIs

Data leaks through all APIs, but you can't protect what you don't know. The moment those APIs are exposed on the Internet, they create attack vectors for hackers to extract data.

When new Lambda-based microservices (serverless apps) are created for others to use, programs generally interact with these functions through Amazon's API gateway. However, certain programming frameworks create APIs that completely bypass any gateway. These are called *shadow APIs* because they operate undetected for long periods before security teams can discover them and inspect their usage. If an API is not well designed, all the data that passes through it is at risk. Most shadow APIs were created as utilities, though some have caused financial and brand damage to companies when exploited by attackers. The concern is that an application or microservice that is pushed quickly, without normal security assurances, might have vulnerabilities.

The absence of authentication or authorization to the API can permit users to take advantage of a serverless function to extract data in ways never intended by the application developer. Two factors exacerbate these problems in modern applications. One is agile development, and the other is the automated nature of scaling serverless resources.

Agile affects the rate of change, and serverless automation affects the scale of exposure. Both have the potential to magnify the effect of a simple mistake.

Securing APIs with Legacy Approaches

The traditional approaches to API security, although helpful at reducing the attack surface, are too slow to keep up with the ever-evolving tactics of bad actors. Decryption algorithms are constantly being improved and released, opening vulnerabilities in technology across the web. Hackers aren't going to stop. Although the key to modern detection is to take advantage of automated security tools that discover and respond to threats within seconds, it's important to look back at traditional security strategies and understand the tradeoffs and limitations.

(Pen)testing manually

A penetration test (*pentest* for short) is an approved manual attempt to infiltrate a system. The outcomes of a pentest are identified security weaknesses as well as the features and data that are well protected from hackers. Typically, an external party

specializes in these tests. The tester supplies the company with a verbose risk assessment that suggests mitigation options for vulnerabilities discovered during the pentest.



TIP

Pentesting consultants are typically expensive and can be an extremely slow approach. Bug bounties are a way of consistently encouraging white hat hackers and developers to report vulnerabilities they discover. Both of these styles of pentest result in inconsistent reports of vulnerabilities because much depends on who is initiating the test, their area of focus, and the scope of the test. Ultimately, although pentests can be helpful, they lack the speed and scalability needed in DevOps-enabled engineering teams and processes.

Using an API gateway

An API gateway was a traditionally effective way of enforcing security settings — such as authentication and authorization — across all APIs. The gateway serves as a single source of entry for all API requests.

This traditional approach requires developers opt into the gateway when they deploy new APIs. This is acceptable for external, customer-facing APIs but less realistic for internal APIs that connect microservices. Developers want to move fast, but an API gateway quickly becomes an obstacle.

Gateways often add performance latency and can potentially hinder your ability to autoscale. Additionally, it's a restrictive architecture for organizations building cloud native applications or looking to move to the cloud through a hybrid approach. Applications designed for public clouds will default to the native gateway or choose to use none.

As you can imagine, if all traffic must flow through the gateway, it becomes a single point of failure unless it's deployed in a cluster, which adds complexity. Ultimately, the additional cost and limitations of an API gateway make it useful for some scenarios (particularly customer-facing APIs) but does not cover every use case.

Creating web application firewalls

A web application firewall (WAF) filters and observes HTTP traffic as requests come into the application. Though not designed to be a one-size-fits-all solution, WAFs add protection against

specific types of attacks: cross-site scripting (XSS), SQL injection, file inclusion, and cross-site forgery.

WAFs are a type of reverse proxy, a sentinel that fields attacks before traffic reaches the server. Specific policies, put in place by the organization, determine the behavior of the WAF and how it filters traffic. For example, in the case of a distributed denial of service (DDoS) attack, the WAF can introduce rate limiting quickly.

Dealing with the Consequences of Traditional API Security

Security breaches of APIs are dominant because APIs are such a ubiquitous method of passing information. Data breaches come in many forms, but they almost always affect company revenue and reputation.



REMEMBER

New API breaches and vulnerabilities are discovered almost daily. T-Mobile, Symantec, McDonald's, Instagram, Salesforce.com, and Venmo have all experienced major breaches in the past few years because of API insecurity. If some of the largest companies in the world have insecurities throughout their codebase of APIs and services, you can bet you do too.

- » Integrating security into CI/CD
- » Securing APIs in production
- » Automating policy enforcement

Chapter 5

Modern Approaches to Securing APIs

As software evolves, so do the security vulnerabilities. Hackers are constantly finding new ways to take advantage of weaknesses in your systems. Although traditional security practices still have their uses, updating your security strategies should be a never-ending iterative process for you and your team.

Catching Bugs Early via CI/CD

Continuous integration and continuous delivery (CI/CD) — the “D” sometimes stands for deployment — is a way of automating your deployment process. You can automate portions of the release or the entire deployment from the development environment all the way to production. Tests and security checks can be injected into any phase of the pipeline.

Injecting checks

The real benefit of a CI/CD pipeline is tracking changes end-to-end and injecting checks to ensure you deliver reliable software to your customers. How you structure your CI/CD pipeline, however, is completely up to you. If you're just getting started, I encourage you to take one piece at a time and think of the process as iterative and additive.

With CI/CD, a deployment will be triggered as soon as your developers commit new code to the master branch in your source control. This doesn't mean that the build will be released to customers (as in continuous deployment), but it will kick off the checks and tests that run automatically in your pipeline. CI/CD pipelines are often designed to halt just before releasing to production and await a manual one-step release by one of a few specified approvers.

Although any security check helps harden your system against vulnerabilities and attack vectors, you should add a few important components to your CI/CD pipeline. Be sure to use vulnerability scans for known security issues. These look for publicly identified and well-known hacking vectors.

Add detections for nonapproved or outdated frameworks. There are open source tools available to look for outdated dependencies, such as <https://github.com/dylang/npm-check>.

Injecting checks that look at nonapproved or dangerous functions, classes, or APIs can identify security vulnerabilities early in the release process and prevent them from being released to customers.

Looking for dangerous code

Examples of dangerous code are vulnerable XML parsing APIs available in the Java standard library and string concatenation for building SQL queries, which has the potential to lead to SQL injection, a common hacking technique. In addition, certain functions are blacklisted for particular languages. Data Theorem produces tools that identify bugs and other issues in the CI/CD process for languages like Python (<https://github.com/datatheorem/flake8-alfred>).

Your goal should be to use approved frameworks and tools across all the software in your organization. You may find through dedicated scanning that you'll develop a long list of security vulnerabilities

in APIs accessible across the entire organization. Developers may feel this will slow development or limit experimentation, but these approaches can prevent catastrophic error and often speed development after an initial investment of time and resources.

Testing in CI/CD

Any CI/CD pipeline should have a robust test suite to support your team's software delivery and check against security, reliability, functionality, visual changes, and more. It is absolutely critical, and any attempt to create a CI/CD process without an automated test suite will fail.

Automated tests are faster and more reliable than manual testing. This method limits the chances that human error will miss critical bugs and vulnerabilities. Perhaps above all else, testing and security checks in CI/CD allow your team to move faster. Be sure to add unit tests with scenarios for detecting critical code paths with a bad configuration or bad logic. These include:

- » Disabled authentication
- » Unenforced authorization
- » APIs that respond with payloads despite unexpected or bad parameters



WARNING

APIs should always respond with error codes with receiving requests that include unexpected values and parameters. This is a common vector used by hackers to gain access to a database.

Documenting APIs

Secure APIs start with documentation based on an API specification. An API spec is essentially a blueprint for every API in your system and can help you avoid key architectural design flaws before the development phase. The industry standard is OpenAPI 3.0 (<https://github.com/OAI/OpenAPI-Specification/blob/master/3.0.0.md>).

Because APIs are the glue that hold together the services in your applications, it's critical that they are designed to integrate well with every service they will interact with. That requires some

forethought through a spec, which will guide developers and designers as they expand your suite of APIs.

Validating the Security of Your APIs in Production

Security checks don't stop once the CI/CD pipeline has deployed a release into production. Due to environmental differences and unexpected complexities of a live system, you should always stress-test your system in production environments that see live traffic from customers.

Live testing in production can catch potential differences in how the code behaves or is configured in different environments from development, testing, staging, and production.

When verifying the configuration of the production environment, be sure to confirm key attack vectors are not exploitable for your APIs and services.

SSL/TLS encryption should be at the top of your list. This ensures your HTTP connection is secure and guarded against interceptions of data and messages during transfers. The next priority is to set up rate limiting safeguards and protect against distributed denial of service (DDoS) attacks. These precautions restrict traffic to your system and prevent it from being overwhelmed.

Auditing plays an important role in your system by tracking when data and information is accessed as well as application usage. Logs monitor the data accessed and permit you and your team to detect and prevent unauthorized access to data. Auditing is critical for organizations that must show compliance with security policies to larger regulatory bodies. Activity to log and audit includes:

- »» Creation and deletion of users
- »» Permissions changes
- »» Password changes
- »» User logins
- »» Session timeouts and terminations

Finally, your configuration should take into account autoscaling controls that will prevent denial of wallet (DoW) attacks. In this relatively new attack vector, hackers attempt to use customer-facing business logic to trick the system into thinking a spike of users are hitting the site. The goal is to drive up infrastructure costs and cause the victim to overuse resources in the cloud. Watch account logins and creations closely as well as unusual spikes in user-driven actions.

Auto-Enforcing Policies

Active protection of your system without human intervention is the only way to achieve real-time enforcement of security policies and protect against attacks as they occur. Ideally, your system automatically discovers and inspects API security vulnerabilities while humans analyze the situational risk and prioritize the work.



TIP

The DevSecOps team or specialists should make ongoing risk-based decisions on when certain issues should be mitigated and where in the CI/CD pipeline those concerns should be tested. However, in key scenarios, active protection is required to prevent a security incident from occurring or escalating. Your system should utilize tools to add this layer of protection without requiring humans to be involved.

For example, imagine a company owns a customer-facing API that transfers sensitive data that is required to have regulatory oversight. If the API received a new configuration to reduce the encryption to TLS 1.0 — which is less secure than most compliance standards — and had its authentication disabled for testing, the active protection would step in to override the configuration change. Though the change may seem harmless at first glance, the automated protection would ensure the organization is safe and the data is protected — without involving a human at all.

Tracking Changes

Shadow APIs are a threat to the security of your system. You simply can't protect against something you don't know exists. These APIs are hidden, forgotten, and often long-lived, which puts your applications at risk of hacking through a vulnerable endpoint.



TIP

Cloud providers and cloud security vendors supply monitoring tools that provide insights and telemetry into your systems. One of the best benefits is these tools detect new APIs and microservices. Using these tools to create an up-to-date and accurate inventory of the APIs currently deployed to your system helps you identify rogue shadow APIs and either harden them with security checks or delete them.

Closing the Loop

When API vulnerabilities are discovered, it's important that the security system sends alerts to your security events and information management system (SIEM) and log aggregators.



TIP

If an issue is critical and threatens the system imminently, the vulnerability should alert the engineer on call and trigger an incident response to ensure the matter is handed as soon as possible. In the event of a critical vulnerability or incident, be sure to schedule and hold a post-incident review and collect action items for future work that addresses the contributing factors. Your security system should auto-generate tickets or work items in your ticket tracking system (such as JIRA) that will be prioritized in your backlog. Each work item should be prioritized and assigned to an engineering team to address the issue.

IN THIS CHAPTER

- » Applying security to key phases of the development life cycle
- » Securing CI/CD pipelines
- » Responding to security incidents

Chapter 6

Benefitting from an API Security Framework

If you're overwhelmed by how many security threats loom and worried your already overworked team of engineers won't be able to handle API security on top of meeting the demands of customers, don't fret. You can build a DevSecOps process that will improve your security and enable your team to rapidly iterate — constantly improving your systems. An API security framework starts with three fundamental parts:

- » Creating security policies
- » Enforcing policies
- » Managing risk

Streamlining Security Processes

Security processes used to be manual, labor-intensive, and slow. Because of modern architecture, such as distributed systems and ephemeral containers, it's critical your security model is able to adapt and respond within minutes, if not seconds.

Your security processes and tooling should constantly analyze your cloud environments, API gateways, logs, web and mobile applications, as well as source code for changes that might create vulnerabilities. Once you're able to identify the changes to your system, you can confirm that authentication, authorization, encryption, access, and verification meet the standards you've set for your team.



TIP

Only with a clear and holistic view of your system's security can you manage risk and enforce security policies. Without telemetry and analytics, every security decision will be a shot in the dark and you'll miss critical holes in your API security. When a security incident occurs, you'll be able to leverage that data to appropriately identify the components of the system affected and the contributing factors that led to the event.

Security at Key Stages of Development

The differentiator between engineering teams who release highly secure APIs and those who don't comes down to discovering and remedying security vulnerabilities at the right time in the process.

The software development life cycle (SDLC) is a series of stages from ideation to production that includes planning, architecting, developing, testing, and releasing code. Security must be a consideration at key phases in the process. If you catch an issue early in the SDLC, you can fix it quickly without significant engineering resources. A security vulnerability in production not only opens you up to a breach, it may require substantial work to remedy.

Releasing with CI/CD Pipelines

Continuous integration and continuous delivery (CI/CD) is the ideal process for releasing cloud native applications — systems built for cloud infrastructure and tooling. If you're unfamiliar, CI/CD is a pathway designed for developers to test and release their code independently (without assistance from operations engineers).



The “D” in CI/CD typically stands for *delivery*, but sometimes *deployment*. The difference is that delivery simply means developers can automate the release of new features through the testing environments but the pipeline halts just before releasing the new code to customers. It requires human approval, typically by a simple click of a button, to complete the deployment process. Continuous deployment, on the other hand, is automated entirely. If code is checked into the master branch, and passes all tests and security gates, it is released to your production environment and is accessible by customers. Continuous deployment, as you might imagine, requires significant investment in thorough testing.

Security can be built into your CI/CD pipelines by checking for security automatically at key stages. This includes:

- » When code is deployed to every pre-release environment (development, testing, staging)
- » When features are released to customers in the production environment

Checking security with a CI/CD pipeline is an improvement on manual security processes instituted at the very end of the software development life cycle, just before release. CI/CD tests will alert developers immediately and allow for remediation quickly and without a massive engineering effort.

Tracking Bugs

API tests typically look at actions, arguments, and responses to ensure that the functionality works as expected. Additional tests might explore error handling and performance under intense customer usage. Every API test suite should also include security gates to locate potential threats. When data exposure is found, tracking that issue (and conveying the appropriate context and information to your engineering team) becomes critical to remediation.

If you don't already have a centralized issue tracking process and tool, start exploring potential solutions. Look for tools that use automation to add and track bugs or make it easy to create new issues. Features to consider include:

- »» Cloud integration
- »» Real-time notifications
- »» High-quality reporting
- »» Easy-to-use user interface
- »» Bug duplication detection

The main benefit of a bug tracking system is to create a single source of truth for your engineers when they submit and track bugs. If you evaluate your security risks and are comfortable delaying certain fixes, it's fine to consider some bugs to be a form of technical debt. But be sure to regularly prioritize the list so engineers can quickly select the next most critical bug to patch. The easier you can make security for your engineers, the more success you'll have in maintaining secure systems.

Accelerating Incident Response

Though not every incident is as dramatic as a British Airways or Target retail exposure, where both have been fined and losses are exceeding \$200 million, the number of security incidents has increased across every system simply because of the increased complexity. There are more areas of an application open to attack and data breach — including APIs.



TIP

Rather than add more security professionals, you need to begin to rely on automation and tooling to provide insights into threats, alert you about incidents or attacks in near real-time, and assist you by taking action without human intervention to mitigate the attack and limit exposure. This security automation and incident response doesn't eliminate the need for human intervention, a topic covered in Chapter 7.

- » Calculating security risks
- » Continuously testing for security
- » Streamlining your incident response process

Chapter 7

Automating Your API Security Framework: Introducing DevSecOps

You don't need to double your engineering staff to implement good API security practices. Instead of working harder or throwing people at the problem, think more strategically about your approach. Automation empowers you to create a four-point security framework:



TIP

- » Continuously discover changes to API specifications.
- » Analyze your API security constantly.
- » Manage risk through setting and enforcing security policies.
- » Alert on security incidents and implement remediation.

Evaluating Risk

Just as DevOps was created to remove the barriers between developers and operations folks, DevSecOps exists as a reminder that security is an essential part of DevOps and should be injected into key phases of the software development life cycle.

Verifying API security early in the process secures the system and reduces the cost of fixing issues discovered. Security threats discovered just before, or after, a release to customers take significantly more engineering resources to assuage.

Ask yourself the following questions:

- »» What areas of your system are exposed?
- »» What is your plan to reduce the attack surface and limit the ways hackers can gain access to your system?
- »» How will you continuously verify security?
- »» What risks are you willing to take because of the cost of prevention?

Being 100 percent secure is impossible. You face too many “unknown unknowns,” constant changes to systems, and ever-evolving threats to be able to harden yourself to every possible exposure. Instead, focus on the most critical aspects of your system, including customer data, regulation and compliance requirements, and the services that supply your business with the most revenue.



WARNING

Although business revenue is a critical aspect of evaluating security risk, so is reputation. Consumer trust is an elusive requirement for doing business. You ask your customers to trust you when they supply you with personally identifying information (PII). In turn, you watch over that data closely and protect customers from as much exposure as possible. If you fail to take the necessary precautions, you may lose more than this quarter’s revenue. You may lose customers for life.

Securing Continuously

Just as DevOps emphasizes continuous improvement (sometimes referred to as *kaizen*), DevSecOps underscores the importance of automating security across the entire development pipeline.

This continuous security increases your overall application security, improves your testing across all environments, reduces the cost to remediate when incidents occur, and increases collaboration among all members of your engineering team.

Testing

Your testing strategy should take into account your business goals and customers as well as any mandated compliance. If your security testing is currently manual, don't expect to have an airtight security test suite overnight. It'll be a slow process, but you have to start somewhere.

Testing for security should prioritize the areas of your apps most likely to be attacked. APIs are a key vector because it's data in transit and open to a variety of well-known threats.

Your test suite should include robust security tests to verify that components are protected against potential threats.

Your testing strategy and test code should be available to everyone on the team and held in a repository where changes will be tracked. Be sure to run the tests in every pre-release environment as you deploy the application. By the time your code reaches customers and new APIs are exposed to potential threats, you should feel confident you've taken the appropriate steps to protect your system.



REMEMBER

Consider security tools to monitor and detect security threats in your apps on a continuous basis.

Monitoring and scanning

Monitoring and scanning is an automated way of discovering new inventory (attack surfaces) and constantly assessing your application's resistance to known weaknesses and threats, typically in production.

Static analysis tools look at code to find glaring security issues and confirm that no confidential information, like secrets and keys, made it into source control.



TIP

Utilizing security scanners at a regular cadence is a good idea, but it's also important to scan your system whenever changes are made. If you bring in a new app, add API, switch vendors, or adopt a third-party or open source tool, scan the system. The scanner serves as a first line of defense against basic and sophisticated threats.

Managing Security Incidents

Reacting quickly to security attacks is key to catching the issue, eliminating the threat, and identifying the contributing factors to ensure appropriate work that prevents a future incident.

Unfortunately, security incidents aren't smooth, linear processes. Every major incident is chaotic and stressful. Incidents have five phases:

- » **Discovery:** There's a problem!
- » **Response:** Where does the threat originate? What services are affected?
- » **Restoration:** Security threats are mitigated and services are restored.
- » **Reflection:** What factors contributed to this incident?
- » **Preparation:** What work must be completed to avoid this event in the future?

Alerting developers

It's important to alert engineers to a potential security incident as soon as it's detected, but be careful not to over-alert. "Alert fatigue" occurs when engineers experience an overabundance of low-priority alerts and become immune to notifications — even when the priority is high.

Different security incidents have different priority levels. Authentication attacks, unauthorized access, and encryption issues are all high priority; you should alert and act immediately. Other attacks, such as probing and explorative attacks, are a lower priority.

Creating playbooks

A *playbook* is a set of rules or procedures that can be utilized by the first responders to a security incident. Typically, playbooks (sometimes referred to as *runbooks*) include at least four components:

- » An event or condition (such as a particular security incident) that initiates the use of the playbook.
- » Action steps to take in response to the event. This is the bulk of the playbook and provides instructions for what to do as well as suggestions for future automation.
- » Specific policies or regulation information relevant to the initiating event.
- » The desired outcome.

Tooling Your DevSecOps Practice

Securing your APIs with automation creates more secure systems and enables you to validate your API security at every stage of the process. I recommend you integrate with security tools for your CI/CD pipelines, mobile apps, and API security. Look for tools that integrate well with your environment, infrastructure, and other management systems.

Ideally, your security analysis will detect authentication issues, authorization gaps, outdated encryption, unvalidated parameters, and shadow APIs in serverless applications.



WARNING

I do not recommend developing your own security scanners and tools. Except in rare cases, you'll lack the expertise and focus to keep track of all vulnerabilities and potential threats. In addition, code has to be maintained, which means you're adding work for your engineers.



TIP

Plenty of third-party vendor tools, as well as open source solutions, help you manage security. Choose tools that work well with your current technology stack and infrastructure choices, address the security issues you're most concerned about, and have a user interface that's comfortable for your engineers.

- » Securing modern tools and architecture
- » Adopting DevSecOps principles

Chapter 8

Ten Takeaways

If you take nothing else from this book, please remember these ten tips on securing your APIs:

- » **RESTing on HTTP:** All APIs — GraphQL, REST, SOAP — utilize the HTTP actions of GET, POST, PUT, and DELETE.
- » **Migrating to the cloud:** The cloud provides endless opportunities for growth, often through APIs. The cloud-driven API economy provides more opportunities for security vulnerabilities — making APIs a critical piece of your system to secure.
- » **Securing APIs:** Authentication, authorization, encryption, availability, and auditing are the pillars of security. They should be applied to all your publicly facing APIs that could lead to data exposure on the open Internet.
- » **Adopting DevOps:** DevOps is a methodology intended to create high-velocity teams that emphasize collaboration, develop a learning culture, and share ownership. Securing your APIs isn't an overnight process and the DevOps principle of continuous improvement is something critical to your team's success in securing their applications.

- » **Shifting security left:** DevSecOps merges security with the fundamentals of DevOps and seeks to move security left in the software development life cycle, meaning you test and fix security early and often.
- » **Diving into security threats:** Your APIs face nearly endless security threats. Be sure to shield and validate API parameters, encrypt data in transit, authenticate and authorize users, remove identifying information from URLs, and utilize modern TLS cryptography.
- » **Moving beyond tradition:** Traditional API gateways and web application firewalls (WAFs) are useful but offer limited security defense against sophisticated attacks on distributed systems. Shadow APIs are a new class of real-time analyzers that continuously assess APIs, especially those built for the cloud.
- » **Authenticating users:** Always confirm users are who they appear to be through authentication. Then ensure those users have authorization to access specific data. Authentication without authorization leaves you open to security threats.
- » **Securing CI/CD:** Initiate security checks at every stage of your CI/CD release pipeline. If a developer checks code in or releases it to a new environment, confirm that the code and tools are secure.
- » **Responding to incidents:** Automation and telemetry help you catch security incidents as they're occurring and respond appropriately. Create playbooks to enable engineers to respond to incidents appropriately. Automating playbooks can lead to auto-remediation for critical areas of your APIs and applications.

Free API Attack Surface Calculation

<https://apicalculator.datatheorem.com/>

API Attack Surface Calculator

1 Web — 2 Mobile — 3 APIs — 4 Clouds — 5 Cloud Services — 6 Framework — 7 Compliance

Do you have public web apps with APIs?

Select one

Yes

No

BACK NEXT

Protect your APIs from data leakage

The short-lived and dispersed nature of modern applications makes traditional security practices — much of which were manual — ineffective. Instead, organizations must adopt modern security practices and automation to secure application programming interfaces (APIs) with appropriate techniques, catch security incidents before they become critical, and alert appropriate engineers in as close to real-time as possible. This book is a high-level introduction to the key concepts of API security and DevSecOps.

Inside...

- Secure serverless (Lambda, Azure/ Cloud Functions)
- Identify shadow APIs
- Leverage OSS and SDKs
- Automate API security for developers
- Respond to security breaches
- Benefit from microservices

data**th**esorem

Emily Freeman is a technologist and a storyteller who helps engineering teams improve their velocity. She is best known for her creative approach to identifying and solving the human challenges of tech. Emily manages modern operations advocacy at Microsoft and is the author of *DevOps For Dummies*.

Go to **Dummies.com**™
for videos, step-by-step photos,
how-to articles, or to shop!

ISBN: 978-1-119-63962-6
Not For Resale

for
dummies®
A Wiley Brand



Also available
as an e-book



9 781119 639626

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.